COMP0168 Machine Learning Seminar Lecture Notes

Melodie Li

UCL Department of Computer Science

Jan 2025

Contents

Chapter 1

Gaussian Processes

1.1 Gaussian Processes: Introduction

1.1.1 Problem Setting

Theorem 1.1.1 (Problem Setting for Gaussian Processes)**.** *Given a set of observations*

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

the objective is to find a distribution over functions $p(f)$ that explains the data. This corresponds to a probabilistic regression problem.

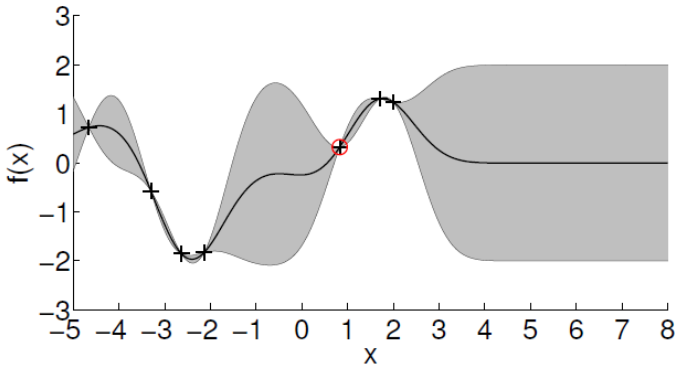


Figure 1.1: The figure illustrates the prior belief about the function $f(x)$. The shaded region represents the uncertainty, with a higher variance in areas without observations. The observed data points are marked as black crosses.

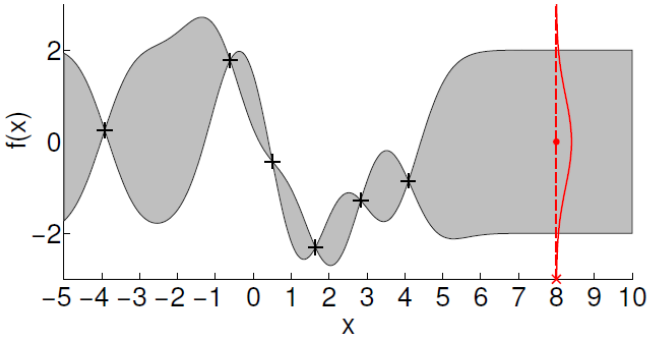


Figure 1.2: The figure shows the posterior distribution over $f(x)$ after observing data. The shaded region represents reduced uncertainty in regions with observations, and the predicted mean is shown as a solid black line.

1.1.2 Recap: Bayesian Linear Regression

Definition 1.1.2 (Prior)**.** The prior distribution on the parameter vector θ is given by:

$$p(\theta) = \mathcal{N}(m_0, S_0),$$

where m_0 is the prior mean, and S_0 is the prior covariance matrix. Normally it's Gaussian prior as a conjugate prior to allow close form computations.

Definition 1.1.3 (Likelihood)**.** The likelihood function for observations y given input x and parameters θ is also Gaussian by default:

$$p(y \mid x, \theta) = \mathcal{N}(y \mid \phi^\top(x)\theta, \sigma_n^2),$$

where $\phi(x)$ is the feature vector, and σ_n^2 is the noise variance. This implies the data model:

$$y = \phi^\top(x)\theta + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2).$$

where θ is treated as latent(random) variable, an unknown quantity.

Definition 1.1.4 (Posterior Distribution)**.** Given the prior $p(\theta)$ and the likelihood $p(y \mid x, \theta)$, the posterior distribution of θ is:

$$p(\theta \mid X, y) = \mathcal{N}(\theta \mid m_N, S_N),$$

where m_N and S_N are the posterior mean and covariance matrix, respectively. The posterior distribution also gives closed-form computations.

Remark (Why Gaussian?)**.** *The parameter vector θ is treated as a latent (random) variable, introducing uncertainty into the regression model. The prior $p(\theta)$ induces a distribution over plausible functions $f(x)$.*

If the prior on θ is Gaussian and the likelihood is Gaussian, the resulting posterior is also Gaussian. This conjugacy allows for ***closed-form computations*** *of:*

- *Predictions: $p(y_* \mid x_*, X, y)$,*
- *Marginal likelihood: $p(y \mid X)$.*

  For both GP and BLR.

Example 1.1.5 (Linear Regression Setting)**.**

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

$$p(a, b) = \mathcal{N}(0, I) \quad \text{Prior for parameters } a \text{ and } b$$

If we generate samples from the prior distribution, we get functions f_i, a plausible function reflecting the output functions (lines):

$$f_i(x) = a_i + b_i x, \quad [a_i, b_i] \sim p(a, b)$$

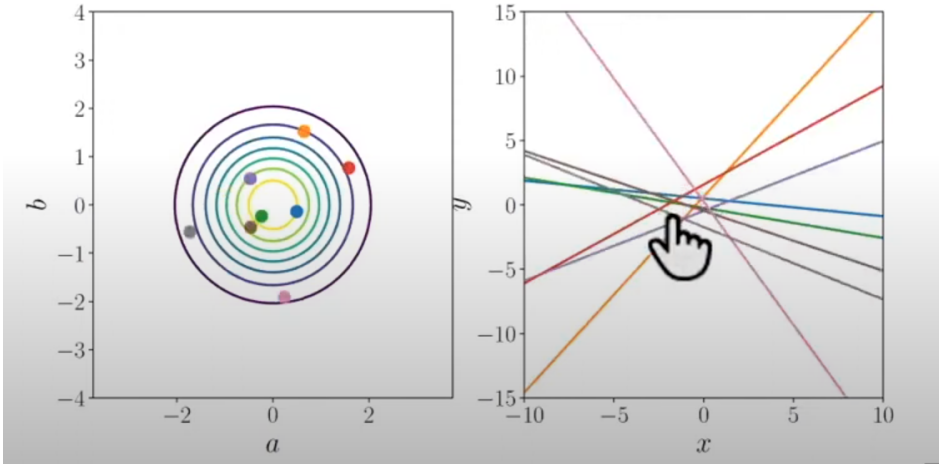


Figure 1.3

Next, we collect training data X and y. We can then compute the posterior distribution of parameters a and b:

$$p(a, b \mid X, y) = \mathcal{N}(m_N, S_N) \quad \text{Posterior on } a \text{ and } b$$

The more data we have, the more concentrated the posterior will be, same with the samples drawn from posterior. So the plausible function f becomes more similar too given a and b varies less.

Remark (Why GP?). • *Instead of sampling parameters, which induce a distribution over functions, sample functions directly*

- *Place a prior on plausible functions directly, rather ran through parameters.*
- *Instead of sampling a and b, the GP samples function values $f(x_1), f(x_2), .., f(x_n)$ at specific input points $x_1, x_2, ..., x_n$.*
- *Make assumptions on the distribution of functions*
- *Not restricted to linear forms for functions.*

- ***Intuition:*** *function = infinitely long vector of function values*
  - ***Gaussian process***

1.1.3 Gaussian Process: Definition

Definition 1.1.6 (Gaussian Process)**.** A Gaussian Process (GP) is a collection of random variables $\{f(x) \mid x \in \mathcal{X}\}$, where any finite subset of these variables $\{f(x_1), f(x_2), \ldots, f(x_n)\}$ has a joint Gaussian distribution.

Formally, a GP is fully specified by:

- A **mean function** $m(x)$, which defines the expected value of the function at any input x:

$$m(x) = \mathbb{E}[f(x)],$$

- A **covariance function** (kernel) $k(x, x')$, which defines the covariance between function values at inputs x and x':

$$k(x, x') = \text{Cov}(f(x), f(x')) = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))].$$

The notation for a Gaussian Process is:

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

For any finite set of inputs $\{x_1, x_2, \ldots, x_n\}$, the corresponding function values $\{f(x_1), f(x_2), \ldots, f(x_n)\}$ follow a multivariate Gaussian distribution:

$$\mathbf{f} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N}(\mathbf{m}, \mathbf{K}),$$

where:

- $\mathbf{m} = \begin{bmatrix} m(x_1) \\ m(x_2) \\ \vdots \\ m(x_n) \end{bmatrix}$ is the mean vector,
- $\mathbf{K} \in \mathbb{R}^{n \times n}$ is the covariance matrix, with entries $K_{ij} = k(x_i, x_j)$.

Intuition:

- A GP generalizes a multivariate Gaussian distribution to infinitely many variables, allowing us to model functions directly.
- Informally, a GP can be seen as a probability distribution over functions f, where $f = [f(x_1), f(x_2), \dots]$ represents an infinitely long vector of function values.

Remark (Comparison between BLR and GP)**.** *In Bayesian Linear Regression (BLR) and Gaussian Processes (GP), the goal is to compute the predictive distribution $p(y_* \mid x_*)$ for a new input x_*. This can be formulated mathematically as follows:*

Bayesian Linear Regression (BLR):

$$p(y_* \mid x_*) = \int p(y_* \mid x_*, \theta) p(\theta) \, d\theta$$

In BLR:

- *θ represents the parameters of the model (e.g., weights in a linear regression setting).*
- *The integration is performed over all plausible settings of θ, which represent different parameter values.*
- *The prior $p(\theta)$ encodes the uncertainty about the parameters before observing the data.*

Gaussian Processes (GP):

$$p(y_* \mid x_*) = \int p(y_* \mid x_*, f) p(f) \, df$$

In GP:

- *f represents the latent function values at all possible inputs, making f an infinite-dimensional object.*
- *The integration is performed over all plausible settings of f, which represent different function realizations.*

- *The prior $p(f)$ is a Gaussian Process prior, which encodes assumptions about the smoothness, periodicity, or other properties of the function.*

Key Difference:

- *In BLR, the predictive distribution depends on integrating over the finite-dimensional parameter space θ.*
- *In GP, the predictive distribution integrates over an infinite-dimensional function space f, enabling more flexible modeling of non-linear relationships.*
- *Conceptually, the GP eliminates the need for explicit parameterization of the function $f(x)$, allowing direct modeling of the function values as random variables.*

This distinction allows GPs to generalize beyond the limitations of linear parameterizations in BLR, making them more suitable for capturing complex, non-linear patterns in the data.

Example 1.1.7 (Predicting Temperature Using a Gaussian Process)**. Problem:** We want to predict the temperature y_* at unseen locations x_* based on observations $y = \{y_1, y_2, y_3\}$ at locations $X = \{x_1, x_2, x_3\}$.

Assumptions:

- The temperature $f(x)$ is modeled as a Gaussian Process:

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

  where $m(x)$ is the mean function and $k(x, x')$ is the covariance function.

- We use the squared exponential kernel:

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

  where σ_f^2 is the amplitude and ℓ is the length-scale.

Gaussian Process Property: For any locations $X \cup x_*$, the joint distribution of function values is:

$$\begin{bmatrix} f(X) \\ f(x_*) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(x_*) \end{bmatrix}, \begin{bmatrix} K(X, X) & K(X, x_*) \\ K(x_*, X) & K(x_*, x_*) \end{bmatrix}\right),$$

where:

- $K(X, X)$: Covariance matrix of observed locations,
- $K(X, x_*)$: Covariance between observed and unseen locations,

- $K(x_*, x_*)$: Covariance matrix of unseen locations.

Posterior Distribution: The predictive distribution at unseen locations x_* is Gaussian:

$$p(f(x_*) \mid X, y, x_*) = \mathcal{N}(\mu(x_*), \sigma^2(x_*)),$$

where:

$$\mu(x_*) = m(x_*) + K(x_*, X)K(X, X)^{-1}(y - m(X)),$$
$$\sigma^2(x_*) = K(x_*, x_*) - K(x_*, X)K(X, X)^{-1}K(X, x_*).$$

This provides the mean prediction $\mu(x_*)$ and uncertainty $\sigma^2(x_*)$ at unseen locations.

1.2 GP Regression as Bayesian Inference

1.2.1 Problem Setting

For a set of observations:

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

Goal is to find a (posterior) distribution over functions $p(f(\cdot)|X, y)$ that explains the data.

1.2.2 Bayes' Theorem

$$p(f(\cdot) \mid X, y) = \frac{p(y \mid f(X))p(f(\cdot))}{p(y \mid X)}.$$

1.2.3 Prior: Gaussian Process $p(f(\cdot)) = \mathcal{GP}(m(\cdot), k(\cdot, \cdot))$

- Mean and covariance functions encode prior knowledge.
- Here, function f_i plays the role of the parameters.
- **Possible Assumptions on the underlying function in GP:**
  - **Periodicity:** If the underlying process is expected to repeat, we might assume periodic behavior, which can be modeled using a periodic kernel.
  - **Smoothness:** The function is assumed to vary smoothly across its domain. This is often encoded using kernels like the squared exponential kernel, which enforces smooth changes in function values.
  - **Stationarity:** The function's properties are invariant to shifts in input space (e.g., the covariance depends only on the distance between inputs, not their absolute positions).
  - Others: Twice differentiable functions, functions with linear trend...

1.2.4 Likelihood

$$p(y \mid f(X)) = \mathcal{N}(f(X), \sigma_n^2 I).$$

Tells us how function value $f(X)$ is related to noise observations y.

1.2.5 Marginal Likelihood (Evidence)

$$p(y \mid X) = \int p(y \mid f(\cdot), X) p(f(\cdot) \mid X)\, df$$

It's just a **number** as we integrate out the random quantity. We divide this to ensure the posterior is normalized.

1.2.6 Posterior Distribution

$$p(f(\cdot) \mid X, y) = \mathcal{GP}(m_{\text{post}}(\cdot), k_{\text{post}}(\cdot, \cdot)),$$

where:

$$\begin{aligned} m_{\text{post}}(\cdot) &= m(\cdot) + k(\cdot, X)[K + \sigma_n^2 I]^{-1}(y - m(X)), \\ k_{\text{post}}(\cdot, \cdot) &= k(\cdot, \cdot) - k(\cdot, X)[K + \sigma_n^2 I]^{-1} k(X, \cdot). \end{aligned}$$

1.2.7 GP Prior Parameter: Mean Functions

Definition:

$$m(x) = \mathbb{E}_f[f(x)], \quad f \sim \mathcal{GP}$$

The mean function $m(x)$ represents the average function of the distribution over functions in a Gaussian Process (GP).

Key Points:

- The mean function allows us to **bias the model**, which can make sense in application-specific settings.
- **It is often set to $m(\cdot) \equiv 0$ everywhere in the absence of data or prior knowledge (e.g., for symmetry reasons). "Agnostic" mean function.**
- Using a non-zero mean function can simplify the learning problem by providing initial guidance to the model.

Parameterized Form:

- The mean function can be a **parameterized function**, such as linear, exponential, or neural networks.
- Example (Linear prior mean function):

$$m_\theta(x) = \theta^\top \phi(x),$$

where $\phi(x)$ is a feature vector, and θ represents the linear parameters.

Figures Illustration:

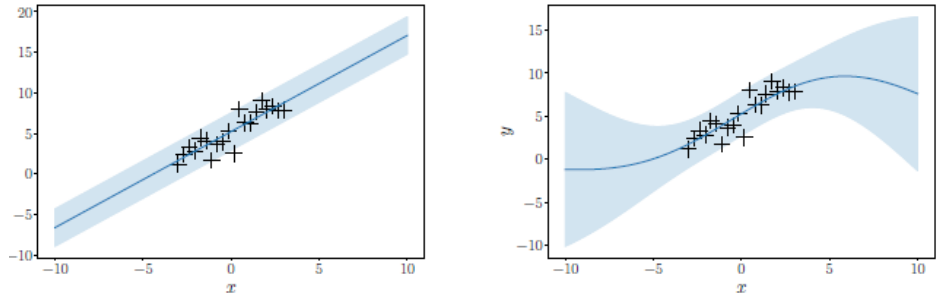


Figure 1.4

- **Left Figure (Linear mean function):** The left panel shows a case where a **linear** mean function $m_\theta(x)$ is used, providing a clear prior belief about the trend of the data. The shaded blue region indicates the uncertainty, which decreases near observed data points.
- **Right Figure:** The right panel shows a case where a **non-linear** mean function $m_\theta(x)$ is used. The prior reflects specific knowledge about the expected shape of the data.

1.2.8 GP Prior Parameter: Covariance Function

Definition: The covariance function (kernel) is **symmetric and positive semi-definite**. It computes covariances or correlations between (unknown) function $f(\cdot)$ values by looking at the corresponding inputs:

$$\text{Cov}[f(x_i), f(x_j)] = k(x_i, x_j).$$

Notes:

- Know: x_i and x_j. Unknown: $f(\cdot)$.
- Encodes **high-level structural assumptions** (e.g., smoothness, periodicity) of the function $f(\cdot)$.
- Enables the **Kernel Trick** for efficient computation in covariance.

Example: Gaussian Covariance Function

$$k_{\text{Gauss}}(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{(x_i - x_j)^\top (x_i - x_j)}{\ell^2}\right),$$

where:

- Assumption on the $f(\cdot)$: Smooth (∞-differentiable).
- σ_f: **Amplitude** of the function $f(\cdot)$. Range.
- ℓ: **Length-scale**, which determines how far we need to move in input space (e.g. x_1 and x_2) before the function values become uncorrelated.
- If x_i and x_j are very far from each other (this "far" depends on ℓ), their $k(x_i, x_j)$ is close to 0.
  - e.g. if ℓ is 100 and x_i and x_j are 50 apart, still have covariance. But if ℓ is 1, then no correlation.

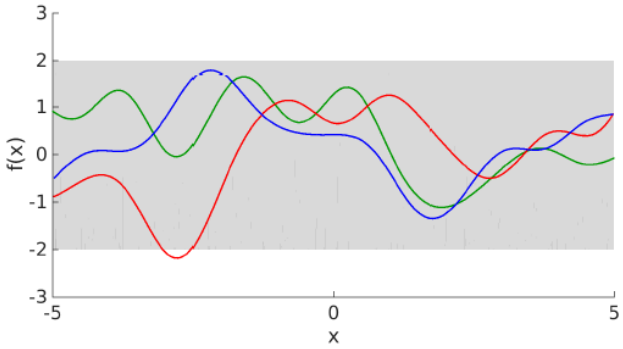


Figure 1.5: **Gaussian Covariance Function**: The figure shows samples from a GP prior with different parameterizations of the Gaussian kernel (smooth). Variations in length-scale ℓ and amplitude σ_f are demonstrated, affecting smoothness and variability.

Amplitude Parameter σ_f^2

The parameter σ_f^2 controls the **amplitude** (vertical magnitude) of the function we want to model. Larger values result in functions with greater variance in their range vertically.

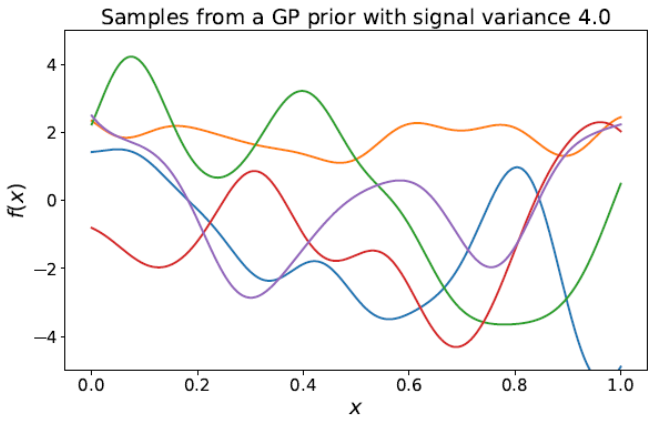


Figure 1.6: The figure shows samples from a GP prior where the signal variance $\sigma_f^2 = 4.0$. Functions exhibit increased variability in their vertical range.

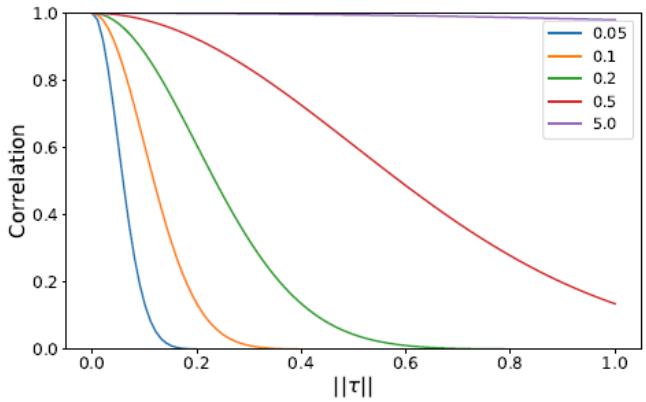


Figure 1.7: The figure shows the correlation between function values as a function of the scaled distance $\|x - x'\|/\ell$. Shorter length-scales imply functions that change more rapidly.

Length-Scale ℓ

The length-scale ℓ controls:

- How much information we can transfer to other function values (correlation between function values).
- The distance in input space from x to x' needed to make $f(x)$ and $f(x')$ uncorrelated.

- **Small** ℓ: x_i need to be **very close** in input space to have strongly correlated outputs. $f(\cdot)$ will **vary very quickly over small distances** and **highly non-smooth**.

- **Large** ℓ: x_i can be far apart and still have correlated outputs. $f(\cdot)$ will **vary slowly** and **smoother**.

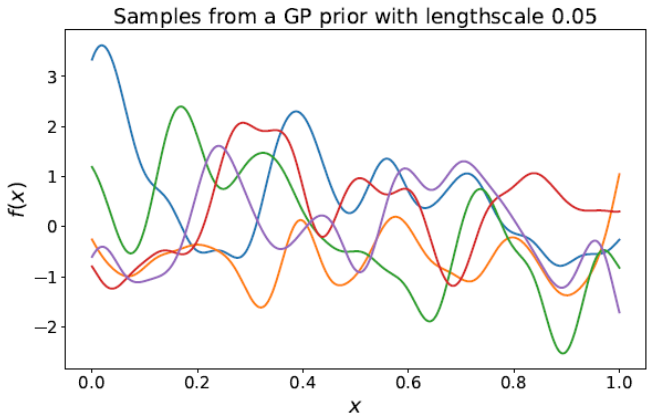


Figure 1.8: **Fast Varying Function with Small Length-scale**: The figure shows samples from a GP prior with a small length-scale ($\ell = 0.05$), resulting in highly oscillatory functions. As l increases, the function becomes smoother.

Example: Matérn Covariance Function

$$k_{\text{Mat},3/2}(x_i, x_j) = \sigma_f^2 \left(1 + \frac{\sqrt{3}\|x_i - x_j\|}{\ell}\right) \exp\left(-\frac{\sqrt{3}\|x_i - x_j\|}{\ell}\right),$$

where:

- Assumption on the function $f(\cdot)$: **1-times differentiable.** (Gaussian: infinitely differentiable)

- σ_f^2: Amplitude of the function $f(\cdot)$.

- ℓ: Length-scale of the function $f(\cdot)$.

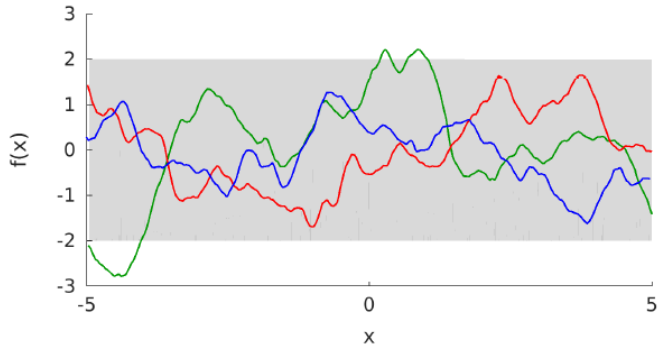


Figure 1.9: The figure illustrates samples from a GP prior using the Matérn kernel with different parameterizations. **Functions are much rougher compared to the Gaussian kernel.**

Example: Periodic Covariance Function

$$k_{\text{per}}(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{2\sin^2\left(\frac{\kappa(x_i - x_j)}{2\pi}\right)}{\ell^2}\right)$$

This can also be expressed as:

$$k_{\text{per}}(x_i, x_j) = k_{\text{Gauss}}(u(x_i), u(x_j)), \quad u(x) = \begin{bmatrix}\cos(\kappa x) \\ \sin(\kappa x)\end{bmatrix},$$

where:

- Assumption: The latent function is assumed to be **periodic**.
- σ_f^2: Amplitude of the function $f(\cdot)$.
- ℓ: Length-scale of the function $f(\cdot)$.
- κ: **Periodicity parameter**, which determines the period of the repeating patterns in the function.

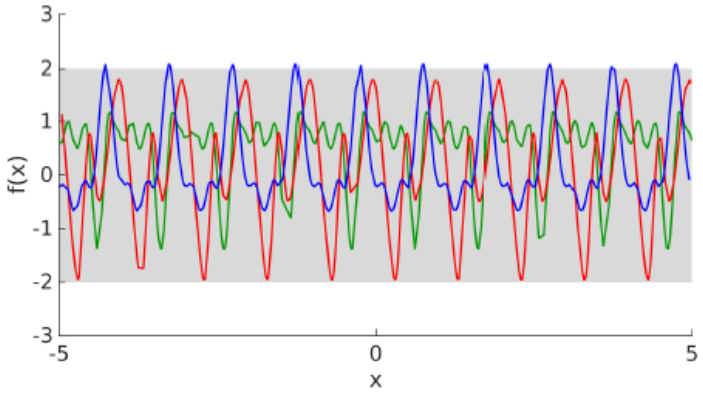


Figure 1.10: **Samples from a GP prior using the periodic covariance function:** The samples exhibit periodic behavior, with repeating patterns defined by the periodicity parameter κ. Variations in amplitude and smoothness are determined by σ_f^2 and ℓ, respectively.

Example: Creating New Covariance Functions

- Assume k_1 and k_2 are valid covariance functions, and $u(\cdot)$ is a (e.g. nonlinear) transformation of the input space. Then:
  - $k_1 + k_2$ is a valid covariance function.
  - $k_1 k_2$ is a valid covariance function.
  - $k(u(x), u(x'))$ is a valid covariance function (kernel evaluated at $u(x)$ and $u(x')$).
    - Periodic covariance function.
    - Manifold Gaussian process (Calandra et al., 2016).
    - Deep kernel learning (Wilson et al., 2016).
    - Automatic Statistician (Lloyd et al., 2014).

$$p(f(\cdot) \mid X, y) = \frac{p(y \mid f(\cdot), X)p(f(\cdot))}{p(y \mid X)}$$

1.2.9 GP Likelihood $p(y \mid f(\cdot), X)$

Gaussian likelihood in linear regression:

$$p(y \mid x, \theta) = \mathcal{N}(y \mid \theta^\top x, \sigma_n^2)$$

- **Likelihood is function (not a distribution)** of the parameters θ because it's sitting in the condition of the distribution.

- Describes how parameters and observed data are connected.
- Tells us how to transform parameters into (noisy) data.

Gaussian likelihood in Gaussian processes:

$$p(y \mid f(\cdot), x) = \mathcal{N}(y \mid f(x), \sigma_n^2)$$

Intuition: Parameters here are the function f itself !

1.2.10 GP Marginal Likelihood $p(y \mid X)$

BLG marginal likelihood with a Gaussian prior $p(\theta) = \mathcal{N}(0, I)$:

$$p(y \mid X) = \int p(y \mid X, \theta) p(\theta)\, d\theta$$

$$= \mathcal{N}(y \mid 0, \Phi\Phi^\top + \sigma^2 I)$$

$$= \mathbb{E}_\theta[p(y \mid X, \theta)]$$

Gaussian process marginal likelihood:

$$p(y \mid X) = \int p(y \mid f(\cdot), X) p(f(\cdot) \mid X)\, df$$

$$= \mathcal{N}(y \mid 0, K + \sigma^2 I)$$

$$= \mathbb{E}_f[p(y \mid f(\cdot), X)]$$

- Normalizes the posterior distribution.
- Can be computed analytically.
- Expected likelihood (under the GP prior).
- Expected predictive distribution of the training targets y (under the GP prior).

Log marginal likelihood:

$$\log p(y \mid X) = -\frac{1}{2} y^\top (K + \sigma_n^2 I)^{-1} y - \frac{1}{2} \log |K + \sigma_n^2 I| - \frac{N}{2} \log(2\pi)$$

where $K_{ij} = k(x_i, x_j)$, $i, j = 1, \ldots, N$.

1.2.11 GP Posterior $p(f(\cdot) \mid X, y)$

Posterior over functions (with training data X, y):

$$p(f(\cdot) \mid X, y) = \frac{p(y \mid f(X))p(f(\cdot))}{p(y \mid X)}$$

Using the properties of Gaussians, we obtain (with $K := k(X, X)$):

$$p(y \mid f(\cdot), X)p(f(\cdot)) = \mathcal{N}(y \mid f(X), \sigma_n^2 I)\, \mathcal{GP}(m(\cdot), k(\cdot, \cdot))$$

$$= Z \times \mathcal{GP}(m_{\text{post}}(\cdot), k_{\text{post}}(\cdot, \cdot))$$

where (conjugate prior):

$$m_{\text{post}}(\cdot) = m(\cdot) + k(\cdot, X)(K + \sigma_n^2 I)^{-1}(y - m(X)),$$

$$k_{\text{post}}(\cdot, \cdot) = k(\cdot, \cdot) - k(\cdot, X)(K + \sigma_n^2 I)^{-1}k(X, \cdot).$$

Marginal likelihood:

$$Z = p(y \mid X) = \int p(y \mid f(X))p(f(X))\, df = \mathcal{N}(y \mid m(X), K + \sigma_n^2 I).$$

1.2.12 Sampling from the GP Prior

- GP is a distribution over functions $f(\cdot)$.
  - **A sample from a GP will be an entire function too.**
- In practice, we cannot sample functions directly.
- **Instead: function = collection of function values.**
- Determine function values at a finite set of input locations:

$$X_* = \left[x_*^{(1)}, \ldots, x_*^{(K)}\right].$$

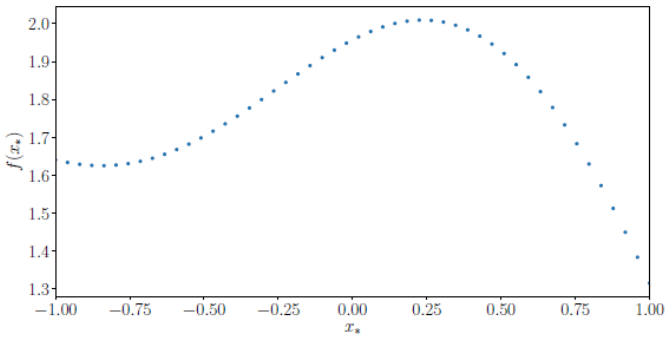


Figure 1.11: Write function f as a collection of function values.

Without any training data, the predictive distribution at test points X_* is:

$$p(f(X_*) \mid X_*) = \mathcal{N}\left(\mathbb{E}_f[f(X_*)],\, \text{Var}_f[f(X_*)]\right),$$
$$= \mathcal{N}(m_{\text{prior}}(X_*), k_{\text{prior}}(X_*, X_*)).$$

- **Exploited:** Gaussian Process defines that **all function values are jointly Gaussian distributed. So here predictive distribution follows Gaussian distribution.**
- Here k is the number of test points and $k_{\text{prior}}(X_*, X_*)$ is a k by k matrix.
- Generate **"function draws"** (sample values from the joint multivariate Gaussian distribution):

$$f_k(X_*) \sim \mathcal{N}(m_{\text{prior}}(X_*), k_{\text{prior}}(X_*, X_*)).$$

$$Define: m_* := m_{\text{prior}}(X_*), \quad K_{**} := k_{\text{prior}}(X_*, X_*).$$

Then:

$$f_k(X_*) \sim \mathcal{N}(m_*, K_{**}).$$

1.2.13 GP Predictions (Posterior)

Objective: Find posterior $p(f(X_*) \mid X, y, X_*)$ for training data X, y and test inputs X_*.

- **GP prior at training inputs:**

$$p(f(\cdot) \mid X) = \mathcal{N}(m(X), K)$$

- **Gaussian Likelihood:**

$$p(y \mid f, X) = \mathcal{N}(f(X), \sigma_n^2 I)$$

- With $f(\cdot) \sim GP$, it follows that $f(\cdot), f(\cdot)_*$ are jointly Gaussian distributed by definition:

$$p(f, f_* \mid X, X_*) = \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K & k(X, X_*) \\ k(X_*, X) & k(X_*, X_*) \end{bmatrix}\right)$$

  where:

  - $f(\cdot)$ is $[f(x_1), f(x_2), ...f(x_n)]$ and $f(\cdot)_*$ is $[f(x_1), f(x_2), ..f(x_k)]$.
  - The mean of f is $m(X)$, and the mean of f_* is $m(X_*)$.
  - $K = \text{Cov}[f(X), f(X)]$: Variance of $f(X)$.
  - $k(X, X_*) = \text{Cov}[f(X), f(X_*)]$: Covariance between $f(X)$ and $f(X_*)$.
  - $k(X_*, X_*) = \text{Cov}[f(X_*), f(X_*)]$: Variance of $f(X_*)$.

  Due to the Gaussian likelihood, we also get (f is unobserved):

$$p(y, f_* \mid X, X_*) = \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K + \sigma_n^2 I & k(X, X_*) \\ k(X_*, X) & k(X_*, X_*) \end{bmatrix}\right)$$

  Additional Notes:

- The variance changes from K to $K + \sigma_n^2 I$ where the additional term is the variance of noise term ϵ. ($y = f(x) + \epsilon$)
- The joint Gaussian property allows us to compute posterior predictions using Gaussian conditioning.

Posterior predictive distribution $p(f_* \mid X, y, X_*)$**:** We want posterior predictive distribution $p(f_* \mid X, y, X_*)$ at test inputs X_* **,** which is obtained by **Gaussian conditioning (Add details)**:

$$p(f_* \mid X, y, X_*) = \mathcal{N}\left(\mathbb{E}[f_* \mid X, y, X_*], \text{Var}[f_* \mid X, y, X_*]\right)$$

- **Posterior mean:**

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

  where

  - $m(X_*)$ is prior mean for X_*.
  - $k(X_*, X)(K + \sigma_n^2 I)^{-1}(y - m(X))$ represents the **"Kalman gain"** times the **error term** $y - m(X)$.

- **Posterior variance:**

$$\text{Var}[f_* \mid X, y, X_*] = k(X_*, X_*) - k(X_*, X)(K + \sigma_n^2 I)^{-1} k(X, X_*).$$

where

- $k(X_*, X_*)$ is prior variance for X_*.
- $k(X_*, X)(K+\sigma_n^2 I)^{-1}k(X, X_*)$ represents non-negative term. Tells how much information we can transfer from training data f to f_*.

Sanity Check on GP posterior:

- **GP posterior (from earlier):**

$$p(f(\cdot) \mid X, y) = GP(m_{\text{post}}(\cdot), k_{\text{post}}(\cdot, \cdot)),$$

where:

$$m_{\text{post}}(\cdot) = m(\cdot) + k(\cdot, X)(K + \sigma_n^2 I)^{-1}(y - m(X)),$$

$$k_{\text{post}}(\cdot, \cdot) = k(\cdot, \cdot) - k(\cdot, X)(K + \sigma_n^2 I)^{-1} k(X, \cdot).$$

- **GP posterior predictions at X_* (we just got):**

$$p(f_* \mid X, y, X_*) = \mathcal{N}(\mathbb{E}[f_* \mid X, y, X_*], \text{Var}[f_* \mid X, y, X_*]),$$

where:

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

$$\text{Var}[f_* \mid X, y, X_*] = k(X_*, X_*) - k(X_*, X)(K + \sigma_n^2 I)^{-1} k(X, X_*).$$

Only difference is replace the $\cdot$ in our earlier GP posterior with X_*!

Predictions: Make predictions by **evaluating the GP posterior mean m_{post} and covariance function $k_{\text{post}}(\cdot, \cdot)$ at a finite number of inputs X_*.**

1.2.14 Illustration: Inference with Gaussian Processes

Prior Belief about the Function $p(f)$:

$$\mathbb{E}[f(x_*) \mid x_*, \varnothing] = m(x_*) = 0,$$

$$\text{Var}[f(x_*) \mid x_*, \varnothing] = \sigma^2(x_*) = k(x_*, x_*).$$

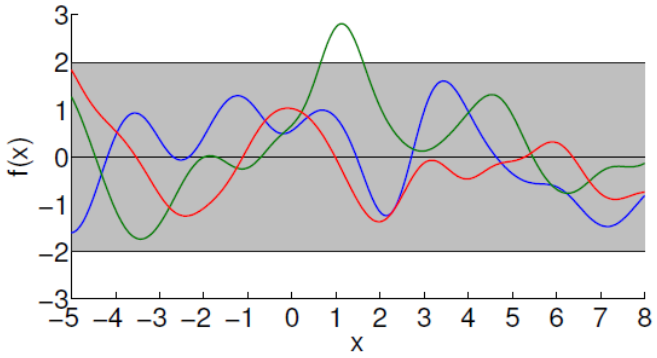


Figure 1.12

Figure Explanation:

- The figure shows samples from the prior distribution of a Gaussian Process.
- Each line represents a sample function $f(x)$, illustrating the uncertainty in $f(x)$.
- The shaded region represents the marginal variance $k(x_*, x_*)$ around the mean $m(x_*) = 0$.

Posterior about the Function f after observations:

$$\mathbb{E}[f(x_*) \mid x_*, X, y] = m_{\text{post}}(x_*) = k(x_*, X)(K + \sigma_n^2 I)^{-1} y,$$

$$\text{Var}[f(x_*) \mid x_*, X, y] = k(x_*, x_*) - k(x_*, X)(K + \sigma_n^2 I)^{-1} k(X, x_*).$$

(here $m(x_*) = 0$) so it disappeared.)

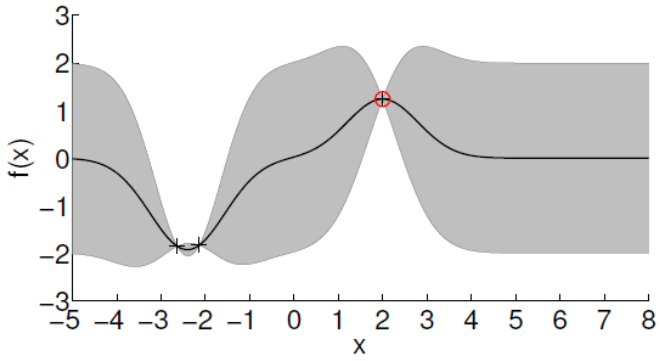


Figure 1.13

Figure Explanation:

- The figure shows the posterior distribution of the Gaussian Process after observing training data X, y.
- The solid black line represents the posterior mean $m(x_*)$, the most likely function given the observations.
- The shaded region represents the posterior variance, illustrating reduced uncertainty near observed data points.
- The red cross marks a specific test point x_*, with its corresponding predicted mean and variance.

1.3 Model Selection

1.3.1 Influence of GP Prior Parameter on Posterior

- We care most about **Generalization error** is measured by the **log-predictive density (lpd):**

$$\text{lpd} = \log p(y_* \mid x_*, X, y, \ell)$$

for different length-scales ℓ and different datasets.

  - y_*: predicted label
  - x_*: test data point
  - X: training data set
  - y: training label

  - ℓ: length-scale
  - Higher Test lpd is better fit!

- **Shorter length-scale:**
  - More flexible model. Higher error bar (blue).
  - Faster increase in uncertainty away from data. Fast varying functions.
  - Can lead to **bad generalization** properties. However, if the length-scale is too large, will still cause issue of model class not rich enough.

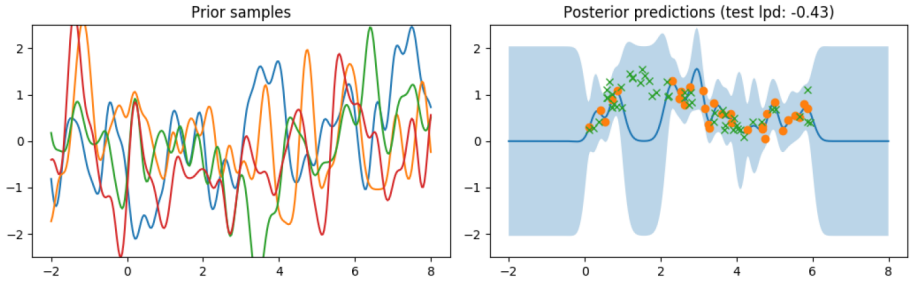


Figure 1.14: The left plot shows prior samples from a Gaussian Process with **different length-scales**. *Shorter length-scales correspond to more fluctuating behavior.* The right plot illustrates **posterior predictions**. The shaded region represents **uncertainty**, and points closer to observed data have less uncertainty. The log-predictive density (lpd) quantifies prediction quality. Orange pointers: training set. Green crosses: test set.

1.3.2 Model Selection: Summary

- **The choice of prior (e.g., length-scale, amplitude, prior type) influences predictions performance.**
- Different tasks require different priors.

The Gaussian Process (GP) possesses a set of **hyper-parameters**, which include:

- *Parameters of the covariance function, such as **length-scales and signal variance.**
- *Periodicity parameters of kernel.
- *Likelihood parameters, e.g., noise variance σ_n^2.
- Parameters of the mean function.

Remark. *Hyper-parameters control the flexibility and behavior of the GP. Properly setting these parameters is critical for accurate predictions and uncertainty quantification.*

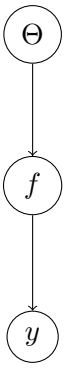
Training a GP

To find a good set of hyper-parameters, one must:

- Train the GP using data to optimize hyper-parameters.
- Perform higher-level **model selection** to determine the most suitable mean and covariance functions.

Example: Automated approaches such as the *Automatic Statistician* (Lloyd et al., 2014) can assist in selecting good mean and covariance functions.

Example: GP vs. Linear Regression in Training



GP Setting:
Hyper-parameters of the GP
Unobserved (latent) function
Observed function values (noisy)

Linear Regression:
Parameters of the model
Optimization:
Maximum Likelihood
MAP
$\arg\max_\Theta p(y \mid \Theta)$
$\arg\max_\Theta p(\Theta \mid y)$

Figure 1.15: Comparison of GP and Linear Regression Training.

Marginal Likelihood in GP:

$$p(y \mid \Theta) = \int p(y \mid f, \Theta) p(f \mid \Theta) df$$

- The likelihood is given by $p(y \mid f, \Theta)$.
- The GP prior is represented by $p(f \mid \Theta)$.
- The marginal likelihood $p(y \mid \Theta)$ is obtained by integrating over the latent function f.

Key Steps:

- Use the sum rule of marginalization to compute the marginal likelihood.
- Optimize $\arg\max_{\Theta} p(y \mid \Theta)$.

Note: This process involves combining the **likelihood and GP prior** to calculate the **marginal likelihood.**

Prior and Posterior Over Hyper-parameters

As proved above, to train a GP, the process involves finding hyper-parameters θ, including kernel/mean function parameters ψ and noise variance σ_n^2.

1.Prior Over Hyper-parameters:

$$p(\theta)$$

represents the prior distribution on hyper-parameters.

2. Posterior Over Hyper-parameters: The posterior over hyper-parameters is given by:

$$p(\theta \mid X, y) = \frac{p(y \mid X, \theta)p(\theta)}{p(y \mid X)},$$

where the marginal likelihood

$$p(y \mid X, \theta) = \int p(y \mid f, X)p(f \mid X, \theta)df.$$

Optimizing Hyper-parameters

Select hyper-parameters θ^* such that:

$$\theta^* \in \arg\max_{\theta} \log p(\theta) + \log p(y \mid X, \theta).$$

Remark. *To maximize the posterior, if we assume uniform prior, is equivalent to maximizing the marginal likelihood.*

Maximizing the marginal likelihood is equivalent to integrating out the latent functions f and finding the most probable model under the data.

Training via Marginal Likelihood Maximization

GP Training:

- Maximize the **evidence/marginal likelihood** (probability of the data given the hyper-parameters), where the latent function f has been integrated out.

- This method is also called **Maximum Likelihood Type-II**.

Marginal Likelihood (with a prior mean function $m(\cdot) \equiv 0$):

$$p(y \mid X, \theta) = \int p(y \mid f(X))p(f(X) \mid \theta)df$$

$$= \int \mathcal{N}(y \mid f(X), \sigma_n^2 I)\mathcal{N}(f(X) \mid 0, K)df$$

$$= \mathcal{N}(y \mid 0, K + \sigma_n^2 I)$$

Here the final answer is a value because y is observed. **Learning the GP Hyper-parameters:**

$$\Theta^* \in \arg\max_{\Theta} \log p(y \mid X, \theta)$$

i.e. We tune the hyperparameters such that the marginal likelihood is maximized.

How to maximize: Gradient-based Optimization

- **Log-marginal likelihood:**

$$\log p(y \mid X, \theta) = -\frac{1}{2} y^\top K_\theta^{-1} y - \frac{1}{2} \log |K_\theta| + \text{const}$$

where

$$K_\Theta := K + \sigma_n^2 I$$

- **Gradient-based optimization to get hyper-parameters Θ^*:**

$$\frac{\partial \log p(y \mid X, \theta)}{\partial \theta_i} = \frac{1}{2} y^\top K_\theta^{-1} \frac{\partial K_\theta}{\partial \theta_i} K_\theta^{-1} y - \frac{1}{2} \text{tr}(K_\theta^{-1} \frac{\partial K_\theta}{\partial \theta_i})$$

$$= \frac{1}{2} \text{tr}((\alpha\alpha^\top - K_\theta^{-1}) \frac{\partial K_\theta}{\partial \theta_i}),$$

where (using gradient descent to find local optima)

$$\alpha := K_\theta^{-1} y$$

Here, a closed form solution doesn't exist.

1.3.3 Inspection of Marginal Likelihood

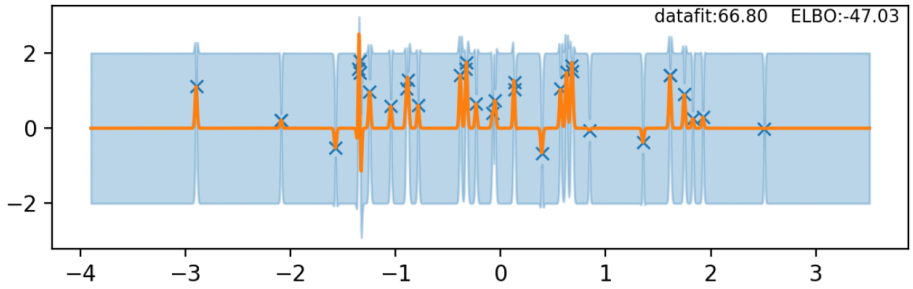


Figure 1.16: **Marginal Likelihood Illustration**: The blue region represents posterior uncertainty in predictions. Orange lines show the data-fit posterior mean. ELBO refers to the log-marginal likelihood.

Observation: The marginal likelihood balances **data fit** and **model complexity**. **When running gradient descent to maximize marginal likelihood, we can see that as ELBO increases, the Data-fit term decreases. So there's a trade-off.** Over-fitting happens when data-fit is very good while ELBO is very low.

Log-Marginal Likelihood

The log-marginal likelihood can be expressed as:

$$\log p(y|X,\theta) = -\frac{1}{2}y^\top K_\theta^{-1} y - \frac{1}{2}\log|K_\theta| + \text{const},$$

where $K_\theta = K + \sigma_n^2 I$.

- **Quadratic term (Data-fit term):** Measures whether the observation y is within the variation allowed by the prior. Always non-positive. Ideally have it close to 0.
- **Determinant (Regularization term):** Represents the product of the variances of the prior (**volume of the prior**). The volume is approximately the **richness of the model class**. Ideally goes to negative.
- Automatic trade-off between **data fit (Quadratic)** and **model complexity (Determinant)**.

Marginal Likelihood Surface

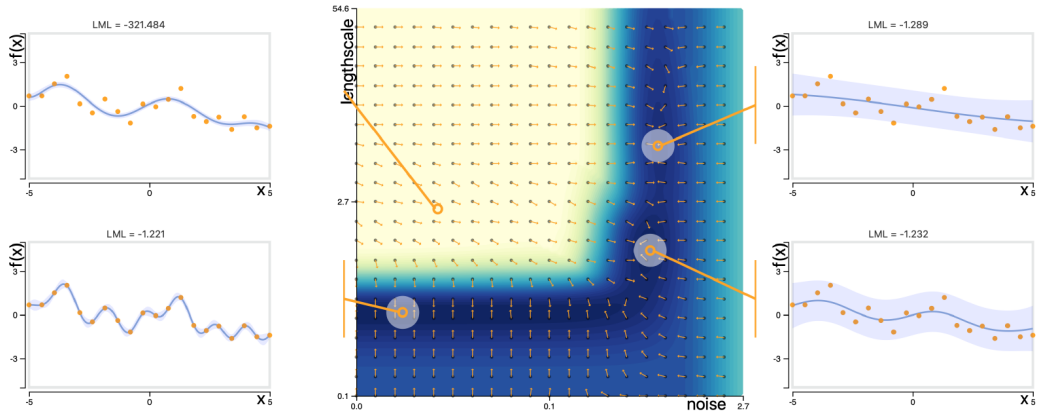


Figure 1.17

The figure above illustrates the marginal likelihood surface with several plausible hyper-parameters (local optima). The key insights are:

- Each local optimum corresponds to a specific combinations of hyper-parameters (e.g. length-scale, noise/uncertainty bar).
- The marginal likelihood surface is **non-convex**, leading to multiple optima.
- LML: Log Marginal Likelihood Values (ELBO) higher, the better.

Marginal Likelihood and Parameter Learning

- In the very-small-data regime, GPs can end up in three different situations when optimizing hyper-parameters:
  - **Short length-scales, low noise** (highly nonlinear mean function with little noise).
  - **Long length-scales, high noise** (everything is considered noise).
  - **Hybrid regime.**
- **Re-start hyper-parameter optimization:** Use random initialization to mitigate local optima issues.
- With increasing dataset size, GPs typically end up in the ”hybrid” mode (i.e. more likely to have unique optimal).

- Ideally, we can also integrate the hyper-parameters out, but there is **no closed-form solution**. This requires **Markov chain Monte Carlo (MCMC)** techniques to approximate, but computationally heavy.

Why Does the Marginal Likelihood Work?

- **Overall goal:** Achieve good generalization performance on unseen test data.
- Minimizing training error alone leads to **overfitting**.
- Adding uncertainty makes predictions more cautious but may not improve performance if the model is wrong.
- The marginal likelihood balances **data fitting and model complexity** (Occam's razor). But Why?

The marginal likelihood works well for generalization because it inherently balances **data fit** and **model complexity** using Bayesian principles. Below, we explain the key reasons why it is effective:

1. **Automatic Complexity Penalization:**
   - The marginal likelihood inherently penalizes both overly complex models (overfitting) and overly simple models (underfitting).
   - This is achieved through the decomposition:

$$\log p(y \mid \mathcal{M}) = \underbrace{\log p(y \mid \theta, \mathcal{M})}_{\text{data fit}} + \underbrace{\log p(\theta \mid \mathcal{M})}_{\text{complexity penalty}} - \underbrace{\log p(\theta \mid y, \mathcal{M})}_{\text{posterior fit to prior}}.$$

2. **Bayesian Occam's Razor:**
   - The marginal likelihood embodies Occam's razor by favoring models that strike a balance between simplicity and fit.
   - It avoids penalizing simple models excessively while disfavoring complex models that overfit.
   - For example, the prior volume scales as:

$$\text{Volume} \propto |K_\theta|^{1/2},$$

   where K_θ is the covariance matrix of the prior distribution, ensuring only rich enough models are selected.

3. **Integration Over Parameters:**

   - Instead of optimizing parameters directly, marginal likelihood integrates over all possible parameter values:
   $$p(y \mid \mathcal{M}) = \int p(y \mid \theta, \mathcal{M}) p(\theta \mid \mathcal{M}) \, d\theta.$$
   - This avoids overfitting by not committing to a specific parameter set that maximizes the likelihood.

4. **Uncertainty Awareness:**

   - The marginal likelihood accounts for uncertainty in both the data and model parameters.
   - This is particularly useful in small-data regimes, where overconfident predictions can lead to poor generalization.
   - The predictive distribution for a Gaussian process, for instance, incorporates the posterior variance:
   $$p(f_* \mid y, X) = \mathcal{N}\left(m_*, k_* - k_{*,X}(K + \sigma_n^2 I)^{-1} k_{X,*}\right),$$
   where m_* and k_* denote the posterior mean and variance.

Incremental Prediction with Marginal Likelihood

Definition 1.3.1. The "probability of the training data" given the parameters is expressed as (factorized):

$$p(y|\theta) = p(y_1, \ldots, y_N|\theta) = p(y_1|\theta)p(y_2|y_1, \theta) \cdots p(y_N|y_1, \ldots, y_{N-1}, \theta).$$

In a compact form:

$$p(y|\theta) = p(y_1|\theta) \prod_{n=2}^{N} p(y_n|y_1, \ldots, y_{n-1}, \theta).$$

Remark (Marginal Likelihood as a Sequence Model)**.** *The marginal likelihood can be interpreted as a sequence model where data arrives sequentially, and the model predicts each observation based on all previous observations:*

$$p(y \mid \theta) = p(y_1 \mid \theta) \prod_{n=2}^{N} p(y_n \mid y_1, \ldots, y_{n-1}, \theta).$$

Key insights:

- *The marginal likelihood predicts the n-th training observation given all the "previous" observations, treating the process as sequential.*

- *This allows us to predict training data y_n that has not yet been accounted for, using only the condition $y_1, \ldots, y_{n-1}$:*

  Treat the next data point as test data.

- *Intuition: If the model predicts well for all N training points sequentially, it is likely to perform well on unseen test data. Hence, the marginal likelihood serves as a Proxy for generalization error on unseen test data.*

1.3.4 Marginal Likelihood Evolution

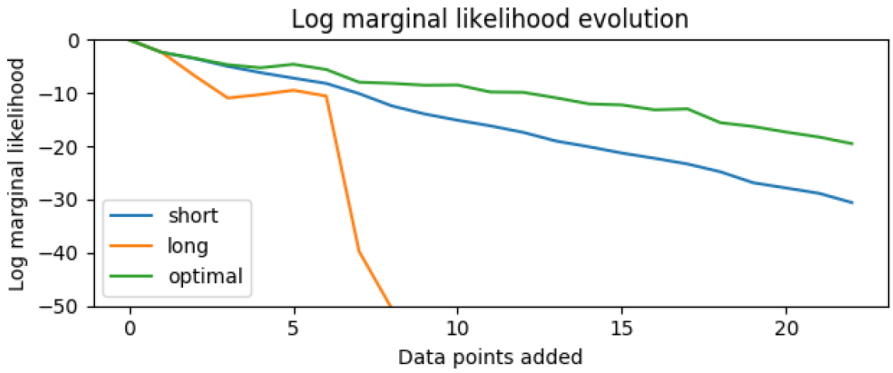


Figure 1.18: Log marginal likelihood evolution as data points are added for different lengthscales. The plot compares the behavior of short, long, and optimal lengthscales.

The figure above illustrates the evolution of the log marginal likelihood as more data points are added. The behavior of different lengthscales is analyzed as follows:

- **Short lengthscale:**
  - Consistently **overestimates variance**.
  - Results in no high density, even with observations inside the error bars.
- **Long lengthscale:**
  - Consistently **underestimates variance**.
  - Observations are outside the error bars, leading to low density.
- **Optimal lengthscale:**
  - **Trades off** both behaviors reasonably well, achieving a balance between variance overestimation and underestimation.

The log marginal likelihood reflects how well the Gaussian Process model fits the data while balancing model complexity, with the optimal lengthscale yielding the best generalization performance.

1.3.5 Model Selection: Optimal Mean Function and Kernel

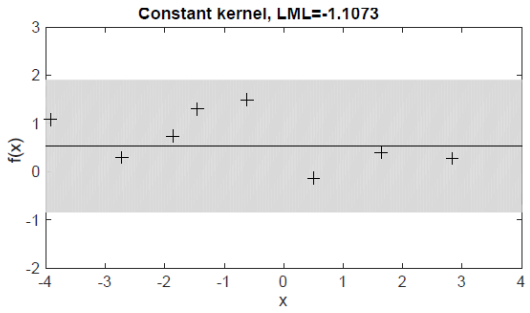
Assume we have a finite set of models M_i, each one specifying a mean function mi and a kernel ki. How do we find the best one?

- Cross validation
- Bayesian Information Criterion, Akaike Information Criterion (AIC, BIC)
- **Compare marginal likelihood values (assuming a uniform prior on the set of models)**

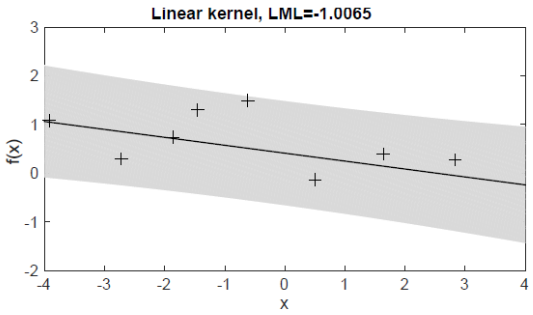
1.4 GP Example: Model Comparsion

In this section, we demonstrate the application of GP models with four different kernels. Each kernel is optimized using Maximum A Posteriori (MAP) hyper-parameters, and the log-marginal likelihood (LML) values are computed for each optimized model. The mean function is fixed to $m \equiv 0$.

- **Constant Kernel:** The LML value is -1.1073. This kernel assumes the function is constant across the input space.
- **Linear Kernel:** The LML value is -1.0065. This kernel captures linear trends in the data.
- **Matérn Kernel:** The LML value is -0.8625. This kernel captures smooth variations in the data with flexible degrees of differentiability.
- **Gaussian Kernel:** The LML value is -0.69308. This kernel models smooth functions with a characteristic length scale.

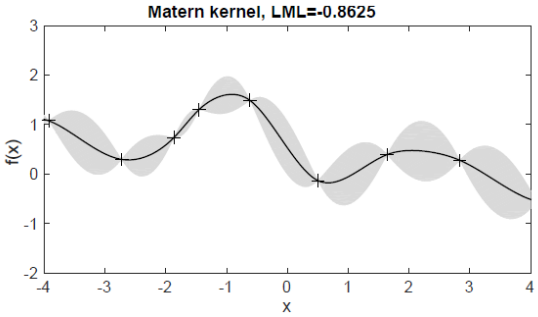


(a) Constant Kernel, LML = -1.1073

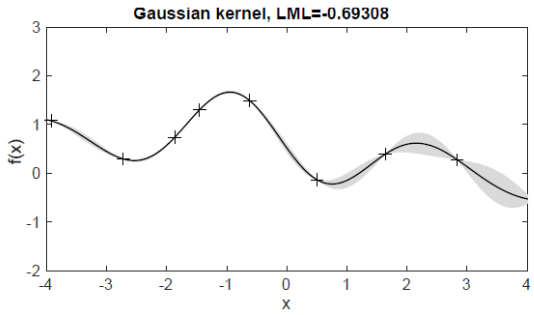


(b) Linear Kernel, LML = -1.0065

Figure 1.19: Comparison of constant and linear kernels. The shaded area represents uncertainty.



(a) Matérn Kernel, LML = -0.8625



(b) Gaussian Kernel, LML = -0.69308

Figure 1.20: Comparison of Matérn and Gaussian kernels. The shaded area represents uncertainty.

1.4.1 GP Training Illustration

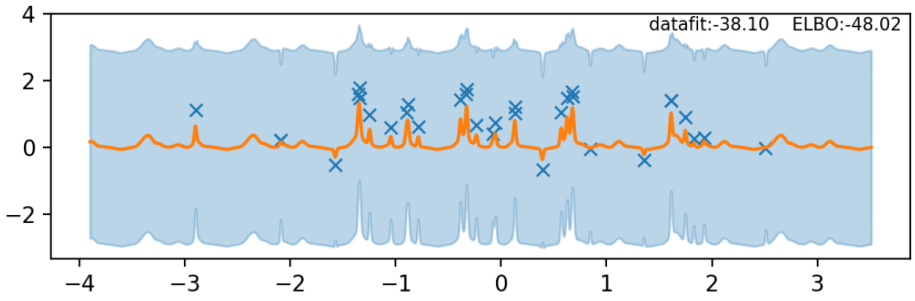


Figure 1.21: combined kernel GP model.

In this example, the GP model combines smooth and periodic priors:

$$f(x) = \theta_s f_{\text{smooth}}(x) + \theta_p f_{\text{periodic}}(x),$$

where θ_s and θ_p are the hyper-parameters controlling smoothness and periodicity, respectively. The marginal likelihood automatically balances smoothness and periodicity to generalize the data well.

1.5 Limitations and Guidelines

1.5.1 Limitations of GP

- **Computational and memory complexity**
  - Training set size: N
  - Training scales in $\mathcal{O}(N^3)$
  - Prediction (variances) scales in $\mathcal{O}(N^2)$
    - Low-rank update of inverse kernel matrix in $\mathcal{O}(N^2)$. Computing inverse matrix is very expensive.
  - Memory requirement: $\mathcal{O}(ND + N^2)$
- **Practical maximum limit:** $N \approx 10,000$

Some solution approaches:

- Sparse GPs with *inducing variables*
- Combination of *local GP expert models*
- *Variational Fourier features* .

1.5.2 Tips and Tricks for Practitioners

- To set initial hyper-parameters, use **domain knowledge**.
- Standardize input data and set **initial length-scales** $\ell \approx 0.5$.
- Standardize targets y and set **initial signal variance** to $\sigma_f \approx 1$.
- Often useful: Set **initial noise level relatively high** (e.g., $\sigma_n \approx 0.5 \times \sigma_f$ amplitude), even if you think your data have low noise. The optimization surface for your other parameters will be easier to move in.
- When optimizing hyper-parameters, try **random restarts** or other tricks to avoid local optima.

- Mitigate the problem of **numerical instability** (Cholesky decomposition of $K + \sigma_n^2 I$) by **penalizing high signal-to-noise ratios** σ_f / σ_n.

For more detailed guidelines, refer to the resource: https://drafts.distill.pub/gp and https://infallible-thompson-49de36.netlify.app/.

Chapter 2

Bayesian Optimization

2.1 Machine Learning Meta-Challenges

Definition 2.1.1 (ML Meta-Challenges)**.** Machine learning models are becoming increasingly complex, involving:

- Large numbers of parameters (e.g., deep neural networks).
- Non-convex and Stochastic Optimization processes with meta-parameters (e.g., learning rates, momentum parameters).

These challenges make it difficult to apply modern techniques or reproduce results reliably.

Remark. *The goal of automated machine learning (AutoML) is to automate the selection of* ***critical meta-parameters****, reducing the manual effort required for hyper-parameter tuning.*

2.1.1 Examples of Optimization Challenges

Deep Neural Networks

Example 2.1.2 (Deep Neural Networks)**.** Deep neural networks are widely used in applications such as:

- Visual object identification,
- Speech recognition, and
- Computational biology.

However, they require careful tuning of hyper-parameters, including *Number of layers, Weight regularization, Choice of non-linearity, Batch size, and Learning rate schedules*.

Online Latent Dirichlet Allocation (LDA)

Example 2.1.3 (Topic Modeling with LDA)**.** Hoffman et al. (2010) introduced approximate inference methods for large-scale text analysis using Latent Dirichlet Allocation (LDA). While empirically successful, the following hyper-parameters must be tuned:

- Dirichlet parameters,
- Number of topics,
- Learning rate schedules, and
- Vocabulary size.

Classification of DNA Sequences

Example 2.1.4 (DNA Sequence Classification)**.** Miller et al. (2012) proposed using Latent Structural Support Vector Machines to predict the binding of DNA sequences to proteins. Critical hyper-parameters include:

- Margin/slack parameters,
- Entropy parameters, and
- Convergence criteria.

2.2 Search for Good Hyper-parameters

Definition 2.2.1 (Objective Function for Hyper-Parameter Search)**.** Usually, we care about the generalization performance. We can define $g(x)$ to be an objective function (e.g. RMSE) representing the performance of a model under hyper-parameter configuration x. The goal is to find:

$$x^* = \arg\min_x g(x),$$

where $g(x)$ is typically evaluated using cross-validation to measure parameter quality.

Remark. *Common search methods include:*

1. *Manual tuning,*
2. *Grid search (give a possible range for each parameters and attempt),*
3. *Random search (random sampling within an interval for each parameter),*
4. ***More advanced methods, such as Bayesian optimization.***

However, evaluating $g(x)$ is often computationally expensive and noisy, especially for complex models like deep neural networks. Many training cycles as well.

2.3 Alternative Approach: Bayesian Optimization

Definition 2.3.1 (Bayesian Optimization)**.** Bayesian optimization is a probabilistic framework for optimizing expensive-to-evaluate black-box functions. Specifically, it aims to find the **global minimizer** of a function $g(x)$:

$$x^* = \arg\min_{x \in \mathcal{X}} g(x),$$

where $\mathcal{X}$ represents the search space, and $g(x)$ **is assumed to be continuous but does not provide gradients or any known functional form**. Observations of $g(x)$ may be noisy. (Discrete?)

This process leverages a **surrogate/proxy model** $\widetilde{g}(x)$, commonly based on a **Gaussian Process (GP)**, which provides:

- A predictive mean function $\mu(x)$, estimating the expected value of $g(x)$.
- A variance function $\sigma^2(x)$, quantifying uncertainty in the predictions of $g(x)$.
- This $\widetilde{g}(x)$ is trained using the outcomes of past experiments of $g(x)$ as training data.
- This $\widetilde{g}(x)$ is much cheaper to evaluate.

2.3.1 Key Steps in Bayesian Optimization

Bayesian optimization operates iteratively by:

1. To avoid evaluating $g(x)$ an excessive number of times, we use a proxy model $\widetilde{g}(x)$.
2. Find a global minimizer $\widetilde{g}(x_*)$ of proxy function $\widetilde{g}(x)$, **where x_* is the optimal hyperparameters.**
3. Evaluate the true objective $g(x)$, at x_*.
4. Obtain new experiments of $g(x)$, which we use as training data to find the next x_*.

   Overall, evaluate g only once per cycle!

   However, this only works well if $\widetilde{g}(x)$ and $g(x)$ are close. We have to repeat this cycle: evaluate the g then update $\widetilde{g}$, iteratively, to make them close.

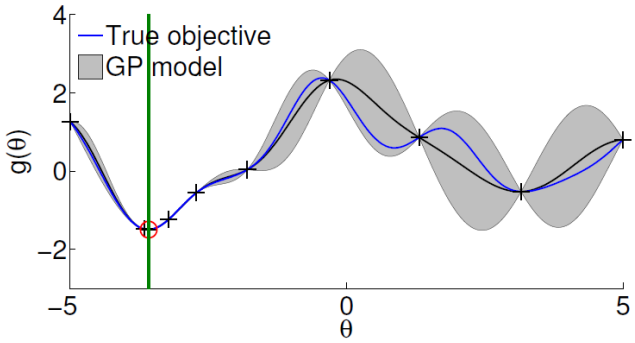


Figure 2.1: How the iteration works in visual: Emphasis on where the $g(\theta)$ is minimal.

2.4 Choosing the Next $g(\theta)$ to Evaluate: Acquisition Functions

2.4.1 Using Uncertainty in Global Optimization

To find a good global optimum, it is essential to **escape local optima**. This is achieved by extrapolating knowledge from the collected data points. **A Gaussian Process (GP) provides a probabilistic model with closed-form expressions for the predictive mean $\mu(x)$ and variance $\sigma^2(x)$.**

Bayesian optimization balances exploration and exploitation:

Exploration: Focuses on regions with **high variance** $\sigma^2(x)$, where there is high uncertainty about the objective function.

Exploitation: Prioritizes areas with a **low predictive mean** $\mu(x)$, where the function is likely to have low values (that is what we want).

This balance is achieved by designing an acquisition function $\alpha(x)$, which guides the search for the next evaluation point.

2.4.2 Pseudo-Code for Optimizing $\alpha(x)$

Theorem 2.4.1 (Acquisition Function Optimization)**.** *The iterative process for $\alpha(x)$ optimization is as follows:*

1. ***Initialization:*** *Start with an initial dataset $\mathcal{D}_0 = \{X_0, y_0\}$, where X_0 is the initial set of input data points (hyper-parameters), and $y_0 = g(X_0)$ are their corresponding evaluations of the objective function.*

2. ***Iterative Steps:*** *For iterations $t = 1, 2, \ldots$, perform the following steps:*

(a) ***Update the Gaussian Process:*** *Using the current dataset $\mathcal{D}_{t-1}$, fit a Gaussian Process (GP) to get proxy function $\widetilde{g}(x)$.*

(b) ***Optimize the Acquisition Function:*** *Select the next evaluation point by:*

$$x_t = \arg\max_x \alpha(x),$$

where $\alpha(x)$ is the acquisition function derived from the GP posterior of $\widetilde{g}(x)$.

(c) ***Query the True Objective Function:*** *Evaluate the objective function $g(x)$ at the chosen point x_t, obtaining $y_t = g(x_t)$.*

(d) ***Augment the Dataset:*** *Update the dataset by adding the new experiment of $g(x)$:*

$$\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(x_t, y_t)\}.$$

3. ***Return the Best Point:*** *After completing the iterations (manually terminate), return the input x^* corresponding to the best observed output:*

$$x^* = \arg\min_x y(x).$$

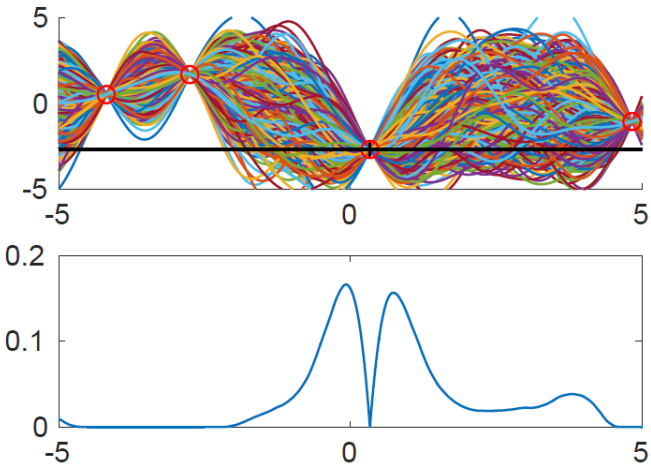


Figure 2.2: **Upper panel: Samples from a probabilistic proxy $\widetilde{g}(x)$.** Each curve represents a possible realization of the $\widetilde{g}(x)$ sampled from the GP posterior. Black line is the minimal value of $\widetilde{g}(x)$ so far. **Lower panel: Corresponding Expected Improvement (EI) acquisition function over the best solution so far.** EI measures how much better we expect $g(x_{best})$ to perform compared to $g(x)$ given the GP posterior model. The peaks in the EI curve indicate the points where evaluating $g(x)$ is expected to yield the most improvement. These peaks guide where to evaluate next (x_t will be the x-axis of the peak point in lower panel).

2.4.3 Closed-Form Acquisition Functions

Let x_{best} denote the best observed point so far (x-axis corresponding to red point in black horizontal line). For all $x \in \mathbb{R}^D$, **the GP posterior provides a predictive mean** $\mu(x)$ **and variance** $\sigma^2(x)$. Define the normalized improvement function as:

$$\gamma(x) = \frac{g(x_{\text{best}}) - \mu(x)}{\sigma(x)}.$$

Acquisition functions are deterministic, unable to be sampled like probabilistic functions.

Probability of Improvement (PI): Proposed by Kushner (1964), the probability of improvement is given by:

$$\alpha_{\text{PI}}(x) = \Phi(\gamma(x)),$$

where $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution. This method prioritizes regions likely to improve upon $g(x_{\text{best}})$.

Expected Improvement (EI): Introduced by Mockus (1978), the expected improvement function combines both the magnitude and probability of improvement:

$$\alpha_{\text{EI}}(x) = \sigma(x)\left[\gamma(x)\Phi(\gamma(x)) + \mathcal{N}(\gamma(x) \mid 0, 1)\right],$$

where $\mathcal{N}(\cdot \mid 0, 1)$ is the probability density function of the standard normal distribution. Normally work the best among all acquisition functions.

GP Lower Confidence Bound (LCB): Proposed by Srinivas et al. (2010), the LCB criterion balances exploration and exploitation by adjusting the trade-off using a parameter $\kappa > 0$:

$$\alpha_{\text{LCB}}(x) = -\left(\mu(x) - \kappa\sigma(x)\right).$$

2.4.4 Probability of Improvement: Details

The probability of improvement focuses on determining whether x_* leads to a better value of $g(x)$ than $g(x_{\text{best}})$. In a sampling-based approach, the probability of improvement can be approximated as:

$$\alpha_{\text{PI}}(x) = p(g(x) < g(x_{\text{best}})) \approx \frac{1}{N}\sum_{i=1}^{N} \delta(g_i(x) < g(x_{\text{best}})),$$

where N is the number of sampled functions g_i, and $\delta(\cdot)$ is the indicator function. However, this can lead to continued exploitation but slow exploration in an ϵ -region around x_{best} **(because**

of the smoothness assumption we made in Gaussian Functions), and the improvement is not sufficient.

To encourage **more aggressive exploration**, a slack variable $\xi > 0$ can be introduced as minimal improvement, leading to a modified probability of improvement (i.e. at least ξ better than the best solution):

$$\alpha_{\text{PI}}(x) = p(g(x) < g(x_{\text{best}}) - \xi).$$

Range for slack variable: *Looking at the amplitude/signal variance hyper-parameters in GP in proxy function. Or mean/std of the experiments you've run so far.*

If the posterior distribution of $g(x)$ is Gaussian, the closed-form expression for the probability of improvement becomes:

$$\alpha_{\text{PI}}(x) = \Phi(\gamma(x,\xi)), \quad \gamma(x,\xi) = \frac{g(x_{\text{best}}) - \xi - \mu(x)}{\sigma(x)}.$$

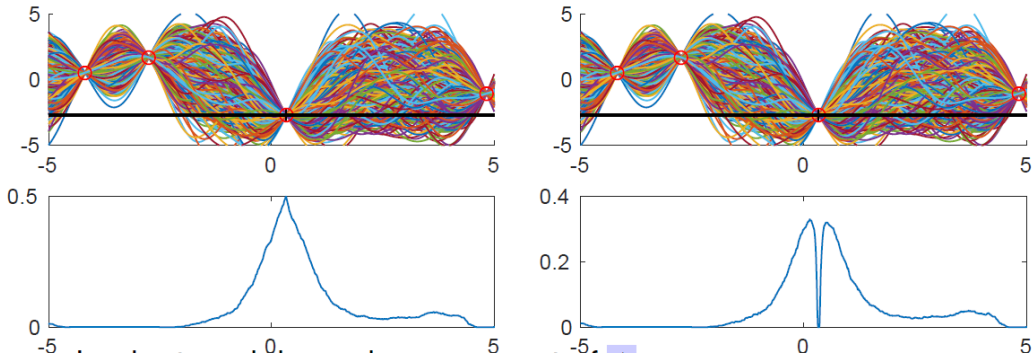


Figure 2.3: **Probability of Improvement:** Left without slack variable, Right with slack variable. In the left plot, the probability reaches 1/2 for the most likely best next point - why?

2.4.5 Expected Improvement: Details

Definition 2.4.2 (Expected Improvement (EI))**.** Expected Improvement (EI) quantifies the **expected amount of improvement at a given point x relative to the current best observed value $g(x_{\text{best}})$.** It is defined as:

$$\alpha_{\text{EI}}(x) = \mathbb{E}\left[\max\{0, g(x_{\text{best}}) - g(x)\}\right].$$

Remark. *The EI is evaluated under the posterior distribution of the objective function $g(x)$, as provided by the Gaussian Process (GP). It balances exploration (uncertainty) and exploitation (minimizing the mean prediction).*

Sampling-Based Approximation In practice, we can approximate the EI in a sampling-based scenario where $g_i \sim p(f)$ are samples from the posterior of the GP:

$$\alpha_{\text{EI}}(x) \approx \frac{1}{N} \sum_{i=1}^{N} \max\{0, g(x_{\text{best}}) - g_i(x)\}.$$

If $f \sim$ GP, the EI can be expressed in closed form:

$$\alpha_{\text{EI}}(x) = \sigma(x) \left[\gamma(x)\Phi(\gamma(x)) + \phi(\gamma(x))\right],$$

where:

$$\gamma(x) = \frac{g(x_{\text{best}}) - \mu(x)}{\sigma(x)},$$

and $\Phi(\cdot)$ and $\phi(\cdot)$ denote the CDF and PDF of the standard normal distribution, respectively.

Remark. *The slack-variable approach (similar to Probability of Improvement) can also be used for more aggressive exploration.*

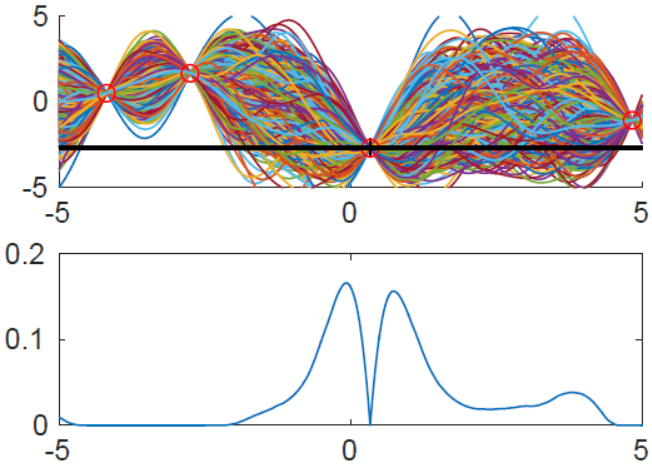


Figure 2.4: Expected Improvement.

2.4.6 GP Lower Confidence Bound

Theorem 2.4.3 (Lower Confidence Bound (LCB) Criterion)**.** *The GP-LCB acquisition function directly uses the predictive mean $\mu(x)$ and variance $\sigma^2(x)$ to guide exploration and exploitation. It is defined as:*

$$\alpha_{LCB}(x_t) = -\left(\mu(x_t) - \sqrt{\kappa}\sigma(x_t)\right),$$

where $\kappa > 0$ controls the ***exploration-exploitation trade-off****.*

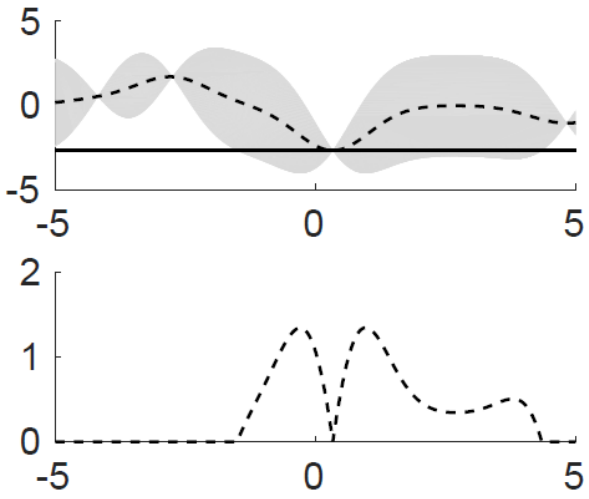


Figure 2.5: GP-Lower Confidence Bound.

Remark. *For iteration-dependent κ_t, which is the amount of training data we have,* ***regret bounds*** *can be established (Srinivas et al., 2010):*

$$\alpha_{LCB}(x_t) = -\left(\mu(x_t) - \sqrt{\kappa_t}\sigma(x_t)\right),$$

where $\kappa_t \in \mathcal{O}(\log t)$ grows with the iteration t. This guarantees continue exploration.

2.4.7 Optimizing the Acquisition Function

- Optimizing the acquisition function **requires running a global optimizer within Bayesian optimization to maximize acquisition function.**
- Evaluating the acquisition function is cheap compared to evaluating the true objective $g(x)$, allowing **repeated evaluations.**
- This enables Bayesian optimization to **efficiently balance exploration and exploitation**.

2.4.8 Limitations of Bayesian Optimization

- **Model Error:** Incorrect choice of the GP model (e.g., covariance function) for proxy function $\widetilde{g}(x)$ can lead to catastrophic errors.
- **Scalability:** Limited scalability in terms of **dimensionality** (i.e. number of hyper-parameters we can optimize each time) and/or **the number of evaluations** of the true objective function.

- This limitation arises due to the **cubic complexity of GPs** with respect to the number of data points. (10,000 as the limit for GP number of evaluations)
- Have to be below 1000 experiments: Running optimizer for acquisition function each time, we need to **query GP at many inputs** to make predictions ($O(N^2)$).
- Why can't optimize 100 hyperparameters in Bayesian optimization - curse of dimensionality. Ideally within 10 parameters.

2.4.9 Covariance Function Selection

Theorem 2.4.4 (Covariance Function Choice)**.** *The choice of covariance function (kernel) is critical for the performance of Bayesian optimization. A sufficiently flexible and adaptive kernel, such as the* ***Matérn kernel****, is recommended over the* ***squared exponential (Gaussian) kernel*** *because Matern kernel is more flexible and adaptive* ***(while the parameter for Matern is yet to decide)****, and the smoothness assumption is not as strong as Gaussian kernel so allowing more exploration.*

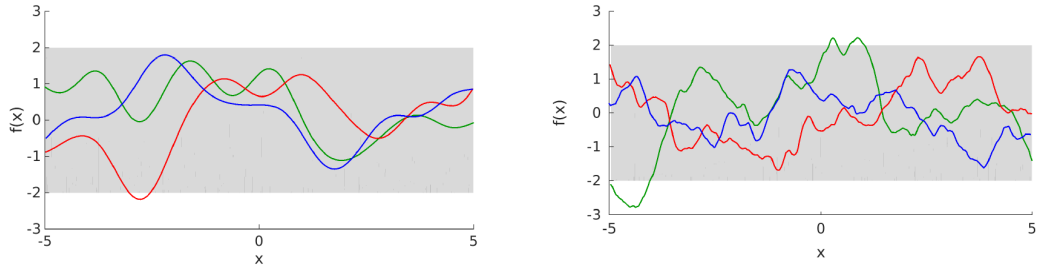


Figure 2.6: Left: Gaussian Kernel. Right: Matern Kernel.

The choice of kernel largely determines how fast the function goes to good values (converge). To find the best kernel, can apply model selections if dataset is sufficient enough. When lack of data, e.g. 2 data points for linear kernel.

2.4.10 Gaussian Process Hyper-Parameters

- Hyper-parameter optimization via marginal likelihood maximization (Empirical Bayes) can fail, especially in early iterations when there are few data points. Leads to multiple local optima.
- **A better approach is to integrate out the GP hyper-parameters θ using Markov Chain Monte Carlo (MCMC) sampling such as slice sampling, which give faster experiments but not compute.**

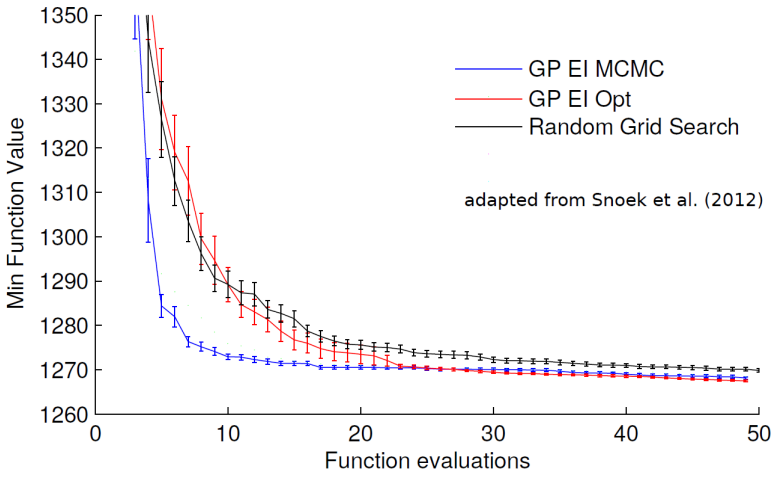


Figure: Figure from Snoek et al. (2012)

Figure 2.7: **Integrating out GP Hyper-parameters (fixed parameters):** this example optimize the two critical hyper-paraemters that control the learning rate. Using three methods including random grid search, GP EI MCMC, GP EI Opt,**where GP EI MCMC shows so much faster learning rate especially in early stage.**

Integrated Acquisition Function The acquisition function can be marginalized over the posterior of the hyper-parameters:

$$\alpha(x) = \mathbb{E}_\theta[\alpha(x,\theta)] = \int \alpha(x,\theta)p(\theta \mid X_n, y_n)d\theta,$$

where $\theta \sim p(\theta \mid X_n, y_n)$ is sampled using MCMC.

2.4.11 Further Topics in Bayesian Optimization

- **Entropy-based acquisition functions:** Directly describe the distribution over the best input location (Hennig & Schuler, 2012; Hernández-Lobato et al., 2014).
- **Non-myopic** Bayesian optimization (e.g., Osborne et al., 2009).
- **High-dimensional** optimization (e.g., Wang et al., 2016).
- **Large-scale** Bayesian optimization (Hutter et al., 2014).
- **Efficient optimization of acquisition functions** (Wilson et al., 2018).

- **Non-GP** Bayesian optimization (Hutter et al., 2014; Snoek et al., 2015).
- **Constraints** (e.g., Gelbart et al., 2014).
- **Automated machine learning** (e.g., Feurer et al., 2015).
- **Multi-tasking, parallelizing, resource allocation, ...** (e.g., Swersky et al., 2014; Snoek et al., 2012; Wilson et al., 2018).
- Multi-Objective Context. (e.g. taking the geometric mean of all objectives - cost could be expensive given the difference in unit/scale)
- Categorial/Classification case: non-conjugage prior issue, approx inference required

2.4.12 Software

- **BoTorch** https://github.com/pytorch/botorch (Balandat et al., 2019).
- **BayesOpt** https://bitbucket.org/rmcantin/bayesopt/ (Martinez-Cantin, 2014).
- **Spearmint** https://github.com/HIPS/Spearmint.
- **Pybo** https://github.com/mwhoffman/pybo (Hoffman & Shariari).
- **GPyOpt** https://github.com/SheffieldML/GPyOpt (Gonzalez et al.).
- **Matlab toolbox (bayesopt)**.

2.4.13 Summary

- Global optimization of black-box functions, which are expensive to evaluate. Meta-challenges in machine learning, Auto-ML.
- Use a probabilistic proxy model that is cheap to evaluate and use this to suggest next experiments.
- Acquisition function trades off exploration and exploitation to ensure we don't end up in local optimum.
- Sometimes more interested in local optima when you want robustness, when global optima gives tiny width (sharp specific values for the parameters)

2.4.14 Supplement: Gaussian Process for Approximating $g(x)$

GP Prior $p(g(x))$

Definition 2.4.5 (GP Prior)**.** A Gaussian Process (GP) defines a prior distribution over a latent function $g(x)$, which we aim to approximate. It assumes that any finite collection of function values follows a multivariate Gaussian distribution:

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

where:

- $m(x)$ is the mean function, representing the expected value of the function $g(x)$. A common choice is $m(x) = 0$, assuming no prior bias.
- $k(x, x')$ is the kernel (or covariance) function, encoding assumptions about the smoothness and structure of $g(x)$. Examples of kernels include the squared exponential (RBF) kernel:

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

  where:

  - σ_f^2: Signal variance, controlling the output scale of the function.
  - l: Length scale, determining how quickly the function changes with respect to x.

The prior captures our initial belief about the behavior of $g(x)$ before observing any data.

Likelihood $p(y \mid g(x))$

Definition 2.4.6 (Likelihood)**.** The likelihood relates the observed data (X, y) to the latent function $g(x)$. It assumes the observed outputs y are noisy evaluations of the true function:

$$y_i = g(x_i) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2),$$

where:

- y_i is the observed value of the objective function at x_i,
- $y = \{y_1, y_2, \ldots, y_n\}$: The corresponding noisy observations of the objective function.
- ϵ represents Gaussian noise with variance σ^2, capturing measurement or model evaluation noise.

- $X = \{x_1, x_2, \ldots, x_n\}$: The set of input points (e.g., hyperparameters) where the objective function $g(x)$ has been evaluated.

The likelihood is given by:

$$p(y \mid g(X)) = \prod_{i=1}^{n} \mathcal{N}(y_i \mid g(x_i), \sigma^2).$$

GP Posterior $p(g(x) \mid X, y)$

Definition 2.4.7 (Posterior Distribution)**.** The posterior combines the prior and likelihood using Bayes' theorem to update our belief about $g(x)$ after observing (X, y):

$$p(g(x) \mid X, y) \propto p(y \mid g(X))\, p(g(x)).$$

The posterior distribution of $g(x)$ given data (X, y) is still a Gaussian Process:

$$g(x) \mid X, y \sim \mathcal{GP}(\mu(x), \sigma^2(x)),$$

where:

- The posterior mean $\mu(x)$ represents the predicted value of $g(x)$:

$$\mu(x) = k(x, X)[K(X, X) + \sigma^2 I]^{-1} y,$$

  where:

  - $k(x, X)$ is the vector of covariances between x and observed points X,
  - $K(X, X)$ is the covariance matrix of observed points,
  - $\sigma^2 I$ accounts for the noise in the observations.

- The posterior variance $\sigma^2(x)$ quantifies the uncertainty at x:

$$\sigma^2(x) = k(x, x) - k(x, X)[K(X, X) + \sigma^2 I]^{-1} k(X, x).$$

The GP posterior provides:

- $\mu(x)$: The predicted value of $g(x)$,
- $\sigma^2(x)$: The uncertainty in the prediction.

Chapter 3

Integration Methods

3.1 Integration Methods: Motivation

Integration problems include many examples:

- **Moment computation:**
$$M_k(x) \;=\; \int x^k \, p(x)\, dx.$$
- **Evidence (marginal likelihood):**
$$p(X) \;=\; \int p(X \mid \theta)\, p(\theta)\, d\theta.$$
- **Relative entropy (KL divergence):**
$$\mathrm{KL}\big(p \,\|\, q\big) \;=\; \int \log\!\Big(\frac{p(x)}{q(x)}\Big)\, p(x)\, dx.$$
- **Prediction in time-series models:**
$$p(x_{t+1}) \;=\; \int p(x_{t+1} \mid x_t)\, p(x_t)\, dx_t.$$
- **Experimental design:**
$$\mathbb{E}_\theta[\, U(x) \,] \;=\; \int U(x;\theta)\, p(\theta)\, d\theta.$$
- **Planning:**
$$J^\pi\big(x(0)\big) \;=\; \int_0^T r\big(x(t), u(t)\big)\, x(0)\, dt.$$

Throughout, we focus on integration for computing expected values of the form

$$\mathbb{E}_{x\sim p}[f(x)] := \int f(x)p(x)\,dx, \tag{1}$$

where f is a **(possibly nonlinear)** function and x is a random variable. Expected values of the form (1) play a central role in machine learning. Here are some examples:

- **Statistical quantities.** For example, mean, variance, and moments of random variables x can be expressed as expected values (in terms of (1)):

$$\mathbb{E}_x[x] = \int xp(x)\,dx =: \mu, \tag{2}$$

$$\mathrm{Var}_x[x] = \int (x-\mu)^2 p(x)\,dx = \mathbb{E}_x[(x-\mu)^2], \tag{3}$$

$$M_k[x] = \int x^k p(x)\,dx = \mathbb{E}_x[x^k]. \tag{4}$$

- **Marginal likelihoods.** Marginal likelihoods, which are important quantities for model selection or model training, can also be expressed as expected values. Consider a supervised learning problem with training inputs $\mathcal{X} := \{x_1, \ldots, x_n\}$, corresponding training targets $\mathcal{Y} = \{y_1, \ldots, y_n\}$, and model parameters θ with a corresponding prior $p(\theta)$. Then, the marginal likelihood is given by

$$p(\mathcal{Y} \mid \mathcal{X}) = \int p(\mathcal{Y} \mid \mathcal{X}, \theta)\, p(\theta)\, d\theta \;=\; \underbrace{\mathbb{E}_{\theta\sim p(\theta)}[\, p(\mathcal{Y} \mid \mathcal{X}, \theta)\,]}_{(6)}, \tag{5}$$

$$= \mathbb{E}_{\theta\sim p(\theta)}[\, p(\mathcal{Y} \mid \mathcal{X}, \theta)\,]. \tag{3.1}$$

  Writing the marginal likelihood **(the predictive distribution of the training targets given the training inputs)** in terms of an expected value as in (6) makes it clear that the marginal likelihood is nothing but an expected likelihood, where the expectation is taken with respect to the **parameter prior** $p(\theta)$**.**

- **Predictions in a Bayesian model.** Considering the same supervised setting as above, we have a parameter posterior $p(\theta \mid \mathcal{X}, \mathcal{Y})$. To make predictions at a test input x_* we compute the **predictive distribution** of the corresponding target as

$$p(y_* \mid x_*, \mathcal{X}, \mathcal{Y}) \;=\; \int p\big(y_* \mid x_*, \theta\big)\, p(\theta \mid \mathcal{X}, \mathcal{Y})\, d\theta \;=\; \underbrace{\mathbb{E}_{\theta\sim p(\theta\mid\mathcal{X},\mathcal{Y})}[\, p(y_* \mid x_*, \theta)\,]}_{(8)}, \tag{7}$$

$$= \mathbb{E}_{\theta\sim p(\theta\mid\mathcal{X},\mathcal{Y})}[\, p(y_* \mid x_*, \theta)\,], \tag{3.2}$$

  which we can again interpret as an expected value, where we take the expectation with respect to the parameter posterior $p(\theta \mid \mathcal{X}, \mathcal{Y})$.

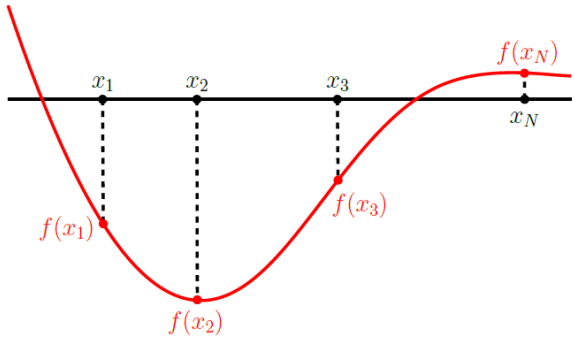


Figure 3.1: In **numerical integration**, we compute the expectation as a linear combination of function values $f(x_n)$ evaluated at nodes x_n.

The marginal likelihood (6) and the Bayesian prediction (8) resemble each other. The main difference is that for the marginal likelihood computation, we **compute an expectation (of training targets) with respect to the parameter prior**, whereas the predictive distribution in (8) is obtained by **an expectation (of a test target) with respect to the parameter posterior**.

Bayesian experimental design and Bayesian decision theory. In Bayesian experimental design and Bayesian decision theory, we are interested in computing expected utilities of the form

$$\int u(x,\theta)\, p(\theta)\, d\theta \;=\; \mathbb{E}_{\theta\sim p(\theta)}[\, U(x,\theta)\,] \tag{9}$$

in order to find optimal designs/decisions x.

The fundamental computational problem to be solved in all examples above is to determine the integral in (1). In nearly all interesting cases, this integral cannot be computed analytically, and approximations are thus required. The remainder of this chapter looks at a range of different ways to solve (1).

3.2 Numerical integration and Bayesian Quadrature

We consider solving integrals of the form

$$\int_a^b f(x)\, w(x)\, dx, \tag{10}$$

where $w(x) \geq 0$ is a **weight function**. We start by considering the case of $w(x) = 1$, i.e., we would solve the standard, unweighted definite integral.

In numerical integration, we normally approximate f, given a set of nodes x_n and corresponding function values $f(x_n)$, using an **interpolating function** that is easy to integrate, e.g., a low-degree polynomial.

- **Newton–Cotes approaches** use **low-degree polynomials and equidistant nodes** x_n to solve the integration problem. For example, if we approximate f locally using a constant, we obtain the midpoint rule. The trapezoidal rule approximates f using piecewise linear functions, and with a quadratic approximation, we obtain the Simpson rule.

- **Gaussian quadrature** approximates f by **a set of orthogonal polynomials**, and the nodes x_n are the roots of these polynomials.

- **Bayesian quadrature** approximates f by **a Gaussian process**, and the nodes x_n are defined by the user.

3.2.1 Newton–Cotes

Newton–Cotes approaches use **low-degree polynomials** to approximate f and **equidistant nodes** x_n to solve the integration problem in the interval $[a, b]$. In the following, we will discuss the trapezoidal rule and Simpson's rule as important examples of Newton–Cotes in more detail.

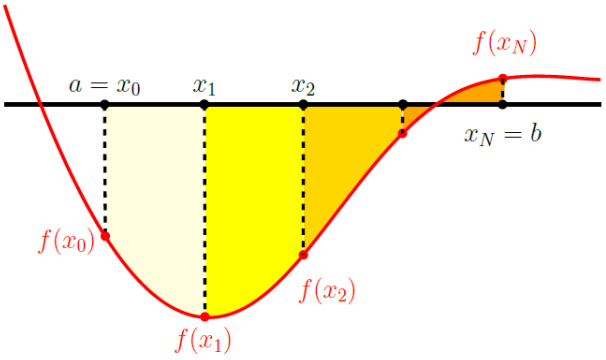


Figure 3.2: *Newton–Cotes quadrature approximates f by a low-degree polynomial at equidistant nodes x_n. We compute the integrals for each sub-interval analytically and sum them up to obtain the desired result.*

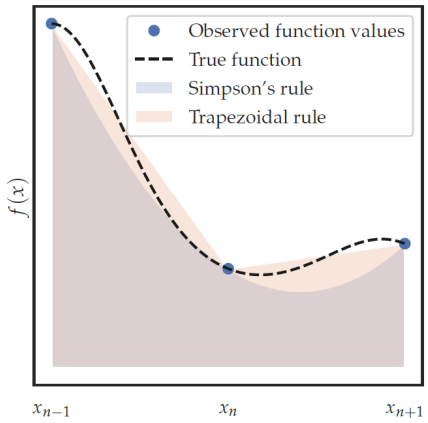


Figure 3.3: Locally linear (trapezoidal rule) and quadratic (Simpson's rule) approximation of the underlying function f, evaluated at nodes x_n.

Trapezoidal rule The trapezoidal rule partitions the interval $[a, b]$ into N segments. Nodes x_n are chosen equidistantly with $x_n - x_{n-1} =: h$. With a locally linear approximation of f, we compute the area between two nodes as

$$\int_{x_n}^{x_{n+1}} f(x)\,dx \approx \tfrac{h}{2}\big(f(x_n) + f(x_{n+1})\big), \tag{11}$$

which is the area of a trapezoid with corners $(x_n, x_{n+1}, f(x_{n+1}), f(x_n))$. Then, we obtain the value for the integral in (10) as

$$\int_a^b f(x)\,dx \approx \tfrac{h}{2}\big(f_0 + 2f_1 + \cdots + 2f_{N-1} + f_N\big), \tag{12}$$

where we used the shorthand notation $f_n := f(x_n)$.

The trapezoidal rule is straightforward, easy to implement, and fairly robust. It is a good choice, **if the integrand is non-smooth**, i.e., it exhibits discontinuities in its first derivative. The trapezoidal rule has an approximation error that shrinks in $O(1/N^2)$, where N **is the number of partitions**.

Simpson's rule Simpson's rule approximates the integrand locally with quadratic functions that connect triplets (f_n, f_{n+1}, f_{n+2}) at neighboring nodes x_n, x_{n+1}, x_{n+2}. The corresponding local integral can then be approximated as

$$\int_{x_n}^{x_{n+2}} f(x)\,dx \approx \tfrac{h}{3}\big(f_n + 4f_{n+1} + f_{n+2}\big) \tag{13}$$

so that we obtain the approximation

$$\int_a^b f(x)\,dx \approx \tfrac{h}{3}\big(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N\big) \tag{14}$$

of the integral in (10).

Simpson's rule yields **a more accurate approximation** than the trapezoidal rule if the integrand f is **smooth**. Simpson's rule has an approximation error that shrinks in $O(1/N^4)$, i.e., **it converges significantly faster than the trapezoidal rule**.

Example: Simpson vs. Trapezoidal Consider solving the integral

$$\int_0^1 \exp\big(-x^2 - \sin(3x)^2\big)\,dx \tag{15}$$

using the trapezoidal and Simpson rules. Figure below shows that Simpson's rule approximates the value of the integral in (15) well, and the error declines significantly faster than with the trapezoidal rule (as a function of the **number of nodes**).

Simpson's rule is the Newton–Cotes rule most often used in practice because it **retains algorithmic simplicity while offering an adequate degree of approximation**.

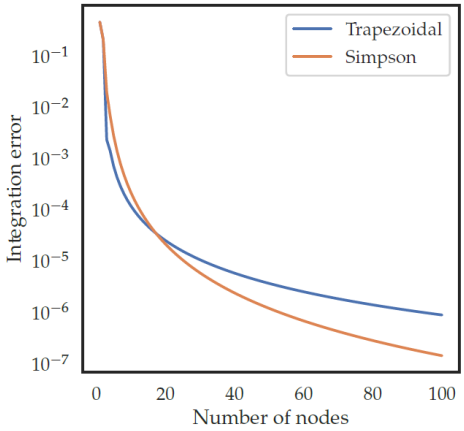


Figure 3.4: Comparison of trapezoidal and Simpson's rule for approximating integral (15)

Summary: Newton-Cotes Quadrature

- Approximate integrand between equidistant nodes with a low-degree polynomial (up to degree 6).

- Trapezoidal Rule: linear interpolation.
- Simpson's rule: quadratic interpolation
  - Better approximation than Trapezoidal and smaller integration error.

3.2.2 Gaussian Quadrature

Gaussian quadrature (irrelavant to GP) no longer relies on equidistant nodes, **giving higher accuracy**. It considers computing integrals of the form

$$\int_a^b f(x)\, w(x)\, dx, \quad w(x) \geq 0. \tag{16}$$

Gaussian quadrature approximates the integral by a weighted sum of function values $f(x_n)$ evaluated at nodes x_n (which are no longer need to be equidistant), i.e.,

$$\int_a^b f(x)\, w(x)\, dx \approx \sum_{n=1}^{N} w_n\, f(x_n), \tag{17}$$

where the nodes x_n and weights $w_n, n = 1, \ldots, N$, are chosen aiming to minimize the approximation error in (17).* **Then, Gaussian quadrature yields an exact result when the integrated f is a polynomial of degree up to $2N - 1$.**

Central Idea

The central idea of Gaussian quadrature is to choose the nodes x_n as **the roots of a family of orthogonal polynomials**. These roots can be found in a look-up table.

The corresponding optimal weights w_n are also known for commonly used orthogonal polynomials. Depending on the integration bounds a, b and the choice of the weight function $w(x)$, different families of orthogonal polynomials are used. Table below gives an overview of the most commonly used Gaussian quadrature rules and their corresponding families of orthogonal polynomials.

*The weights w_n and the weight function $w(x)$ are two different quantities and $w_n \neq w(x_n)$.

$[a, b]$	$w(x)$	Orthogonal polynomial
$[-1, 1]$	1	Legendre polynomials
$[-1, 1]$	$(1-x^2)^{-\frac{1}{2}}$	Chebychev polynomials
$[0, \infty]$	$\exp(-x)$	Laguerre polynomials
$[-\infty, \infty]$	$\exp(-x^2)$	Hermite polynomials

Figure 3.5

Example: Gauss–Hermite quadrature An important quadrature rule is the Gauss–Hermite quadrature. Here, we are interested in solving

$$\int f(x)\, \exp\bigl(-x^2\bigr)\, dx, \tag{18}$$

which we can also interpret as an expectation with respect to a Gaussian, i.e.,

$$\int f(x)\, \exp(-x^2)\, dx = \int f(x)\, \sqrt{2\pi}\, \mathcal{N}\bigl(x \mid 0, 1\bigr)\, dx = \sqrt{2\pi}\, \mathbb{E}_{x\sim\mathcal{N}(0,1)}\Bigl[\frac{f(x)}{\exp(-x^2)}\Bigr], \tag{19}$$

which is of high practical relevance in machine learning and many engineering disciplines. With a **change of variables**, we can compute expected values of function values under a general Gaussian distribution as

$$\mathbb{E}_{x\sim\mathcal{N}(\mu,\sigma^2)}\bigl[f(x)\bigr] \;\approx\; \frac{1}{\sqrt{\pi}} \sum_{n=1}^{N} w_n\, f\bigl(\sqrt{2}\, x_n + \mu\bigr). \tag{20}$$

In Gauss–Hermite quadrature, the nodes x_n are the roots of the physicists' version of the Hermite polynomial

$$H_N(x) \;=\; (-1)^N \exp\Bigl(\frac{x^2}{2}\Bigr)\, \frac{d^N}{dx^N} \exp\bigl(-x^2\bigr), \tag{21}$$

and the weights are given by

$$w_n \;=\; \frac{2^{N-1} N!\, \sqrt{\pi}}{N^2\, H_{N-1}^2(x_n)}. \tag{22}$$

Discussion. Comparing Gaussian quadrature with Newton–Cotes approaches, we see that Gaussian quadrature solves integration problems with **higher accuracy at the same number of function evaluations** $f(x_n), n = 1, \ldots, N$. In practice, a good choice of N is unknown in advance, so that one tries out different values of N. While Newton–Cotes approaches can re-use computations from smaller values of N, this does not hold for Gaussian quadrature, so that some of the computational advantages vanish.

Overall, Gaussian quadrature is the **method of choice for numerical integration in low dimensions (1-3)**. Therefore, Gaussian quadrature is often a key component in software packages to accurately solve low-dimensional integrals. Important application areas of Gaussian quadrature include

- computing probabilities for rectangular bivariate/trivariate Gaussian and t-distributions
- marginalization of a few hyper-parameters in latent-variable models
- making predictions with a Gaussian process classifier.

However, Gaussian quadrature also has limitations. It does not scale to more than **three-dimensional inputs** and it cannot deal with **noisy function evaluations** $f(x_n)$**.** To address these issues, there are alternative integration methods, such as Bayesian quadrature and Monte Carlo estimation.

3.2.3 Bayesian quadrature

In all quadrature schemes we discussed so far, we assumed that evaluating the function f at nodes x_n is **cheap**. That is typically the case when functions are analytic or simple lookup tables. However, we may encounter situations where evaluating a function is costly. For example, if we were to calculate the risk or expected utility of a new drug, evaluating f may require testing of a drug, which may take weeks or months. In these cases, we are interested in **computing expectations (such as risks and expected utilities) with a small number of function evaluations**.

Bayesian quadrature addresses this issue by formulating numerical integration as a **statistical inference problem**.[†] We are interested in finding something out about the integral value

$$Z := \int f(x)\, p(x)\, dx \;=\; \mathbb{E}_{x\sim p}[f(x)], \tag{22}$$

using previously observed data $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, where $y_n = f(x_n) + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. Bayesian quadrature places a **Gaussian process (GP) prior** on f and uses Bayes' theorem to determine a posterior distribution on f, which induces a posterior distribution on the value Z of the integral. Here we exploited the linearity of the integral (integral of a GP is another GP).

Variance of the integral value Z The GP on f in (22) induces a distribution $p(Z) = \mathcal{N}\big(Z \mid \mu_Z, \sigma_Z^2\big)$ on Z itself. The expected value of Z in (22) is

$$\mu_Z = \mathbb{E}_f[Z] \;=\; \mathbb{E}_{x\sim p}[\mathbb{E}_{f\sim \mathrm{GP}}[f(x) \mid \mathbf{x}]] \;=\; \mathbb{E}_{x\sim p}[\mu_{\mathrm{post}}(x)], \tag{23}$$

$$= \int k(x_*, X)\,(K + \sigma_\varepsilon^2 I)^{-1}\mathbf{y}\, p(x_*)\, dx_* \;=\; \mathbf{z}^\top (K + \sigma_\varepsilon^2 I)^{-1}\mathbf{y}, \tag{24}$$

[†]The research area of *probabilistic numerics* very much exploits the close relationship between statistical inference and quadrature.

where we exploited the law of iterated expectations. With a GP prior $f \sim \mathrm{GP}(0, k)$

$$\mu_{\text{post}}(\cdot) = k(\cdot, X)\,(K + \sigma_\varepsilon^2 I)^{-1}\mathbf{y} \tag{25}$$

is the GP's posterior mean function, where X and $\mathbf{y}$ are the training inputs and targets, respectively. Furthermore, k is the GP's kernel and K the corresponding kernel matrix. Then, we obtain the expected value of the integral Z as

$$\mu_Z \;=\; \int k(x_*, X)\, p(x_*)\, dx_* \left(K + \sigma_\varepsilon^2 I\right)^{-1}\mathbf{y} \;=\; \mathbf{z}^\top (K + \sigma_\varepsilon^2 I)^{-1}\mathbf{y}, \tag{26}$$

where

$$\mathbf{z}_n \;=\; \int k(x_n, x)p(x)\, dx_* \;=\; \mathbb{E}_{x\sim p}[k(x_n, x)] \tag{27}$$

is a *kernel expectation*, which computes the expected covariance between $f(x)$ and $f(x_n)$.

Variance of the integral value Z The variance of the integral Z is given by

$$\sigma_Z^2 \;=\; \mathrm{Var}[Z] \;=\; \iint k_{\text{post}}(x_*, x'_*)\, p(x_*)\, p(x'_*)\, dx_*\, dx'_* \;-\; \int k(x_*, x_*)\, p(x_*)\, dx_* \tag{28}$$

where $k_{\text{post}}(\cdot,\cdot)$ is the posterior covariance function

$$k_{\text{post}}(\cdot,\cdot) \;=\; k(\cdot, X)\,(K + \sigma_\varepsilon^2 I)^{-1}\, k(X, \cdot). \tag{29}$$

We now obtain the variance of Z (by using (29) in (28)) as

$$\mathrm{Var}_f[Z] = \sigma_Z^2 = \mathbb{E}_{x,x'\sim p} \underbrace{\left[k_{\text{post}}(x, x')\right]}_{\text{expected posterior covariance}}$$

$$= \iint \underbrace{k(x, x') - k(x, \mathbf{X})\mathbf{K}^{-1}k(\mathbf{X}, x')}_{\substack{\text{prior covariance} \\ \text{information from training data}}} p(x)p(x')\, dx\, dx'$$

$$= \iint k(x, x')p(x)p(x')\, dx\, dx' - \underbrace{\int k(x, \mathbf{X})p(x)\, dx}_{=z^\top}\, \mathbf{K}^{-1}\, \underbrace{\int k(\mathbf{X}, x')p(x')\, dx'}_{=z'}$$

$$= \mathbb{E}_{x,x'}[k(x, x')] - \mathbf{z}^\top\mathbf{K}^{-1}\mathbf{z}'$$

$$= \mathbb{E}_{x,x'}[k(x, x')] - \mathbb{E}_x[k(x, \mathbf{X})]\mathbf{K}^{-1}\mathbb{E}_{x'}[k(\mathbf{X}, x')].$$

where the kernel expectations $\mathbf{z}, \mathbf{z}'$ are defined in (27).

Computing Kernel Expectations Computing the kernel expectations $\mathbf{z}_n$ in (27) and (31) still requires solving an integration problem, but arguably this integration problem is easier to solve than the original one. Kernel expectations can be computed analytically in a few cases, e.g., when p is a Gaussian and the kernel $k(\cdot, \cdot)$ is a polynomial or an RBF kernel. If p is non-Gaussian (but we still use an RBF or polynomial kernel), we could use an importance-sampling scheme

$$\int f(x)\, p(x)\, dx \;=\; \int \frac{f(x)\, p(x)}{q(x)}\, q(x)\, dx, \tag{32}$$

where the GP now models $\frac{f(x)\, p(x)}{q(x)}$ where q is Gaussian and p arbitrary. If k is a kernel for which we cannot compute the kernel expectations (27) and (31) analytically, we may be able to resort to numerical integration (e.g., using the techniques discussed earlier) or Monte Carlo integration. Kernel expectations appear not only in Bayesian quadrature but also in kernel MMD, time-series analysis with Gaussian processes, Deep Gaussian processes, or model-based reinforcement learning with GPs.

While the use of Gaussian process priors may seem unwieldy, they also allow numerical integration to be applied to situations where observations of f are noisy, evaluating f is expensive, we wish to exploit correlations between function values, or we know something about the structure of f, which can be exploited when choosing the GP's mean and covariance functions.

Iterative procedure: Where to measure f next?

- **Define an acquisition function** (similar to Bayesian optimization)
- **Example:** Choose next node x_{n+1} so that the **variance of the estimator is reduced maximally** (e.g., O'Hagan, 1991; Gunter et al., 2014)

$$x_{n+1} = \arg\max_x \; \underbrace{\mathbb{V}[Z \mid \mathcal{D}]}_{\text{current variance}} \; - \mathbb{E}_{y_*}\Big[\underbrace{\mathbb{V}[Z \mid \mathcal{D} \cup \{(x_*, y_*)\}]}_{\text{new variance}}\Big]$$

Example with EmuKit (Paleyes et al., 2019)

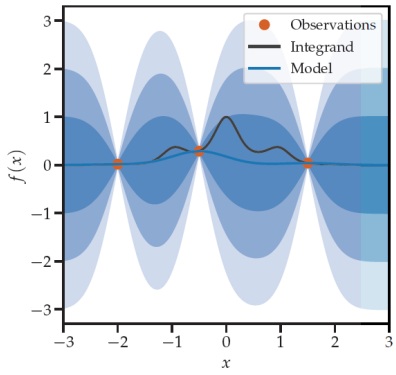


Figure 3.6: Gaussian process fitting with EmuKit

$$Z = \int_{-3}^{3} e^{-x^2 - \sin^2(3x)}\, dx$$

- Fit Gaussian process to observations $f(x_1), \ldots, f(x_n)$ at nodes $x_1, \ldots, x_n$
- Determine $p(Z)$
- Find and include new measurement
    1. Find optimal node x_{n+1} by maximizing an acquisition function
    2. Evaluate integrand at x_{n+1}
    3. Update GP with $(x_{n+1}, f(x_{n+1}))$

Updated example with EmuKit (Paleyes et al., 2019)

$$Z = \int_{-3}^{3} e^{-x^2 - \sin^2(3x)}\, dx$$

- Fit Gaussian process to observations $f(x_1), \ldots, f(x_n)$ at nodes $x_1, \ldots, x_n$
- Determine $p(Z)$
- Find and include new measurement
- Compute updated $p(Z)$

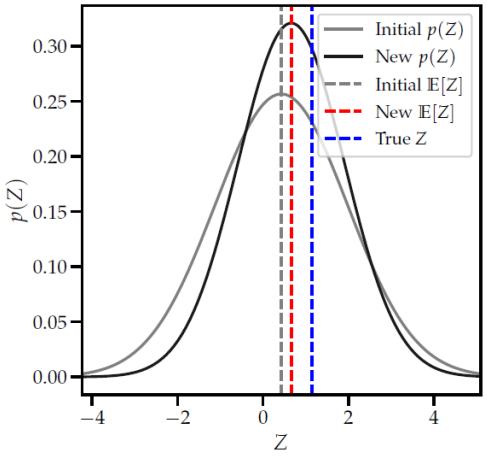


Figure 3.7: Updated Gaussian process estimation

3.2.4 Summary

Numerical integration methods are good for solving low-dimensional integration problems quickly and accurately. The central approximation was

$$\int_a^b f(x)\,dx \;\approx\; \sum_{n=1}^{N} w_n\, f(x_n). \tag{33}$$

- Newton–Cotes methods use equidistant nodes x_n and low-degree polynomial approximations of f. In Gaussian quadrature, the nodes x_n are the roots of interpolating orthogonal polynomials.
- Bayesian quadrature formulates integration as a statistical inference problem and uses a global approximation of f by means of a GP. These methods can be applied to low and moderate-dimensional integration problems.
- For higher-dimensional settings (maybe up to 100–1000 dimensions), we may want to resort to Monte Carlo integration (Section 3) to get an exact solution in the limit.

Table below gives an overview of when various integration techniques can be applied.

However, Monte Carlo estimation can be fairly slow, especially when we have to use MCMC methods to **draw samples from high-dimensional complex distributions**. In this case, we may want to consider approximation techniques that will not give us the exact solution to our integral problem, but will give us an approximate solution relatively fast. Algorithms that fall into this category make approximating assumptions on p and include variational inference or the Laplace approximation.

Table 3.1: *Table 3: Overview of integration techniques based on the dimensionality of the input dimension.*

Dimensionality	Method
≤ 3	Gaussian quadrature
≤ 10	Bayesian quadrature
≤ 1000	Monte Carlo integration

3.3 Monte Carlo integration

Monte Carlo methods a computational techniques that make use of random numbers. We consider two typical problems:

1. Generate samples $x^{(s)}$ from a given probability distribution $p(x)$, e.g., for simulation (generative models) or particle representations of distributions.
2. Computing expectations of functions under that distribution, i.e., we need to solve integrals of the form

$$\mathbb{E}_{x\sim p}[f(x)] \;=\; \int f(x)\, p(x)\, dx. \tag{34}$$

In the context of this note, we will exclusively focus on the second problem, i.e., we will avoid the (more complicated) problem of generating samples from a probability distribution.

Examples where expectations of the form (34) are required include:

- **Computation of moments** (e.g., means, variances) of random variables. The kth moment is defined as

$$M_k(x) = \int x^k p(x)\, dx = \mathbb{E}_{x\sim p}[x^k]. \tag{35}$$

- **Computation of the marginal likelihood.** For parameters θ and corresponding prior $p(\theta)$, the marginal likelihood is

$$p(X) = \int p(X \mid \theta)\, p(\theta)\, d\theta = \mathbb{E}_{\theta\sim p(\theta)}[p(X \mid \theta)]. \tag{36}$$

  In this sense, the marginal likelihood can also be thought of as an averaged likelihood $p(X \mid \theta)$.

- **Predictions in a Bayesian model.** Given training data X and parameters θ, we make a prediction of a test value x_* as follows:

$$p(x_* \mid X) = \int p(x_* \mid \theta)\, p(\theta \mid X)\, d\theta = \mathbb{E}_{\theta\sim p(\theta|X)}[p(x_* \mid \theta)]. \tag{37}$$

Here, $p(\theta \mid X)$ is the parameter posterior.

In all cases, moment computation, marginal likelihoods and Bayesian predictions, we need to compute expectations of the form (34).

The key idea behind Monte Carlo integration is to use random samples to approximate an integral. Specifically, Monte Carlo methods compute expectations by **statistical sampling**, so that

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)\, p(x)\, dx \approx \frac{1}{S}\sum_{s=1}^{S} f\big(x^{(s)}\big), \quad x^{(s)} \sim p(x). \tag{38}$$

Properties of Monte Carlo Estimation

The Monte Carlo estimator (38) is **unbiased and asymptotically consistent**, i.e.,

$$\lim_{S\to\infty} \frac{1}{S}\sum_{s=1}^{S} f\big(x^{(s)}\big) = \mathbb{E}_{x\sim p}[f(x)] + \varepsilon, \tag{39}$$

where the error ε is Gaussian distributed and its variance shrinks linearly in $O(1/S)$, independent of the dimensionality.

Monte Carlo Estimation: How to Sample?

$$\mathbb{E}[f(x)] = \int f(x)p(x)\, dx \approx \frac{1}{S}\sum_{s=1}^{S} f(x^{(s)}), \quad x^{(s)} \sim p(x)$$

- How do we get these samples?
- Sampling from simple distributions
  - Use libraries if the distribution has a “name”
- Sampling from complicated distributions
  - Rejection sampling (does not scale to high dimensions)
  - Importance sampling (does not scale to high dimensions)
  - Markov chain Monte Carlo (MCMC)

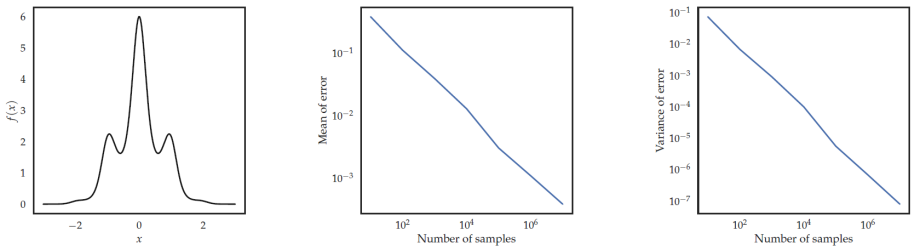


Figure 3.8: Graphical representation of Monte Carlo estimation

Example

$$Z = \mathbb{E}_x[f(x)] = \int f(x)p(x)\,dx = \int_{-3}^{3} 6\exp\left(-x^2 - \sin(3x)^2\right)\mathcal{U}[-3,3]\,dx$$

- Monte-Carlo estimator

$$\mathbb{E}_{x\sim\mathcal{U}}[f(x)] \approx \frac{1}{S}\sum_{s=1}^{S} f(x^{(s)}), \quad x^{(s)} \sim \mathcal{U}[-3,3]$$

Consideration

$$\mathbb{E}[f(x)] \approx \frac{1}{S}\sum_{s=1}^{S} f(x^{(s)}), \quad x^{(s)} \sim p(x)$$

- Require many samples to get a good estimate of the value of the integral
- Design efficient samplers (computationally efficient, low variance)
- Function needs to be cheap to evaluate
- Good for learning, if we are just interested in an **unbiased estimator**
- Estimator does not take the locations of the samples into account
  - Could be problematic in small-sample regimes

Summary

- Random numbers to compute expectations
- **Estimator has nice properties (e.g., unbiased estimator, asymptotically consistent)**
- Scales to very high dimensions
- General approach and straightforward
- Generating samples is the key challenge (not covered here)

3.4 Normalizing flows

Normalizing flows provide a great way to build complex distributions from simple distributions via a flow of successive (invertible) transformations.

Flow-based models are special versions of latent-variable models. Here, the distribution $p(x)$ of the data is obtained by marginalizing out a latent variable z according to

$$p(x) = \int p(x \mid z)\, p(z)\, dz. \tag{40}$$

This latent-variable model consists of a prior distribution $p(z)$ on the latent variable (code) z and a "prescription" $p(x \mid z)$ of how to generate data x for a given realization z of the latent variable. Instead of "hand-crafting" a complicated likelihood $p(x \mid z)$ directly, a normalizing flow model uses a composition of (simple) invertible transformations to map z into x.

Instead of modeling a complicated data distribution $p(x)$ directly, which is often hard, we can parametrize the prior $p(z)$ and the generator $p(x \mid z)$ as an indirect way to get $p(x)$.

3.4.1 Change of Variables

Key Idea

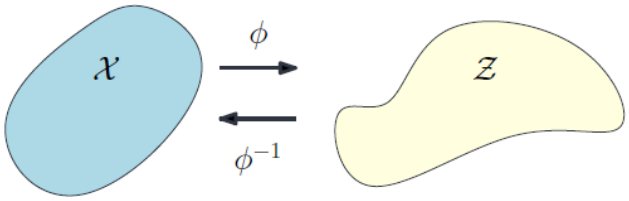


Figure 3.9: Change of Variables.

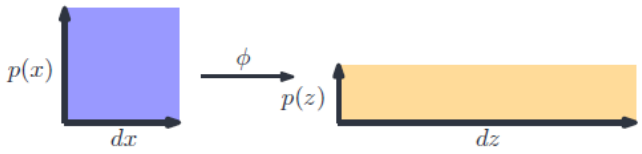


Figure 3.10: Jacobian Determinant.

Key idea
Transform random variable X into random variable Z using an invertible transformation ϕ, while keeping track of the change in distribution.

- Distribution p_X induces distribution p_Z via transformation ϕ
- Distribution p_Z induces distribution p_X via transformation ϕ^{-1}

3.4.2 Jacobian Determinant

$$\left|\det\left(\frac{dz}{dx}\right)\right| = \left|\det\left(\frac{d\phi(x)}{dx}\right)\right|$$

tells us how much the domain dx is stretched to dz. It's the scaling factor between two differnet domains.

How it Works

Constraint: volume preservation

$$\int_{\mathcal{X}} p_X(x)\,dx = 1 = \int_{\mathcal{Z}} p_Z(z)\,dz$$

Volume preservation: Rescale p_Z by the inverse of the Jacobian determinant

$$p_Z(z) = p_X(x)\left|\det\left(\frac{d\phi(x)}{dx}\right)\right|^{-1}$$

Considerations

- Express target distribution p_Z in terms of known distribution p_X and the Jacobian determinant of an invertible mapping ϕ
- No need to invert ϕ explicitly

- Generate expressive distributions p_Z by simple p_X and flexible transformation ϕ

3.4.3 Normalizing Flow

How it works

A normalizing flow is a specific way to define $p(z)$ and, more importantly, $p(x \mid z)$. Focusing on $p(x \mid z)$, a normalizing flow defines a chain of K transformations

$$z = z_K = f_K \circ f_{K-1} \circ \cdots \circ f_1(z_0) \tag{41}$$

with $f_k : \mathbb{R}^D \to \mathbb{R}^D$ invertible and $z_0 \sim p_0$, where p_0 is called a *base distribution*. The path $(z_0, \ldots, z_K)$ of random variable $z_k = f_k(z_{k-1}), k = 1, \ldots, K$, is called a *flow*, and the path $(p_0, \ldots, p_K)$ of the corresponding distributions is called a *normalizing flow*. Figure **??** illustrates a normalizing flow.

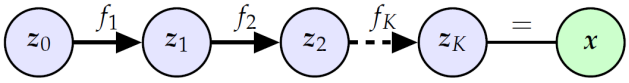


Figure 3.11: Normalizing flow. The normalizing flow defines a chain of invertible transformations f_k of a simple base distribution $p_0(z_0)$ into a complex data distribution $p(x)$.

By repeated application of the change-of-variables-trick[‡], we obtain the marginal distribution at the "end" of the flow as

$$p(x) = p(z_K) = p(z_0) \prod_{k=1}^{K} \left| \det \frac{dz_k}{dz_{k-1}} \right|^{-1} = \frac{p(z_0)}{\prod_{k=1}^{K} \left| \det \frac{df_k(z_{k-1})}{dz_{k-1}} \right|}, \tag{42}$$

and the entropy can be efficiently computed as

$$\log p(x) = \log p(z_K) = \log p(z_0) - \sum_{k=1}^{K} \log \left| \det \frac{df_k(z_{k-1})}{dz_{k-1}} \right|. \tag{43}$$

Note that we do not need to invert f_k explicitly, since we are only interested in the determinant

$$\det\left(\frac{df_k^{-1}(z_k)}{dz_k} \right) = \frac{1}{\det \frac{df_k(z_{k-1})}{dz_{k-1}}} \tag{44}$$

for which we only need the forward transformation f_k.

Under mild assumptions, a normalizing-flow model can express any distribution, even if the base distribution p_0 is simple.

[‡]See Appendix B for a brief introduction.

Example Illustration with PyMC3

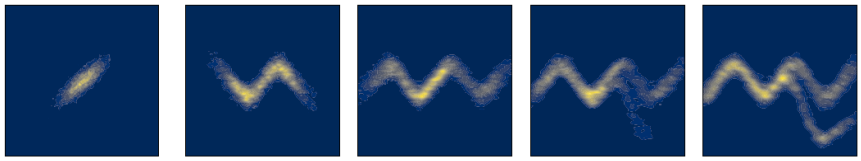


Figure 3.12: Illustration of a normalizing flow with 1, 2, 3, 7, and 12 planar flows; generated using a PyMC3 tutorial

We consider an example where $p_0 = \mathcal{N}(0, I)$. We successively apply planar flows

$$z_k = f_k(z_{k-1}) = z_{k-1} + u\sigma\big(w^\top z_{k-1} + b\big). \tag{45}$$

Figure below shows the transformed distributions from Gaussian to complex distribution after application of 1, 2, 3, 7, and 12 applications of the planar flow (45). The more often the planar flow is applied the more complex distributions we can model.

Computing expectations

Expectations with respect to p_k can be computed without explicitly knowing p_k (law of the unconscious statistician; LOTUS). Assume we want to compute an expected loss $E_{p_k}[l(z)]$, then we can get this as

$$E_{p_x}[l(x)] = E_{p_k}\big[l(z_K)\big] = E_{p_0}\big[l\big(f_K \circ f_{K-1} \circ \cdots \circ f_1(z_0)\big)\big]. \tag{46}$$

We can obtain this expectation as a Monte Carlo estimate. First, we generate a sample $z_0^{(s)} \sim p_0(z_0)$ from the base distribution and then push that sample through $f_1, \ldots, f_K$ (ancestral sampling), which yields a valid sample $x^{(s)} \sim p_X(x)$ from our target distribution. Using these samples, we can use Monte Carlo integration (see Section 3) to compute the expected value in (46).

Computational aspects

The computational challenge in normalizing flows lies in the computation of the (log-)determinant terms of the Jacobians, e.g., (43). These determinants can be computed efficiently, i.e., in linear time, if the Jacobian is diagonal or a triangular matrix, in which case the determinant is the product of the diagonal elements. To construct Jacobians that are triangular, we can define functions f_k in a particular way.

Looking at the Jacobian $df_k/dz_{k-1} = dz_k/dz_{k-1},\ z_{k-1} \in \mathbb{R}^D$, we get

$$\begin{pmatrix} \frac{\partial z_k^{(1)}}{\partial z_{k-1}^{(1)}} & \frac{\partial z_k^{(1)}}{\partial z_{k-1}^{(2)}} & \cdots & \frac{\partial z_k^{(1)}}{\partial z_{k-1}^{(D)}} \\ \frac{\partial z_k^{(2)}}{\partial z_{k-1}^{(1)}} & \cdots & & \vdots \\ \vdots & & & \\ \frac{\partial z_k^{(D)}}{\partial z_{k-1}^{(1)}} & \cdots & & \frac{\partial z_k^{(D)}}{\partial z_{k-1}^{(D)}} \end{pmatrix} \in \mathbb{R}^{D\times D}. \tag{47}$$

To make this Jacobian (collection of partial derivatives) a triangular matrix, we require the partial derivatives $\frac{\partial z_k^{(d')}}{\partial z_{k-1}^{(d)}}$ in the upper-triangular matrix (in red) to vanish.

Autoregressive Flows

One way to achieve this is to use an autoregressive model (applied to the dimensions of z_k) to sequentially build up z_k. In this autoregressive flow, each dimension d of z_k can be written as

$$z_k^{(d)} = f_k^{(d)}\big(z_{k-1}^{(1)}, \ldots, z_{k-1}^{(d)}\big) = f_k^{(d)}\big(z_{k-1}^{(\leq d)}\big). \tag{48}$$

A special example of an autoregressive flow uses the transformation

$$z_k^{(d)} = \tau\big(z_{k-1}^{(d)}\, c\big(z_{k-1}^{(\leq d)}\big)\big), \tag{49}$$

where τ is a *transformer* and c a *conditioner*. The conditioner parametrizes the transformer, but does itself not need to be invertible.

Applications

We can think of the re-parametrization trick used in **variational autoencoders (VAEs)** as a special case of a normalizing flow while no assumption of underlying data distribution, so that the key ideas of normalizing flows are used to **perform (variational) inference** in deep generative model. In a VAE, $p(z)$ is a complex posterior over latent variables z, and f transforms a simple input distribution (for example, a standard normal distribution) over x into a complex approximate posterior $q(z)$. Other application areas of normalizing flows include graph neural networks, parallel WaveNet , but also neural ODEs . Normalizing flows have also been generalized to flows on manifolds.

3.4.4 Summary

Normalizing flows provide a constructive way to **generate rich data distributions**. The key idea is to transform a simple distribution using a flow of successive (invertible) transformations. Key

ingredient is the change-of-variables trick. From a practical (computational) perspective, Jacobians can be computed efficiently, if the transformations are defined appropriately. Normalizing flows can be used as a generator and inference mechanism.

3.5 Inference in time series models

3.5.1 Setting

As an application of integration, we consider the problem of inference in time series. We focus on discrete-time settings. Assume a distribution $p(x_0)$ of the initial state x_0 and a Markovian state evolution

$$x_{t+1} = f(x_t) + \epsilon, \quad x_0 \sim p(x_0), \quad \epsilon \sim \mathcal{N}(0, Q) \tag{50}$$

where f is a transition function and $p(x_0)$ the distribution of the initial state.[§]

We are often interested in computing an expected utility

$$\mathbb{E}_\tau\big[U(\tau)\big], \tag{51}$$

where the expectation is taken with respect to **state trajectories** $\tau := (x_0, \ldots, x_T)$. A trajectory is a sequence of states from the initial state at time 0 to a final state at time T. Examples include:

- Reinforcement learning and optimal control, where U is a long-term cost/reward function
- Logistics, when we forecast demand and associated costs
- Weather/climate forecasts, which can be used for assessing the risk level of flooding.

The main challenge in all these scenarios is to make long-term predictions, which requires **uncertainty propagation**. Uncertainty could enter through uncertain initial states, noise in the system or, if we learn f, uncertainty about some model parameters.

3.5.2 Approaches

The problem we consider is to determine a (predictive) distribution of the state x_t for $t = 1, \ldots, T$. There are multiple ways to look at this problem:

1. **Deterministic inference via Iterative prediction** (similar to Kalman filtering). In this *distributional perspective*, we compute marginal distributions $p(x_1), \ldots, p(x_T)$ iteratively. In the context of this tutorial, we consider Gaussian marginals that are parametrized by a mean μ_t and a covariance matrix Σ_t.

[§]We make the simplifying assumption of additive Gaussian noise in (50) as this greatly simplifies some of the problems we have to solve.

2. **Stochastic inference via Trajectory sampling** (done in RL quite often). In this *pathwise perspective*, we generate multiple trajectories $\tau^{(s)} := (x_1^{(s)}, \dots, x_T^{(s)})$ from which we can extract the marginals $p(x_1), \dots, p(x_T)$.

In the following, we will discuss two approaches for making long-term predictions: *Deterministic* and *stochastic* approximate inference. In deterministic approximate inference, we iteratively compute the state marginal distributions $p(x_1), \dots, p(x_T)$, compute expected utilities $\mathbb{E}_{x_t}[u(x_t)]$ at every time step, and sum them up to obtain

$$\mathbb{E}_T\big[U(\tau)\big] = \sum_{t=0}^{T} \mathbb{E}_{x_t}\big[u(x_t)\big] = \sum_{t=0}^{T} \int u(x_t)\, p(x_t)\, dx_t. \tag{52}$$

In stochastic approximate inference, we generate sample trajectories $\tau^{(s)}$ and use **Monte Carlo integration** to determine the expected utility (51) via

$$\mathbb{E}_T\big[U(\tau)\big] \approx \frac{1}{S} \sum_{s=1}^{S} U\big(\tau^{(s)}\big). \tag{53}$$

3.5.3 Deterministic approximate inference

Predicting the (marginal) distributions $p(x_1), \dots, p(x_T)$ of the state trajectory iteratively, requires us to repeatedly solve the following integral:

$$p(x_{t+1}) = \int p(x_{t+1} \mid x_t)\, p(x_t)\, dx_t \tag{54}$$

$$= \int \mathcal{N}\big(x_{t+1} \mid f(x_t), Q\big)\, p(x_t)\, dx_t. \tag{55}$$

which has no closed-form solution for non-linear f.

Commonly, the marginals $p(x_t)$ is approximated by Gaussians, i.e., $p(x_t) \approx \mathcal{N}(\mu_t, \Sigma_t)$. Classical approaches for determining an approximate Gaussian distribution include *linearization* (e.g., extended Kalman filter), *unscented transformation*(e.g., unscented Kalman filter), and *moment matching* (e.g., assumed density filter).

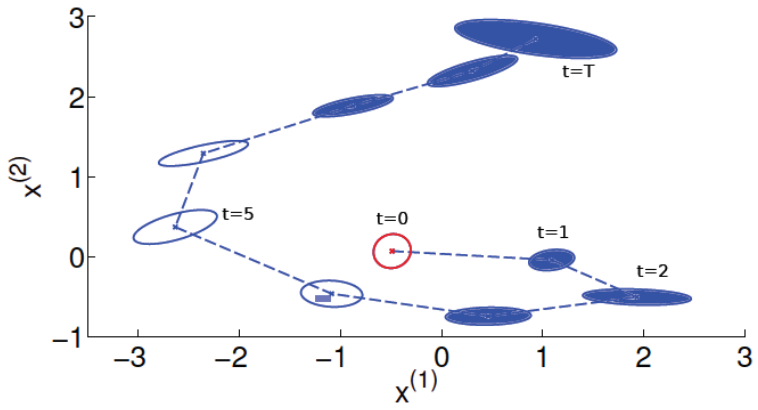


Figure 3.13: Iterative Gaussian Approximation.

Linearization: Approximate f Linearization exploits that a linear/affine transformation of a Gaussian distribution remains Gaussian. With a Gaussian distribution $p(x_t) = \mathcal{N}(\mu_t, \Sigma_t)$, an affine transformation of x_t via

$$x_{t+1} = A\, x_t + b \tag{56}$$

results in a Gaussian distribution $p(x_{t+1}) = \mathcal{N}\big(A\,\mu_t + b,\, A\,\Sigma_t\,A^\top\big)$.

In the context of the time-series model (50), function f is a non-linear transformation of x_t. To locally approximate f using a linear function f_{lin} (thereby turning the nonlinear transformation into a linear one), we can use a **first-order Taylor-series expansion** around μ_t to obtain

$$f_{\text{lin}}(x) = f(\mu_t) + J(\mu_t)\,(x - \mu_t), \tag{57}$$

where

$$J(\mu_t) := \frac{df}{dx}\Big|_{x=\mu_t} \tag{58}$$

is the Jacobian of f evaluated at $x = \mu_t$. Approximating f by f_{lin} allows us to compute an approximate Gaussian distribution of x_{t+1} as

$$p(x_{t+1}) \approx \mathcal{N}\big(\mu_{t+1},\, \Sigma_{t+1}\big) \tag{59}$$

$$\mu_{t+1} = \mathbb{E}\big[f_{\text{lin}}(x_t)\big] = f(\mu_t) \tag{60}$$

$$\Sigma_{t+1} = \mathbb{E}_{x_t}\big[f_{\text{lin}}(x_t)\big] = J(\mu_t)\,\Sigma_t\,J(\mu_t)^\top + Q, \tag{61}$$

where the additional Q in the expression of the predictive covariance is due to the additive noise term in (50).

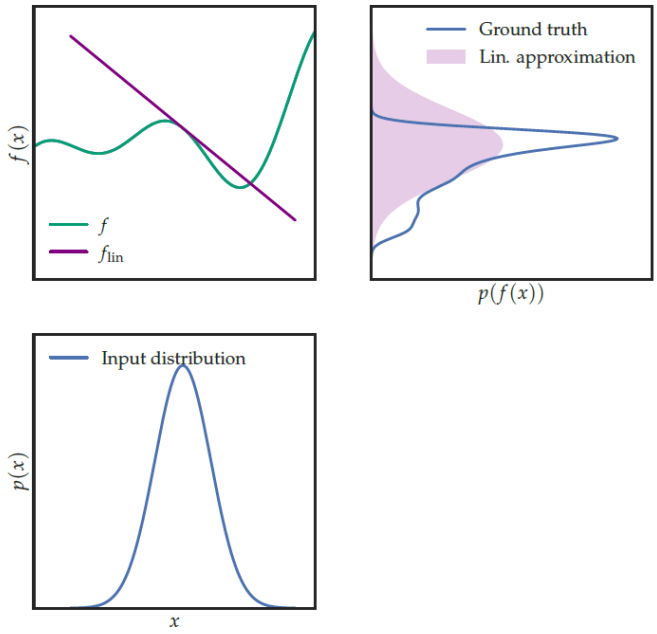


Figure 3.14: Illustration of linearization. Prediction by locally linearizing a nonlinear function and then analytically pushing the Gaussian through the linearized function.

Computing a Gaussian approximate predictive distribution via linearization is conceptually straightforward, but **it requires a differentiable transition function f.**

Moreover, in practice, the linearization approach tends to **underestimate the true covariance**, which results in overconfident predictions. This can lead to problems in downstream applications that rely on reasonable uncertainty estimates.

Computing the predictive distribution scales in $\mathcal{O}(D^3)$, where D is the dimensionality of the state. Linearization, as a way to approximate (55), is widely used in engineering. For example, the extended Kalman filter (EKF) exploits linearization to compute a posterior distribution on unobserved states. The EKF is at the core of GPS and was used in multiple Apollo missions.

Unscented transformation: Approximate $p(x_t)$ An alternative approach to linearization to approximate the integral in (55) is the *unscented transformation*. The key idea behind the unscented transformation is that instead of approximating function f (which linearization does), we can approximate $p(x_t)$. More specifically, the unscented transformation represents $p(x_t)$ using **a small set of $2D + 1$ *sigma points***, which we can think of as deterministically chosen particles. We then **evaluate the original function f at those sigma points** and **compute a (weighted) Monte Carlo estimate of the predictive mean and covariance of the mapped sigma points.**

Sigma points are deterministically chosen as

$$X_t = \{\mu_t + \alpha(\sqrt{\Sigma_t})_i,\ i = 1, \ldots, D\}, \tag{62}$$

where $\sqrt{\Sigma_t}$ is a square-root of the covariance matrix (the Cholesky factor would be one option), and α spreads the sigma points symmetrically around the mean. The sigma points satisfy some nice properties, such that their mean and variance match μ_t and Σ_t of the input distribution.

The predictive mean and covariance are then given by

$$\mu_{t+1} \approx \sum_{d=1}^{2D+1} w_d^m\, f\big(X_t^{(d)}\big) \tag{63}$$

$$\Sigma_{t+1} \approx \sum_{d=1}^{2D+1} w_d^c \Big(f\big(X_t^{(d)}\big) - \mu_{t+1}\Big)\Big(f\big(X_t^{(d)}\big) - \mu_{t+1}\Big)^\top, \tag{64}$$

respectively, where D is the dimensionality of x_t, $X_t^{(d)}$ are the sigma points and w_d^m and w_d^c are weights for the mean and covariance, respectively.

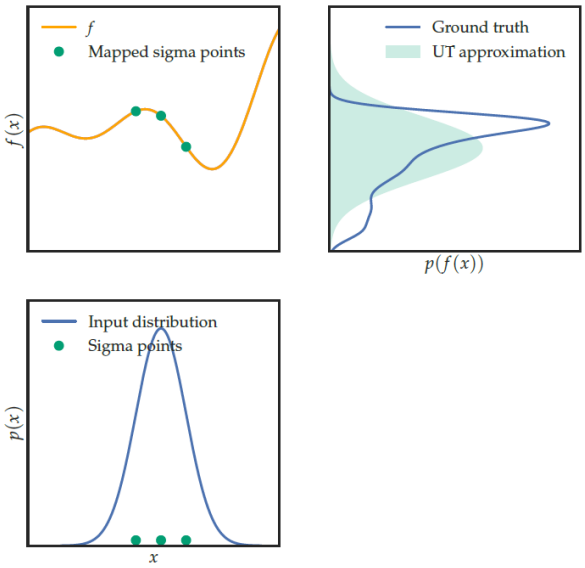


Figure 3.15: Illustration of unscented transformation. Prediction by approximating the input distribution by means of sigma points.

The unscented transformation:

- Does **not require an explicit calculation of the Jacobian**
- Achieves a slightly higher accuracy than linearization for the covariance estimate
- UT approximations are accurate to the third order for Gaussian inputs for all non-linearities
- For non-Gaussian inputs, approximations are accurate to at least second order
- **Is not a Monte Carlo method since the unscented transformation does not use any random numbers; sigma points are deterministically determined.**

Moment matching In *moment matching*, we choose a distribution that is easy for us to work with and project the true predictive distribution onto this distribution family. The moments of the approximate distribution (which lies in the distribution family of our choice) are found by minimizing the KL divergence between the true posterior and the distribution of our choice. In the end, this comes down to moment matching.

To compute the mean and covariance of x_{t+1}, both linearization and the unscented transformation implicitly approximate the joint distribution $p(x_t, x_{t+1})$ using a Gaussian distribution [**?**, Deisenroth et al., 2012]. However, the moments of this approximate Gaussian distribution are not exact. It is sometimes possible to compute the moments of the joint distribution analytically, which then yields a better approximation in a KL sense. This means that the approximating (Gaussian) distribution is the best (unimodal) approximation to the true (unknown) distribution.

Assuming $p(x_t)$ is Gaussian, moment matching computes the exact mean and covariance of the predictive distribution

$$p(x_{t+1}) = \int p(x_{t+1} \mid x_t)\, p(x_t)\, dx_t \tag{65}$$

and approximates $p(x_{t+1})$ by means of a Gaussian $\mathcal{N}\big(\hat{\mu}_{t+1}, \hat{\Sigma}_{t+1}\big)$, where the mean and the covariance correspond to the mean and the covariance of $p(x_{t+1})$. Note that this is not true for linearization or the unscented transformation. This Gaussian approximation can be then taken as the new input distribution, so that this approximation scheme, when iterated, yields approximate Gaussian state distributions $p(x_1), \ldots, p(x_T)$.

Given that $p(x_t)$ is Gaussian, not all hope is lost to compute the mean and covariance of $p(x_{t+1})$ in (65). For example, if the transition function f is a polynomial, a radial-basis function network (with Gaussian basis functions), or a Fourier series, these quantities can be computed analytically. Gradshteyn and Ryzhik [**?**, 2007] provide a great overview of analytical solutions for these (and more) settings. It is also possible to use Monte Carlo integration to compute the moments as it is easy to sample from a Gaussian $p(x_t)$.

Moment matching is used in the context of *assumed density filtering*, *expectation propagation* (a message passing algorithm), reinforcement learning and robotics and for inference in Bayesian neural networks, etc.

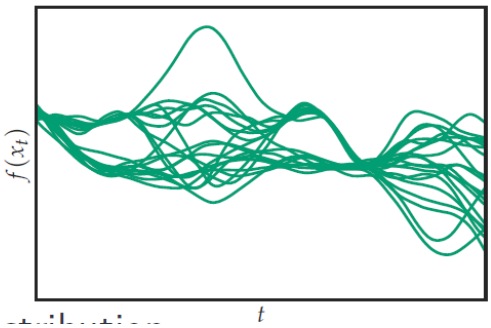


Figure 3.16: Time evolutions of a state. Trajectories are generated according to (67). The distribution over trajectories is represented by the collection/ensemble of samples.

3.5.4 Stochastic approximate inference

The expectation (51) requires averaging over state trajectories $\tau = (x_0, \dots, x_T)$. In stochastic approximate inference, we would sample trajectories $\tau^{(s)}$ and compute a Monte Carlo estimate

$$\mathbb{E}_T\big[U(\tau)\big] \approx \frac{1}{S}\sum_{s=1}^{S} U\big(\tau^{(s)}\big). \tag{66}$$

To sample a trajectory, we start with a sample $x_0^{(s)} \sim p(x_0)$. Subsequently, we will need to sample

$$x_t^{(s)} \sim p\big(x_t \mid x_0^{(s)}, \dots, x_{t-1}^{(s)}\big) = p\big(x_t \mid x_0^{(s)}\big), \tag{67}$$

where the latter equality holds for Markovian systems. Figure below illustrates a distribution over trajectories represented by a collection/ensemble of these sampled trajectories. We do not require parametric assumptions on the distribution of trajectories; however, we will have to store all samples, which may result in memory issues. Sequential Monte Carlo and particle filtering rely on this kind of inference.

3.5.5 Discussion: Long-term Predictions

Table 4 gives a brief overview of properties of deterministic and stochastic inference techniques:

- Deterministic approximations typically have a parametric representation of the marginals, whereas in the stochastic case, we extract the marginals from particles/samples.
- Deterministic approaches introduce bias, whereas stochastic approaches do not.

Table 3.2: **Table 4** Properties of deterministic and stochastic approximations.

	Deterministic	**Stochastic**
Marginal representation	Parametric	Particles
Bias	Yes	No
Time correlation	No	Yes
Speed	Fast	(Slow)
Parallelization		Easy
Memory consumption	Low	(High)
Gradients	Deterministic	Stochastic

- Typically, deterministic approaches are fast (they only require a single "sweep" of computations) while stochastic approaches could be considered slower. That said, typically, each single computation in the stochastic approach is fairly cheap and these computations can be easily parallelized. Therefore, the wall-clock time of stochastic approaches may not be too bad.
- Memory consumption in deterministic approaches is low compared to stochastic approaches, because one only has to store the moments of the marginals, whereas in the stochastic case one needs to keep all particles around.
- Looking at gradients, deterministic approximate inference gives rise to deterministic gradients, while stochastic approximate inference will invariably yield only stochastic gradients.

3.6 Example: Time-series inference with Gaussian processes

In the following, we consider the case of time-series inference where

$$f \sim \text{GP}$$

is distributed according to a Gaussian process, i.e.,

$$x_{t+1} = f(x_t) + \epsilon, \quad x_0 \sim p(x_0), \quad \epsilon \sim \mathcal{N}(0, Q), \quad f \sim \text{GP}(m, k). \tag{68}$$

We will outline approaches to deterministic and stochastic approximate inference in these systems. Figure illustrates the difference between both approaches: In sampling-based (stochastic) approaches, the GP distribution is represented by **a set of sampled trajectories (green)**, while deterministic approaches use **parametric (Gaussian) distributions** to represent marginals of the GP.

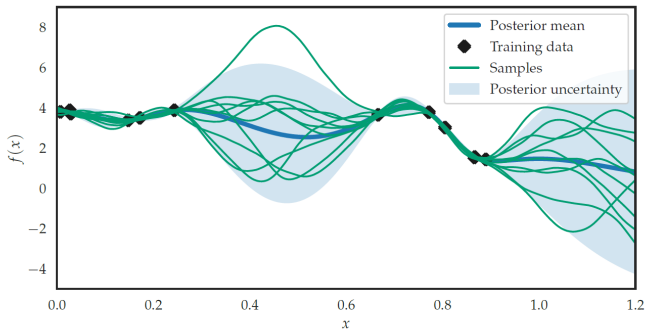


Figure 3.17: Samples from a GP posterior (green). The marginal Gaussian posteriors are indicated by the mean (black) and the shaded region representing the corresponding uncertainty.

Deterministic approximate inference Here, we aim to extract marginal distributions

$$p(x_{t+1}) = \int p(x_{t+1} \mid x_t)\, p(x_t)\, dx_t \tag{69}$$

$$= \int p(x_{t+1} \mid f, x_t)\, p(f)\, df\, p(x_t)\, dx_t \tag{70}$$

by means of deterministic approximations, i.e., **without the use of randomness**. Similarly to non-GP systems, deterministic inference can be performed by means of linearization, unscented transformation or moment matching . In all cases, the uncertainty of the GP must be taken into account when predicting the mean and covariance matrix of x_{t+1}.

For example, to do moment matching with GPs, we aim to compute the mean and covariance of x_{t+1}. If we assume $x_t \sim p$ and $f \sim \mathrm{GP}(m, k)$, then

$$\mu_{t+1} := \mathbb{E}_{f\sim\mathrm{GP},\, x_t\sim p}[f(x_t)] = \mathbb{E}_{x_t\sim p}\big[\mathbb{E}_{f\sim\mathrm{GP}}[f(x_t)]\big] = \mathbb{E}_{x_t\sim p}[m(x_t)], \tag{71}$$

where we exploited the law of iterated expectations and where we use m for the (posterior) mean function of the GP. Computing this expectation will require computing kernel expectations as discussed in (27) in Section 2.3.

The predictive variance is obtained as

$$\Sigma_{t+1} = V_{f\sim\mathrm{GP},\, x_t\sim p}[f(x_t)] \tag{72}$$

$$= V_{x_t\sim p}\big[\mathbb{E}_{f\sim\mathrm{GP}}[f(x_t)]\big] \;+\; \mathbb{E}_{x_t\sim p}\big[V_{f\sim\mathrm{GP}}[f(x_t) \mid x_t]\big], \tag{73}$$

where we used the law of total variance. Again, we will need to compute kernel expectations to compute the predictive variance. Figure below llustrates moment matching with Gaussian processes. More details on moment matching with GPs in dynamical systems can be found in [**?**, **?**, Deisenroth et al., 2012, 2015].

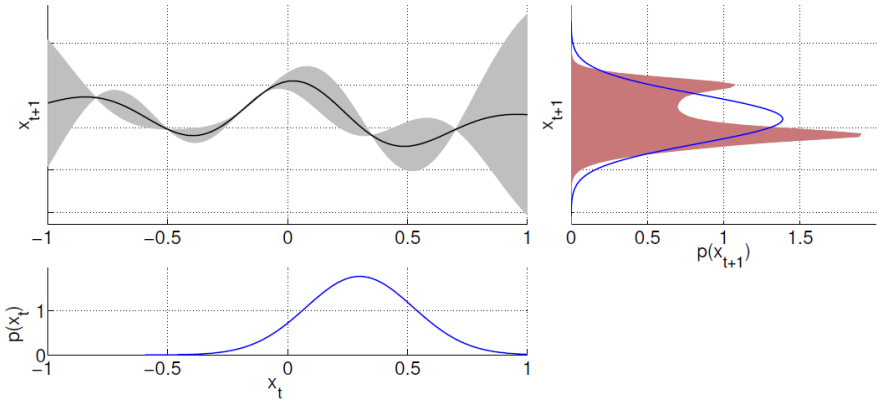


Figure 3.18: Moment matching with Gaussian processes. An input distribution $p(x_t)$ is pushed through the Gaussian process. The exact predictive distribution (red) cannot be computed analytically, but the mean and the variance can be determined via kernel expectations. Then, the true predictive distribution can be approximated by a Gaussian that has the correct mean and variance.

Example: Model-based reinforcement learning

- Learn dynamics of a physical system from data Gaussian process
- Given the learned system, find policy parameters θ^* that minimize an expected long-term cost

$$\mathbb{E}_\tau[U(\tau)] = \sum_{t=1}^{T} \mathbb{E}_{x_t}[c(x_t)]$$

- GP moment matching for long-term predictions to get an estimate of the expected long-term cost
- Gradient descent to find θ^*

$$x_{t+1} = f(x_t, u_t) + \epsilon, \quad f \sim GP$$

$$u_t = \pi(x_t; \theta)$$

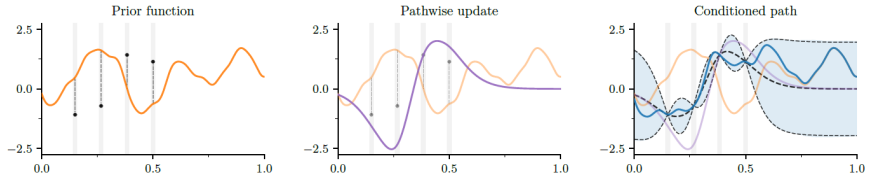
Stochastic approximate inference Instead of iteratively computing marginal distributions of future states, we can also follow the **pathwise perspective and sample trajectories from (posterior) Gaussian processes**. To ensure we get consistent trajectories from a GP posterior, we need to account for samples $x_1^{(s)}, \ldots, x_t^{(s)}$ when sampling $x_{t+1}^{(s)}$. This can be done by augmenting the training dataset with these samples[¶]. The issue with this approach to trajectory sampling is that it scales cubically in the length T of the trajectory, when implemented naively, as it would require sampling from a T-dimensional multivariate Gaussian distribution.

Decoupled Sampling

Wilson et al. propose a more efficient way to sample trajectories from GP posteriors. The key insight is to exploit Matheron's rule, which allows us to **write a posterior sample as a sum of a sample from the prior and a data-dependent update term**:

$$f^{(s)}(\cdot) + k_x(\cdot, X)\,(K + \sigma^2 I)^{-1}\big(y - f^{(s)}(X)\big) = f^{(s)}(\cdot) \mid X, y, \tag{74}$$

where the data-dependent update term depends on error/residual between the prior sample and the training data y. The update can be thought of as a mapping from prior to posterior. Figure below illustrates the decomposition of the GP posterior into a sum of the prior and the data-dependent update term.



(a) Sample from the prior (orange) ignores the data (dots).

(b) Data-dependent update (purple) of the prior sample.

(c) Sample from the posterior (blue) is obtained as the sum of the prior sample and the data-dependent update.

Figure 3.19: Efficiently sampling from posterior GPs. Matheron's rule allows us to express a sample from the posterior GP as the sum of a sample from the GP prior and a (deterministic) data-dependent update term.

Sample functions from a Gaussian process by exploiting **Matheron's rule** (for Gaussian random variables):

$$\text{posterior} = \text{prior} + \text{data-dependent update}$$

[¶]We will need to delete these samples from the training set once the full trajectory has been generated.

- Think about the posterior in terms of samples, not in terms of (conditional) distributions.
- Samples from the posterior can be obtained through a two-step procedure:
  1. Sample from prior: Source of randomness
  2. "Correct" sample using a data-dependent update term: Deterministic transformation

Properties

$$\underbrace{f^{(s)}(\cdot)}_{\text{sample from prior}} + \underbrace{k(\cdot, \mathbf{X})\mathbf{K}^{-1}(y - f^{(s)}(\mathbf{X}))}_{\text{data-dependent update}} = \underbrace{f^{(s)}(\cdot) \mid \mathbf{X}, y}_{\text{sample from posterior}}$$

- **Update term** depends on error/residual between the prior sample and data y.
- **Update term** is a mapping from prior to posterior.
- Different representations for prior and update terms (e.g., RFF for prior and finite basis-function representation for update).
  - Sampling from RFF prior scales **linearly in the number T of test inputs.**
  - Update term can be computed **linearly in the number T of test inputs.**
- **Functions can be sampled efficiently** (linearly in the number of test inputs).

To increase computational efficiency, Wilson et al. use different representations for the GP prior and the data-dependent update term. By separately representing the prior using Fourier basis functions (assuming a stationary kernel) and the update term using canonical basis functions $k(\cdot, x_j)$, an efficient **approximation of the posterior GP** is obtained.[||]

The data-dependent update term depends on the error/residual between the prior sample and the training data y. The update can be thought of as a mapping from prior to posterior. Different representations for prior and update terms can be used, e.g., random Fourier features (RFF) for the prior and finite basis-function representation for update. Then,

- Sampling from RFF prior scales linearly in the number T of test inputs.
- The update term can be computed linearly in the number T of test inputs.

Overall, functions can be sampled efficiently, i.e., linearly in the number of test/query inputs, which is a significant speedup from the **original cubic scaling with a naive implementation**.

[||] The Fourier basis is well-suited for representing the prior [**?**, Rahimi and Recht, 2008], and the canonical basis is well-suited for representing the data [**?**, Burt et al., 2019].

Application

Applications of this efficient sampling strategy include deep convolutional GP auto-encoders, Bayesian optimization with Thompson sampling, sampling from GPs on manifolds, and model-based reinforcement learning. It is also integrated into BoTorch, a software toolbox for Bayesian optimization.

3.6.1 Summary

- Propagate uncertainty through a nonlinear dynamical system
- Deterministic approximate inference (linearization, unscented transformation)
- Stochastic approximate inference (sampling)
- Examples in the context of GP dynamical systems

Chapter 4

Message Passing

Message passing algorithms play a crucial role in probabilistic inference on graphical models such as Bayesian networks and Markov random fields. These algorithms efficiently compute marginal distributions and are essential in various applications, including:

- Image processing
- Molecular biology
- Social network analysis
- Belief propagation in AI

4.1 Graphs: Definition and Properties

4.1.1 Graph Definition

A graph G is defined as:

$$G = (V, E)$$

where V is a set of nodes (vertices) and E is a set of edges, which can be either directed or undirected. An edge can be associated with a pair of nodes (u, v).

4.1.2 Types of Graphs

Directed Graphs

A directed graph (or digraph) has edges represented as **ordered pairs**:

$$\varphi : E \to V \times V.$$

So the ordering of the nodes associated to an edge matters. For any edge $e = (u, v)$, the **direction matters**, meaning $(u, v) \neq (v, u)$.

Undirected Graphs

In an undirected graph, edges have no inherent direction:

$$e = \{u, v\}, \quad u, v \in V.$$

This implies that $(u, v) = (v, u)$.

4.1.3 Graph Representation

Adjacency Relation in Graphs

The edges E of a graph define an **adjacency relation** $\sim$ on V: For $x, y \in V$,

$$x \sim y \iff \{(x, y)\} \cup \{(y, x)\} \subset \varphi(E).$$

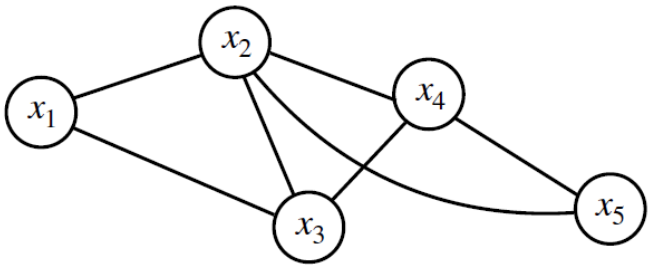


Figure 4.1

On the graph on the left, we have e.g.

- $x_1 \sim x_2$
- $x_4 \sim x_5$
- $x_1 \sim x_4$
- $x_3 \sim x_5$

If $x \sim y$, we say that y is a **neighbour** of x and vice versa.

Adjacency Matrix

The adjacency matrix $\mathbf{A}$ encodes the adjacency structure of a graph G:

$$A_{ij} = \begin{cases} 1, & \text{if } x_i \sim x_j, \\ 0, & \text{no edge.} \end{cases}$$

Degree Matrix

The degree matrix $\mathbf{D}$ encodes the degree of connectivity of each node (diagonal matrix):

$$D_{ij} = \begin{cases} |\text{Neighbours}(x_i)|, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

Laplacian Matrix

The Laplacian matrix is given by:

$$L = D - A.$$

4.1.4 Types of Graphs

Fully-connected Graphs

A fully-connected graph (or complete graph) is an **undirected graph** where **each node is connected to every other node**. Mathematically, this means that for a graph with n nodes, the total number of edges is:

$$E = \frac{n(n-1)}{2}.$$

These graphs are used in applications where every entity interacts with every other entity, such as in **neural networks and social networks**.

Directed Acyclic Graphs (DAG)

A Directed Acyclic Graph (DAG) is a directed graph that contains **no cycles**. This means there is **no path where a node can return to itself** following the directed edges.

DAGs are widely used in:

- Bayesian Networks: To model probabilistic dependencies.
- Task Scheduling: Dependencies in computational workflows.
- Causal Inference: Representation of cause-and-effect relationships.

Mathematically, if $G = (V, E)$ is a DAG, then there exists a topological ordering of the vertices such that for every directed edge $(u, v) \in E$, node u appears before v in the ordering.

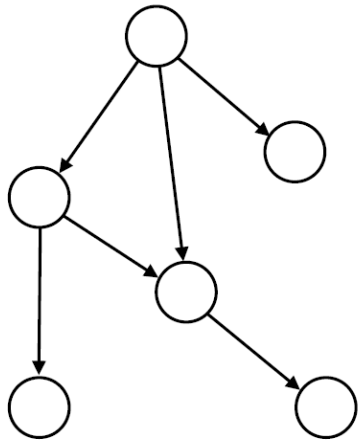


Figure 4.2: A Directed Acyclic Graph (DAG) with no directed cycles.

Trees and Polytrees

A tree is an **undirected graph** in which **any two nodes are connected by exactly one unique path**. No loop at all. It has the following properties:

- A tree with n nodes has exactly $n - 1$ edges.
- Removing any edge disconnects the graph.
- Trees are connected and acyclic.

A polytree is a **directed acyclic graph (DAG)** where the underlying undirected graph forms a tree. This means that every node (except the root) has exactly one parent.

Trees and polytrees are useful in:

- Hierarchical structures (e.g., organizational charts).
- Search algorithms (e.g., binary search trees).
- Probabilistic modeling (e.g., decision trees, Bayesian networks).

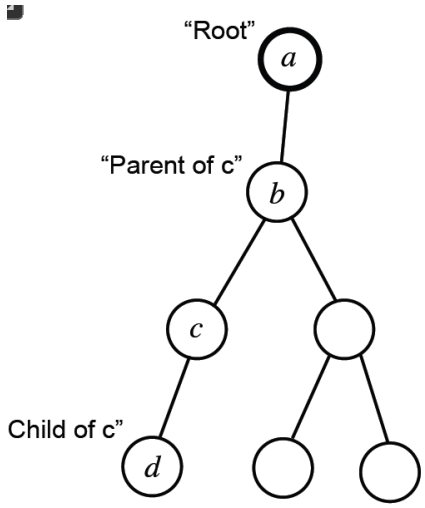


Figure 4.3: A tree is an undirected graph where nodes are connected by unique paths.

Bipartite Graphs

A bipartite graph is a graph whose nodes can be **partitioned into two sets** (A, B) such that each edge connects a node in A to a node in B. This means there are **no edges between nodes within the same set**. Can be either directed or undirected.

Mathematically, a graph $G = (V, E)$ is bipartite if and only if it does not contain an odd-length cycle.

Bipartite graphs are useful in:

- Matching problems (e.g., job assignment).
- Recommender systems (e.g., users and movies).
- Network flow algorithms (e.g., maximum flow in transport networks).

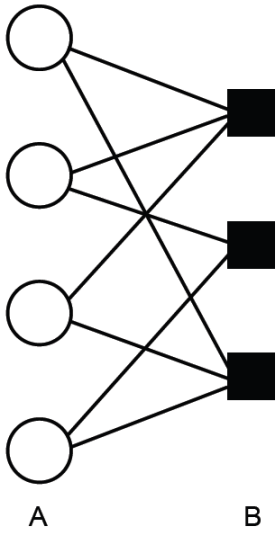


Figure 4.4: A bipartite graph where edges connect nodes from two distinct sets.

Subgraphs and Cliques

A subgraph $G_1 = (V_1, E_1)$ of a graph $G = (V, E)$ is a graph where $V_1 \subset V$ and $E_1 \subset E$.

A **clique** is a subgraph where every pair of nodes is connected. The largest clique in a graph is called the **maximum clique**, and finding it is an NP-hard problem.

Applications of subgraphs and cliques include:

- Community detection in social networks.
- Biological networks (e.g., protein interaction networks).
- Graph-based clustering (e.g., in machine learning).

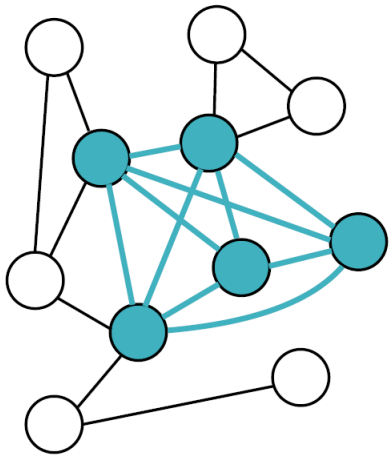


Figure 4.5: A subgraph G_1 of a larger graph G, with highlighted nodes and edges.

4.2 Probabilistic Graphical Models (PGMs)

Probabilistic Graphical Models (PGMs) are powerful tools for representing and reasoning about uncertainty in complex systems. These models combine probability theory and graph theory, where:

- Nodes represent random variables.
- Edges represent probabilistic dependencies between variables.

PGMs provide a structured way to model joint distributions and factorize them efficiently using **Bayesian Networks (BNs) or Markov Random Fields (MRFs)**.

4.2.1 Example: Joint Probability Factorization

Consider a set of variables $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$, where the joint probability distribution is factorized as:

$$p(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3)$$

$$\times p(x_5|x_1, x_3)p(x_6|x_4)p(x_7|x_4, x_5).$$

This factorization follows from the conditional independence assumptions encoded by the graphical structure.

Key questions:

- If x_4 is observed, are x_2 and x_6 independent?

$$p(x_2, x_6|x_4) \stackrel{?}{=} p(x_2|x_4)p(x_6|x_4).$$

- Which variable should we observe for x_6 and x_7 to be independent?

$$p(x_6, x_7|?) = p(x_6|?)p(x_7|?).$$

4.2.2 Bayesian Networks

A **Bayesian Network (BN)** is a directed acyclic graph (DAG) that represents how a **joint probability distribution** factorizes into **conditional probability distributions**.

Example: A Bayesian network for four variables:

$$p(x_1, x_2, x_3, x_4) = p(x_4|x_3)p(x_3|x_1, x_2)p(x_2|x_1)p(x_1).$$

Properties:

- Each node represents a random variable.
- Each edge represents a **causal relationship** (causality) or **probabilistic dependency**. e.g. $p(x_4|x_3)$ corresponds to the arrow from x_3 to x_4.
- The network follows a directed acyclic structure, ensuring **no feedback loops**.

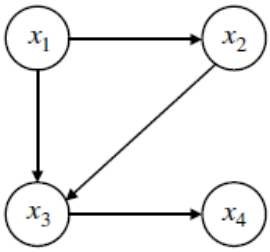


Figure 4.6: A Bayesian Network (BN) with four nodes, showing how the joint probability distribution factorizes into conditional probabilities.

4.2.3 Independence in Bayesian Networks

Two nodes in a Bayesian network are **independent** if there are no paths connecting them.

Example: If variable x_4 has no incoming or outgoing edges, then:

$$p(x_1, x_2, x_3, x_4) = p(x_4)p(x_1, x_2, x_3),$$

meaning x_4 is **independent** of all other nodes.

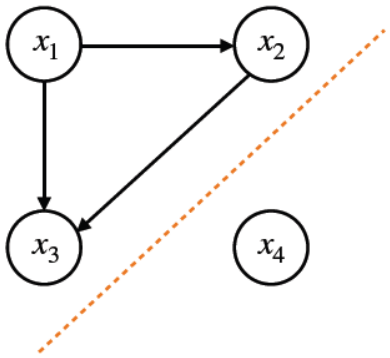


Figure 4.7: Node x_4 is independent of all other variables since no paths connect it.

4.2.4 d-Separation and Conditional Independence

Definition: Two nodes a and b in a DAG are **d-separated** by a set of nodes Z if all loop-free paths from a to b are blocked by Z.

A path can be blocked in three ways:

1. **Chain:** $a \to c \to b$ and $c \in Z$.
2. **Fork:** $a \leftarrow c \to b$ and $c \in Z$.
3. **Collider:** $a \to c \leftarrow b$, and $c \notin Z$ nor any descendant of c is in Z. Path contains a collider.

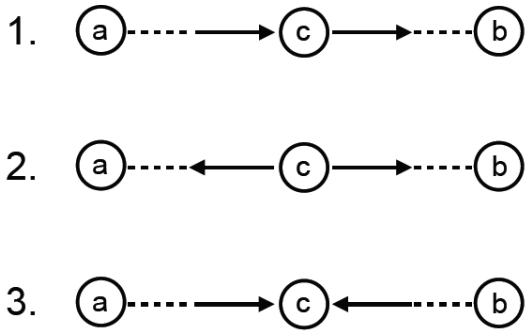


Figure 4.8: Illustration of d-separation: a and b are d-separated if c blocks all paths between them.

Property: Variable a and b are independent given Z = they are separated by Z.

4.2.5 Example of d-Separation

Consider three nodes a, b, c. If c is part of a **chain** between a and b, then conditioning on c makes a and b independent.

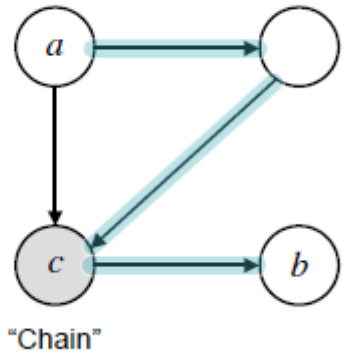


Figure 4.9: Example of d-separation: a and b are independent when conditioning on c.

4.2.6 Non-example of d-Separation

If c is a collider between a and b, then observing c introduces a dependency between a and b.

i.e. a and b are **d-connected** because the graph contains a **collider** and c is a **descendant** of the collider node.

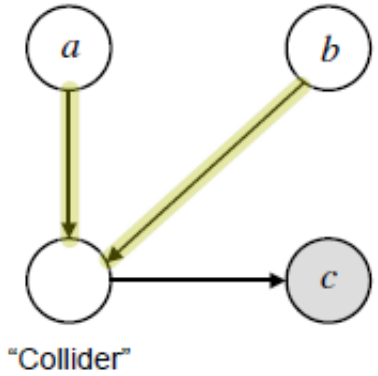


Figure 4.10: A collider c does not d-separate a and b, making them dependent.

4.2.7 Markov Random Fields (MRFs)

Unlike Bayesian networks, Markov Random Fields (MRFs) use **undirected graphs** to represent probabilistic dependencies.

Key property:

- A and B are conditionally independent given C **if and only if paths between points in A and B are blocked by any points in C.**
- Thus, two nodes a and b are **non-adjacent** if and only if they are **conditionally independent given all other nodes**.

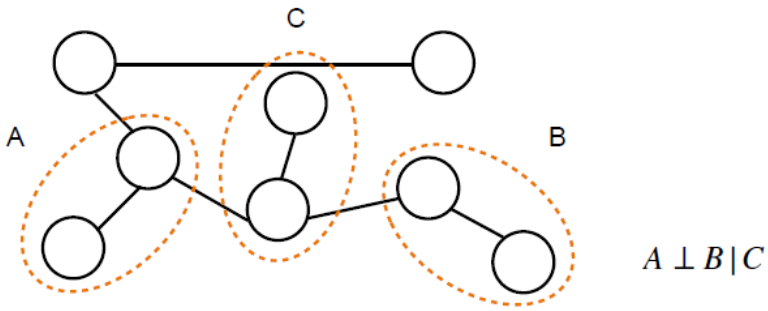


Figure 4.11: A and B are conditionally independent given C.

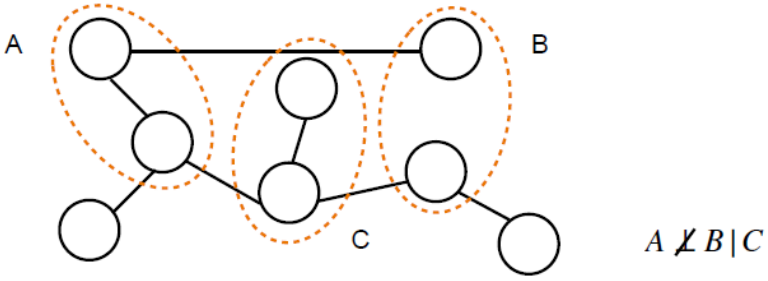


Figure 4.12: As there exists a path between A and B that doesn't include any element of C. So A and B are **conditionally dependent**.

4.2.8 Hammersley-Clifford Theorem

In Markov Random Fields (MRFs), we can consider factorizations into **potential functions** $\psi_C(x_C) \geq 0$:

$$p(x_1, \ldots, x_n) \propto \prod_C \psi_C(x_C),$$

where x_C is a subset of variables from the graph, and C is a **clique** of the graph*.

This is akin to factorizing **joint distributions** into **conditional distributions** in Bayesian Networks (BNs).

Key Properties:

- Potential functions **need not** have a probabilistic interpretation.
- Factorization is **not unique**.

Example: Factorization into Maximal Cliques

The probability distribution can be written as a product over the maximal cliques of the graph:

$$p(x_1, x_2, x_3, x_4) = \psi_{123}(x_1, x_2, x_3)\psi_{34}(x_3, x_4).$$

where ψ_{123} and ψ_{34} are potential functions over their respective maximal cliques.

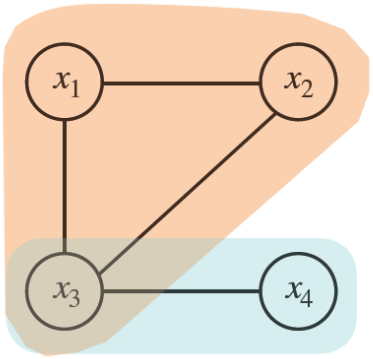


Figure 4.13: Factorization into maximal cliques. The graph is divided into two maximal cliques: $\{x_1, x_2, x_3\}$ and $\{x_3, x_4\}$, each assigned a clique potential.

Example: Factorization into Pairwise Cliques

Alternatively, the probability distribution can be factorized into pairwise potentials:

$$p(x_1, x_2, x_3, x_4) = \psi_{12}(x_1, x_2)\psi_{13}(x_1, x_3)\psi_{23}(x_2, x_3)\psi_{34}(x_3, x_4).$$

*A clique is a fully-connected subgraph of a graph.

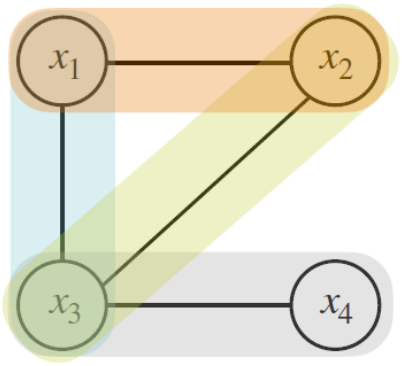


Figure 4.14: Factorization into pairwise cliques. Instead of maximal cliques, the graph is now decomposed into smaller pairwise interactions between nodes.

Factorization of Bayesian Networks

A Bayesian network factorizes a joint distribution into conditional probabilities:

$$p(x_1, x_2, x_3, x_4) = p(x_4|x_3)p(x_3|x_1, x_2)p(x_2|x_1)p(x_1).$$

This can be rewritten using clique potentials:

$$p(x_1, x_2, x_3, x_4) = \psi_{34}(x_3, x_4)\psi_{123}(x_1, x_2, x_3)\psi_{12}(x_1, x_2)\psi_1(x_1).$$

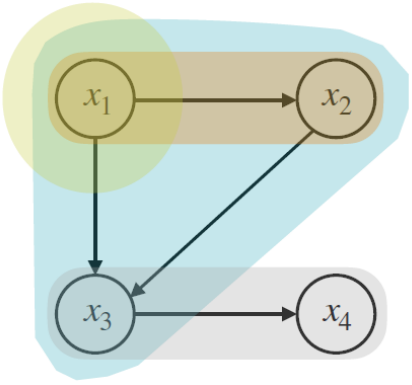


Figure 4.15: Factorization of a Bayesian network. Here, directed dependencies determine the factorization into conditional probabilities rather than undirected potentials.

4.2.9 Markov Random Fields vs Bayesian Networks

Differences Between MRFs and BNs

Markov Random Fields (MRFs) are **not equivalent to** Bayesian Networks (BNs) without modifying the graph structure.

- BNs are **directed acyclic graphs (DAGs)**, where edges encode causal relationships.
- MRFs are **undirected graphs**, representing symmetric dependencies.

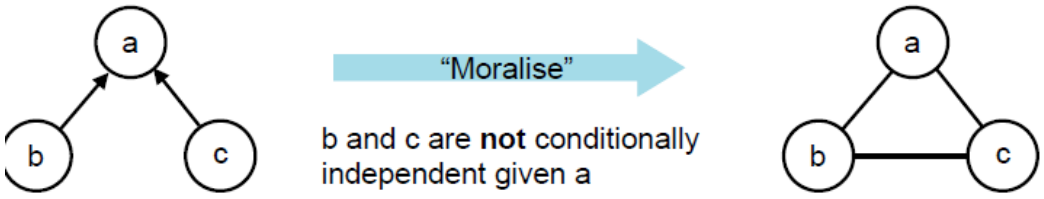


Figure 4.16: MRFs and Bayesian networks are not equivalent. **The process of "moralization", which is adding link between the b and c here, is required to convert a BN into an MRF.**

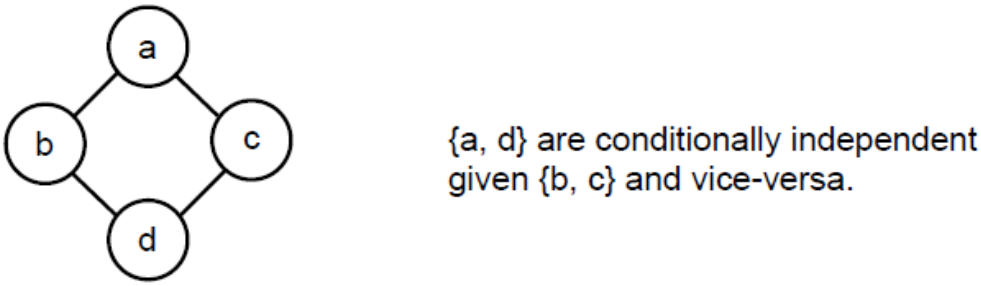


Figure 4.17: **Not all MRFs can be represented as a BN.** This figure can't happen in DAG , so this MRF couldn't be written as a BN.

4.2.10 Factor Graphs

Definition

Factor graphs are an alternative way to represent both Bayesian networks and MRFs by explicitly showing the factorization of the probability distribution.

Key Properties:

- Circle nodes (◯) represent random variables.
- Square nodes (□) represent **factors (individual components in the factorization)**. It's connected to the RVs related.
- The graph is **undirected and bipartite**.

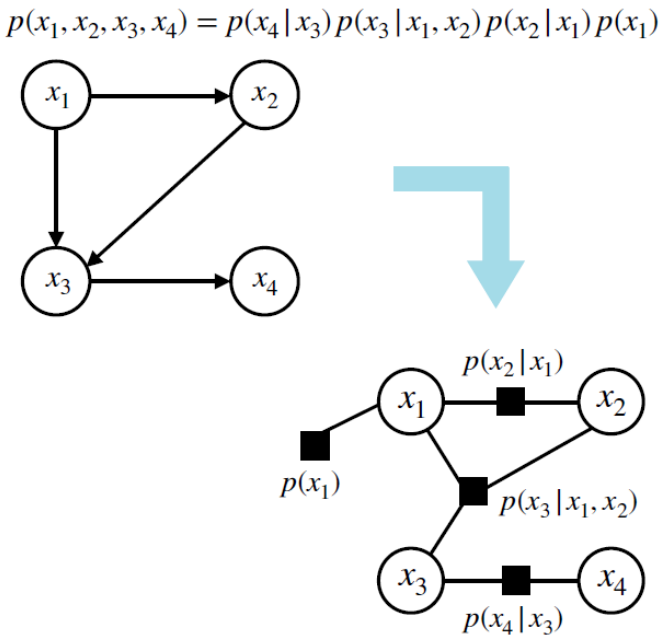


Figure 4.18: Factor graphs explicitly represent how a probability distribution factorizes. Variables are connected via factor nodes, making dependencies clear.

Different Factorization Methods

Factor graphs make the factorisation explicit, so very useful for MRFs where factorization is non-unique. Multiple ways exist to define potentials.

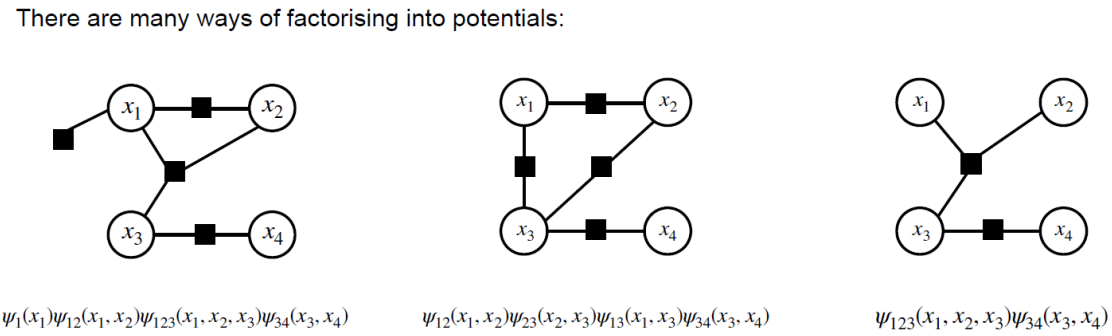


Figure 4.19: **Different factorization methods in factor graphs.** The same underlying probability distribution can be expressed in different ways.

4.2.11 Useful Notations

Plate Notation

Plate notation is a compact way to represent repeated structures in graphical models. It simplifies models where variables are replicated multiple times.

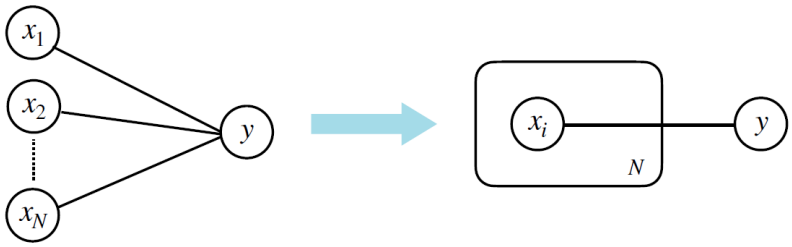


Figure 4.20: Plate notation represents repeated variables concisely. Instead of drawing multiple identical nodes, **one node with a repetition indicator (N)** is used.

Shaded vs Unshaded Nodes

- Shaded nodes indicate **observed variables**.
- Unshaded nodes represent **latent (hidden) variables**.

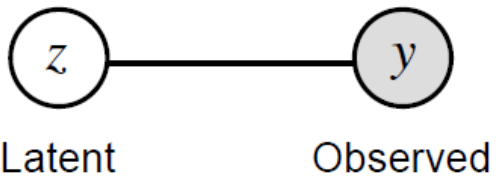


Figure 4.21: Shaded vs. unshaded nodes. Observed variables are filled, while latent variables remain unshaded.

4.2.12 Examples of Bayesian Networks

Naïve Bayes Classifier

A simple directed graphical model used for classification:

$$p(x_1, x_2, \ldots, x_N|c) = p(c) \prod_{i=1}^{N} p(x_i|c).$$

Hidden Markov Model

A sequential probabilistic model, where states follow a Markov process:

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(z_1)p(z_2|z_1)p(x_2|z_2) \ldots .$$

Bayesian Linear Regression

Bayesian linear regression introduces prior distributions over model parameters:

$$y_i = f_i + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma),$$

$$f_i = wx_i + b.$$

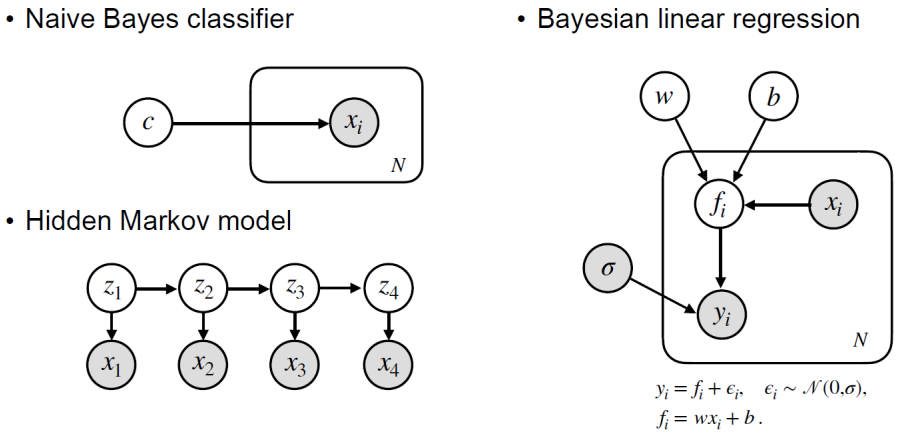


Figure 4.22: Examples of Bayesian Networks: (Left) Naïve Bayes Classifier, (Middle) Hidden Markov Model, (Right) Bayesian Linear Regression. These models illustrate different ways of structuring probabilistic dependencies.

4.2.13 Examples of Markov Random Fields

Markov Random Fields (MRFs) are widely used in various domains where spatial and contextual dependencies play a critical role. Below are two key applications:

Spatial Analysis and Image Processing

MRFs are commonly used in image segmentation, denoising, and object recognition. The idea is to model pixels as nodes in a grid-structured graph, where the relationship between neighboring pixels determines how smooth or discontinuous a region is.

- Each node represents a pixel or region in an image.
- Edges enforce spatial continuity by encouraging neighboring pixels to have similar values.
- The MRF model helps in tasks such as denoising, where noise can be reduced by enforcing similarity constraints.

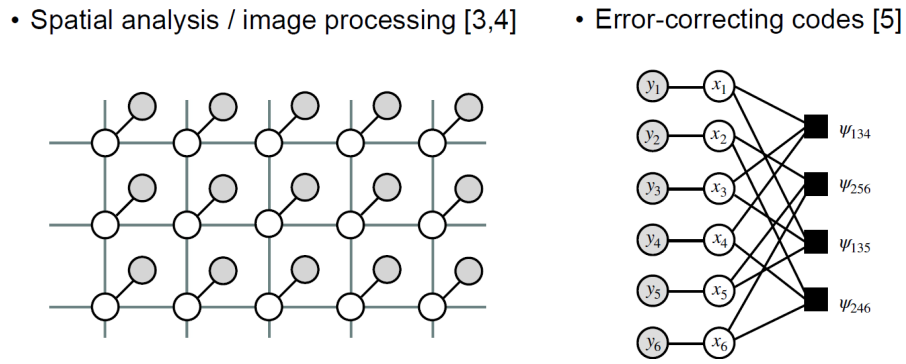


Figure 4.23: **(1)Markov Random Fields in spatial analysis and image processing.** The nodes represent pixels in an image, and edges enforce smoothness constraints by capturing local dependencies. **(2)Markov Random Fields in error-correcting codes.** Circles represent observed codewords (y_i), while squares represent constraints (ψ factors) used in decoding algorithms such as belief propagation.

Error-Correcting Codes

MRFs are also useful in error-correcting codes, such as those used in communication systems and digital storage. The graphical representation helps in efficient decoding using probabilistic methods.

- Nodes represent codewords (y_i) and parity-check constraints (x_i).
- Factors $\psi_{134}, \psi_{256}, \psi_{135}, \psi_{246}$ enforce relationships between the variables.
- Algorithms such as belief propagation operate on this graph to correct errors by estimating the most probable transmitted message.

4.3 Belief Propagation on PGMs

Belief propagation (BP) is an algorithm used for efficient inference in graphical models, particularly for computing marginal probabilities in tree-structured graphs.

4.3.1 Statistical Inference with PGMs

In Bayesian statistics, we often need to compute:

1. The marginal likelihood $p(y)$ of observed data.

2. The marginal distribution $p(z)$ of latent variables.
3. The conditional distribution $p(x_i|x_j)$ for any $i, j \in V$.
4. The mode of the distribution: $x^* = \arg\max_x p(x)$.

Here the focus is **marginal distributions.** Since computing exact marginal probabilities can be computationally expensive, **belief propagation/PGMs** provides an efficient alternative.

4.3.2 Example: Computing Marginal Distributions

Consider a set of random variables $X_1, \ldots, X_N$, each taking values in a discrete space of size K. The joint probability mass function is given by $p(x_1, \ldots, x_N)$.

The marginal distribution for a given variable X_i is:

$$p(X_i = x_i) = \sum_{j \neq i} \sum_{x_j \in \{1,\ldots,K\}} p(x_1, \ldots, x_N).$$

However, this naive approach has a **computational cost** of:

$$\mathcal{O}(K^N),$$

which is infeasible for large number of variables N!

Assuming Independence

Let $N = 3$. If we assume independence of x_3, the joint probability distribution factorizes:

$$p(x_1, x_2, x_3) = p(x_1, x_2)p(x_3).$$

Then, the marginal probability simplifies to:

$$\begin{aligned} p(x_1) &= \sum_{x_2} \sum_{x_3} p(x_1, x_2, x_3), \\ &= \sum_{x_2} \sum_{x_3} p(x_1, x_2)p(x_3), \\ &= \sum_{x_2} p(x_1, x_2) \sum_{x_3} p(x_3). \end{aligned}$$

Since $\sum_{x_3} p(x_3) = 1$, we obtain:

$$p(x_1) = \sum_{x_2} p(x_1, x_2).$$

A single sum is computationally more efficient than a double sum.

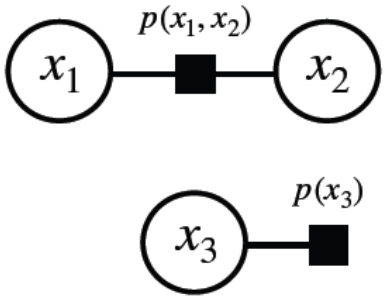


Figure 4.24

Assuming Conditional Independence

If we assume the distribution factorizes conditionally as:

$$p(x_1, x_2, x_3) = p(x_1|x_3)p(x_2|x_3)p(x_3),$$

then the marginal probability becomes:

$$p(x_1) = \sum_{x_2}\sum_{x_3} p(x_1, x_2, x_3),$$

$$= \sum_{x_2}\sum_{x_3} p(x_1|x_3)p(x_2|x_3)p(x_3),$$

$$= \sum_{x_3} p(x_1|x_3)p(x_3)\sum_{x_2} p(x_2|x_3).$$

Since $\sum_{x_2} p(x_2|x_3) = 1$, this simplifies to:

$$p(x_1) = \sum_{x_3} p(x_1|x_3)p(x_3).$$

Key observation: Using **independence or conditional independence** (i.e. sparsity of graph) significantly reduces computational complexity.

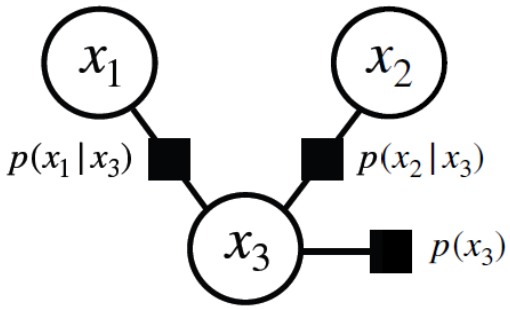


Figure 4.25

4.3.3 Belief Propagation Algorithm

Belief propagation is an algorithm for computing marginal probabilities efficiently on tree-structured graphs.

Key principles:

- Operates on factor graphs.
- Uses a message-passing approach.
- Reduces computation from exponential to polynomial time.

The probability of a configuration x is given by this factorization:

$$p(x) = \prod_{i \in V} \psi_i(x_i) \prod_{(i,j) \in E} \psi_{ij}(x_i, x_j).$$

where the first term (blue factor) denotes **potential function defined over single nodes**, the second term (red factor) denoting **pairs of nodes**.

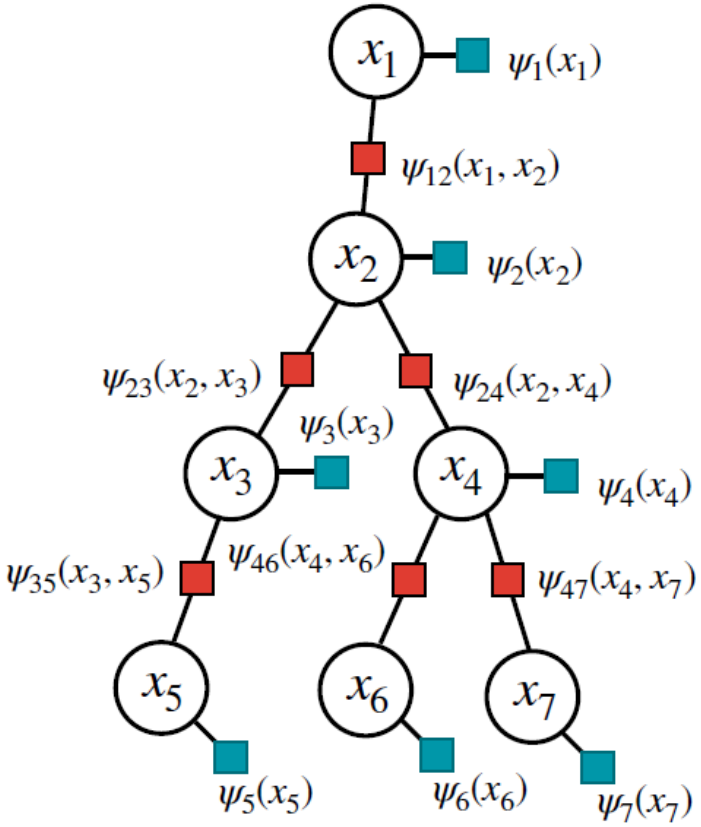


Figure 4.26: Factor graph representation of belief propagation. Messages are passed between variables and factor nodes to compute marginals efficiently.

4.3.4 Message Passing Updates

Belief propagation proceeds iteratively by updating:

1. The **messages** between two nodes:

$$M_{j\to i}(x_i) = \sum_{x_j} \psi_{ij}(x_i, x_j)\psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j).$$

   i.e. the message passed from j to i depends both on potential function and the other messages (causing the iterative schemes).

2. The **state** of each node:

$$p(x_i) = \psi_i(x_i) \prod_{j\sim i} M_{j\to i}(x_i).$$

i.e. the state is computed by aggregating all the incoming messages M from its neighbouring nodes.

Step 1: Message Update

To compute $p(x_2)$, we first update messages to x_2 from x_3, x_4, and x_1.

Computing the message from x_1 to x_2:

$$M_{1\to 2}(x_2) = \sum_{x_1} \psi_{12}(x_1, x_2)\psi_1(x_1) \prod_{k\neq 2} M_{k\to 1}(x_1).$$

Rule: ere x_1 has no neighbours other than x_2 so we can ignore the product term. Ignore incoming messages to a node i if there are none.

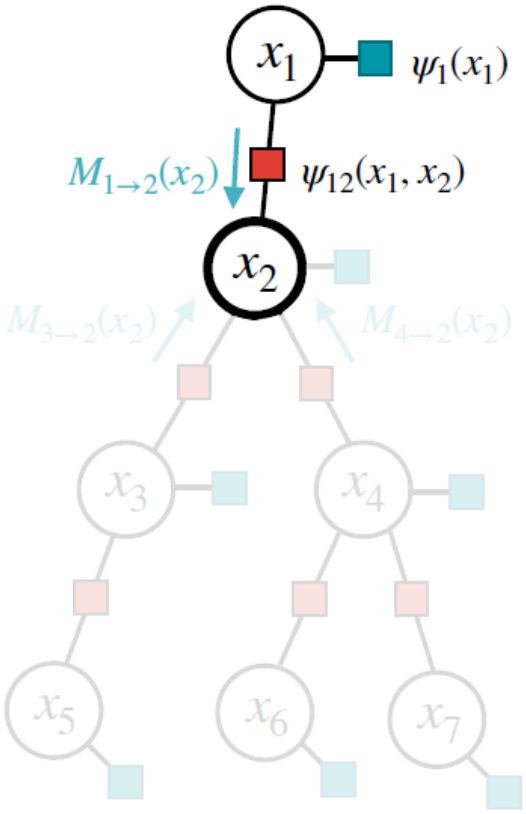


Figure 4.27: Step 1.1: Computing the message from x_1 to x_2. Messages aggregate information from neighboring nodes.

Computing Subsequent Messages

Next, we compute message from x_3 to x_2:

$$M_{3\to 2}(x_2) = \sum_{x_3} \psi_{23}(x_2, x_3)\psi_3(x_3)M_{5\to 3}(x_3).$$

To calculate the second term here, we then compute x_5 to x_3 as x_5 is the only neighbour of x_3, and take this below back to the above:

$$M_{5\to 3}(x_3) = \sum_{x_5} \psi_{35}(x_3, x_5)\psi_5(x_5) \prod_{k\neq 3} M_{k\to 5}(x_5).$$

We ignore the product term as there's no other node next to x_5 other than x_3.

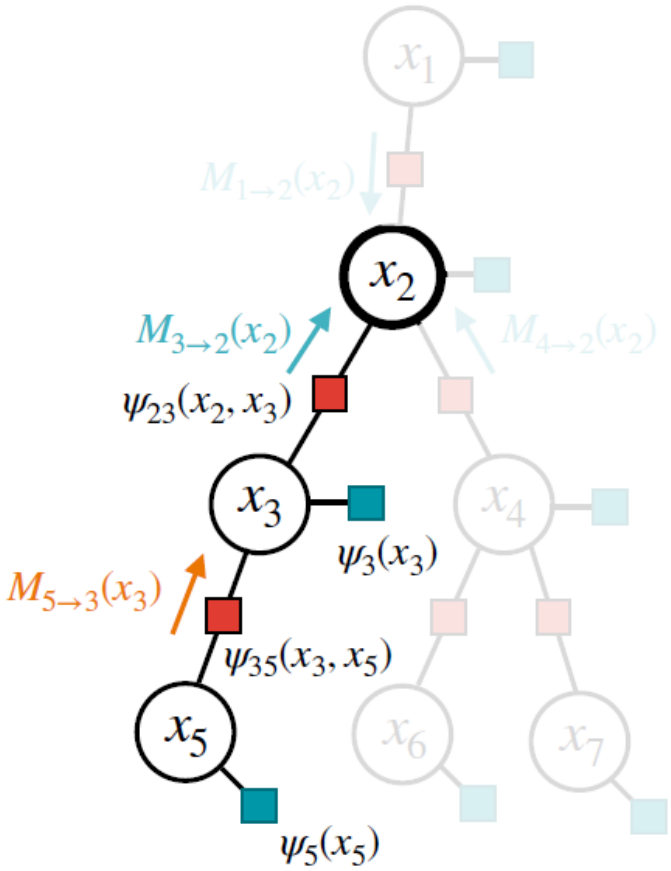


Figure 4.28: Step 1.2: Computing messages from x_3 and x_5 to x_2.

Final Message Updates

Finally, we compute the message from x_4 to x_2:

$$M_{4\to 2}(x_2) = \sum_{x_4} \psi_{24}(x_2, x_4)\psi_4(x_4)M_{6\to 4}(x_4)M_{7\to 4}(x_4).$$

where x_4 has two neighbours: x_6 and x_7

$$M_{6\to 4}(x_4) = \sum_{x_6} \psi_{46}(x_4, x_6)\psi_6(x_6),$$

$$M_{7\to 4}(x_4) = \sum_{x_7} \psi_{47}(x_4, x_7)\psi_7(x_7).$$

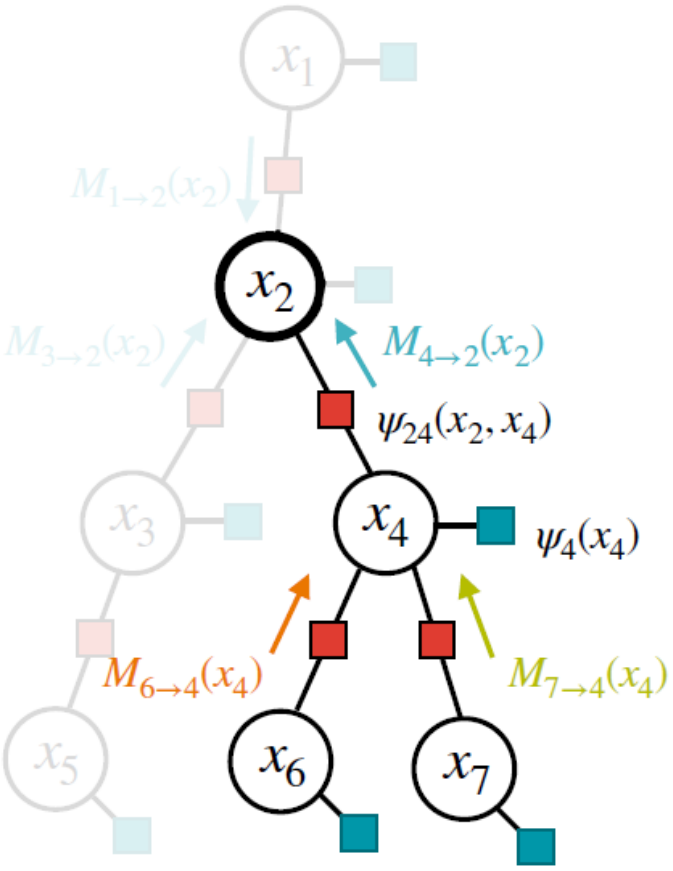


Figure 4.29: Step 1.3: Computing messages to finalize belief propagation.

4.3.5 State Updates

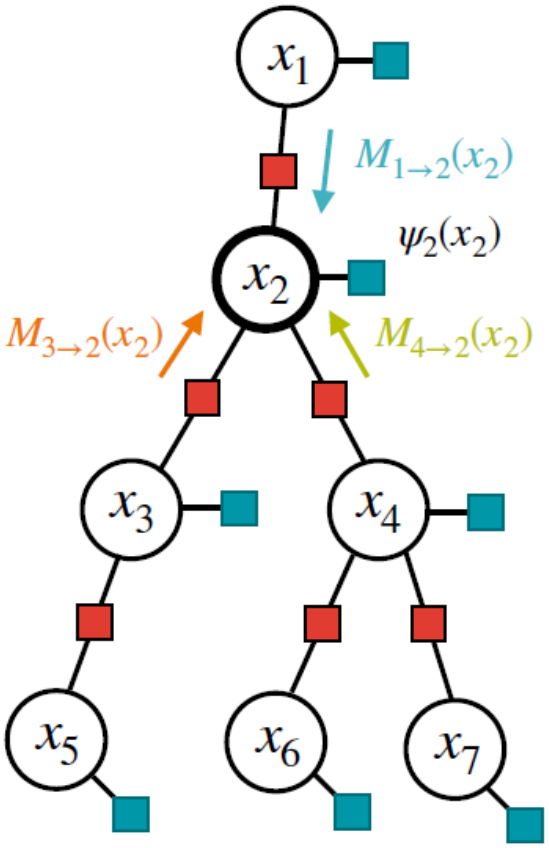


Figure 4.30: Computing $p(x_2)$ using the incoming messages. Normalization ensures that probabilities sum to one.

Once messages have been passed between nodes to x_2, we update the **state** of x_2 using:

$$p(x_i) = \psi_i(x_i) \prod_{j \to i} M_{j \to i}(x_i).$$

To compute $p(x_2)$, we normalize:

$$p(x_2) = \frac{1}{Z} \psi_2(x_2) \times M_{1\to 2}(x_2) \times M_{3\to 2}(x_2) \times M_{4\to 2}(x_2).$$

where the normalization constant is:

$$Z = \sum_{x_2} \psi_2(x_2) \times M_{1\to 2}(x_2) \times M_{3\to 2}(x_2) \times M_{4\to 2}(x_2).$$

4.3.6 Efficient Implementation of Belief Propagation

Tree-Structured Graphs

By leveraging the tree structure, we can compute all marginal distributions efficiently.

Step 0: Initialization

We initialize all the **states** in each node as uniform distribution:

$$p(x_i) = \frac{1}{K}1, \quad \forall i \in V,$$

And all the **messages** as a constant function:

$$M_{j\to i}(x_i) = 1, \quad \forall (i,j) \in E.$$

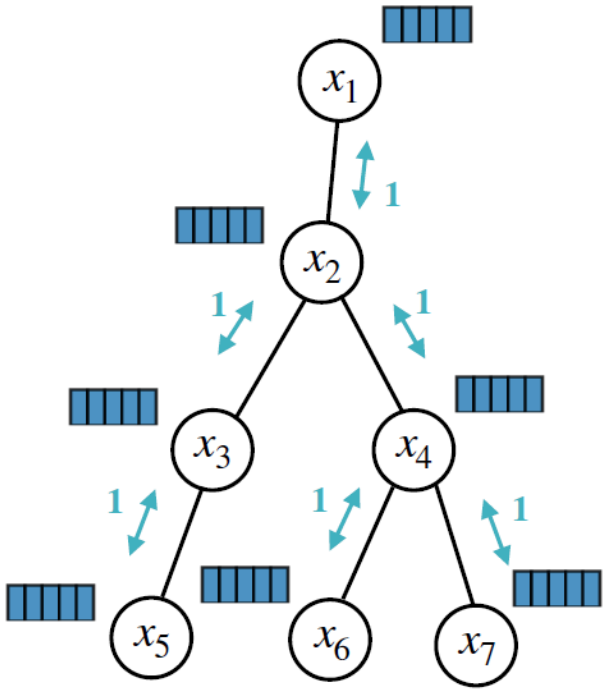


Figure 4.31: Step 0: Initializing the states and messages before running belief propagation.

Step 1: Root and Leaf Nodes

We choose a **root node**, which can be any node in the graph, and identify the corresponding **leaf nodes**.

Note: Leaves are the furthest descendants of the root.

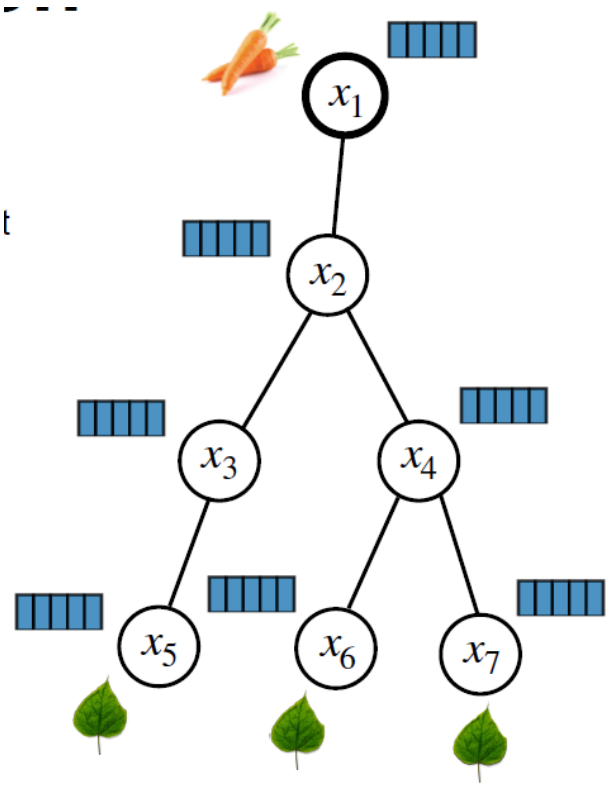


Figure 4.32: Step 1: Selecting a root node and identifying leaf nodes. The algorithm proceeds from leaves to root.

Step 2: Message Passing

Next, we update the:

- all **messages** propagate from leaf nodes to its parent node towards the root node (by using the message update formula)
- all **states** of the **parent node** propagate accordingly too (by using the state update formula)

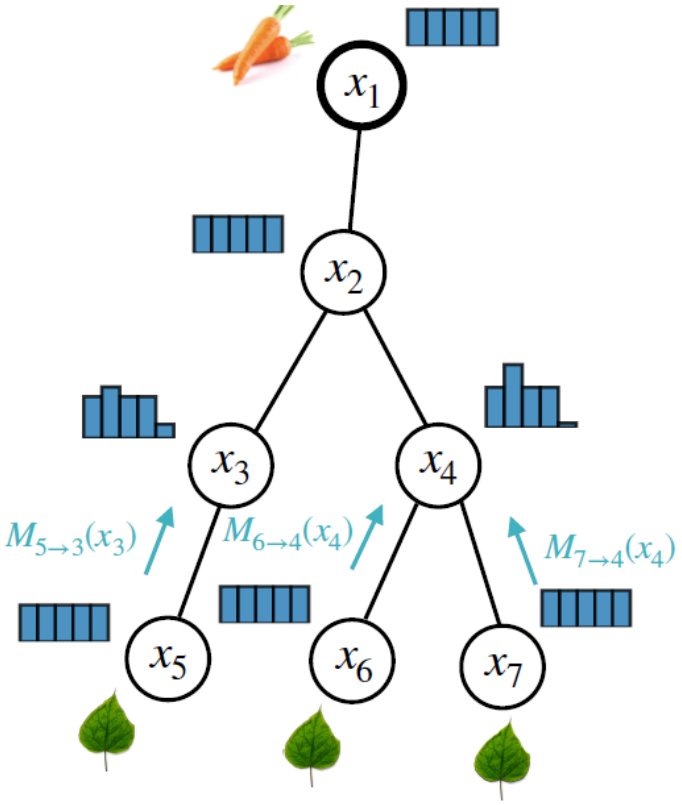


Figure 4.33: Step 2: Messages propagate upwards from leaves to their parent nodes, refining marginal estimates.

Step 3: Root-to-Leaf Updates

Once we update the **messages and states** all the way up to the root node , we update the **states and propagate messages back down to the leaves**.

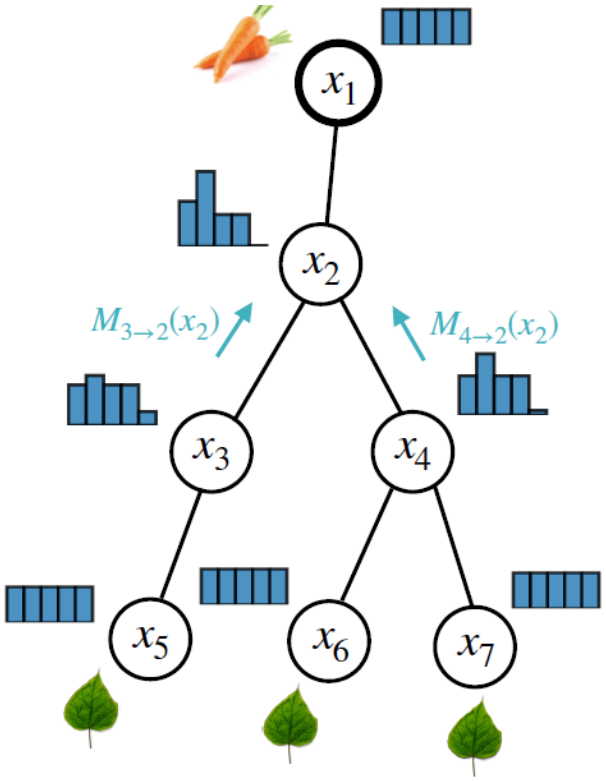


Figure 4.34: Step 3: Updating states and propagating messages from the root down to the leaves.

Step 4: Downward Sweep

Starting from the root back down to the leaves, a second pass occurs, refining marginal estimates further.

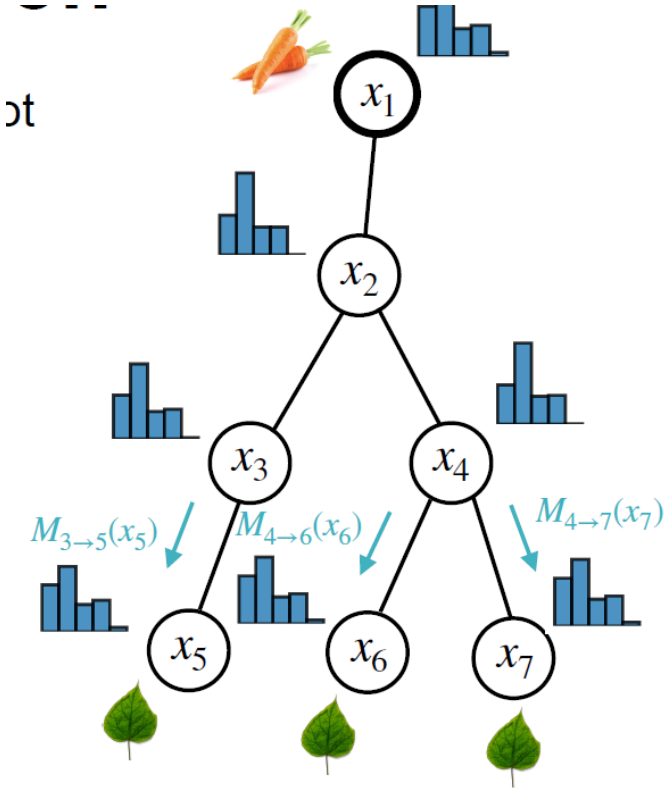


Figure 4.35: Step 4: Final downward sweep to ensure all nodes have correct marginal probabilities.

With these steps, we have successfully computed the **marginal probabilities** from x_1 to x_7.

4.3.7 Remarks on Belief Propagation

- **Guaranteed convergence** after a single sweep in tree-structured graphs.
- **Linear complexity** in N, not exponential, making it significantly more efficient than **brute-force marginalization**.
- Can be implemented in real-world applications like error correction and probabilistic inference.

4.3.8 Computing Checklist

Given a graph $G = (V, E)$, we verify:

1. Can we compute the marginal likelihood $p(y)$ of observed data? ✓

2. Can we compute the marginal distribution $p(z)$ of latent variables? ✓

3. Can we compute the conditional distribution $p(x_i|x_j)$ for any i, j in V? ??✓

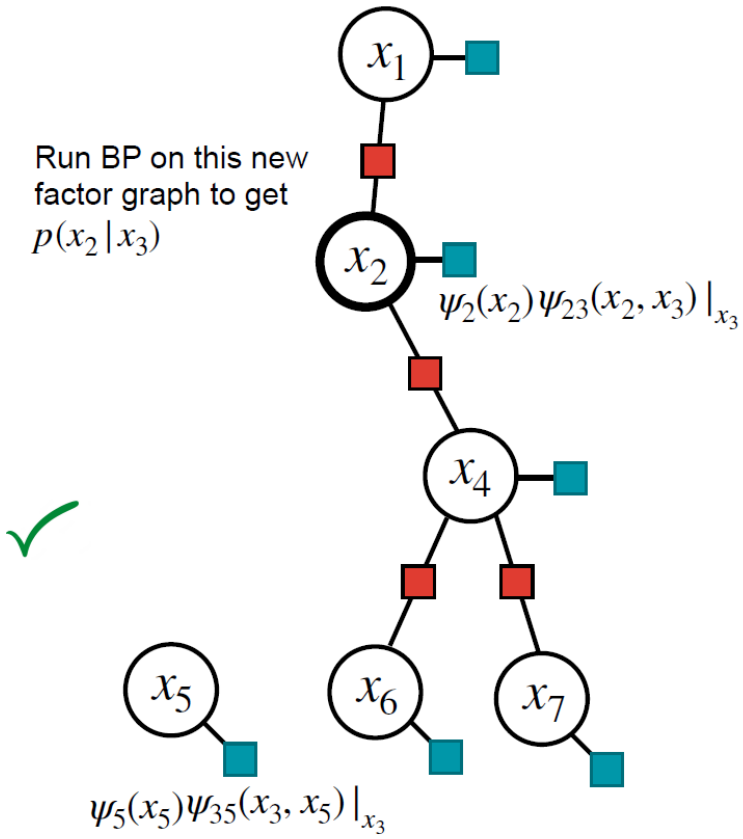


Figure 4.36: Checklist: Confirming that belief propagation allows computation of key distributions in a tree-structured factor graph.

However, we are unable to compute the mode using this vanilla belief propagation.

4.4 Extensions of Belief Propagation

4.4.1 Message Passing Protocol

The belief propagation (BP) algorithm follows a message passing protocol consisting of two main steps: **Message update:**

$$M_{j\to i}(x_i) = \sum_{x_j\in\{1,\dots,K\}} \psi_{ij}(x_i,x_j)\psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j).$$

State update:

$$p(x_i) \propto \psi_i(x_i) \prod_{j\to i} M_{j\to i}(x_i).$$

We assume that the graph is tree-structured. But what if not?

4.4.2 Extension 1: Continuous States

If states are continuous, i.e., $x_i \in \mathbb{R}^d$, the **summation** in the message update is replaced with an **integral**:

Message update:

$$M_{j\to i}(x_i) = \int_{\mathbb{R}^d} \psi_{ij}(x_i,x_j)\psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j)\, dx_j.$$

State update:

$$p(x_i) \propto \psi_i(x_i) \prod_{j\to i} M_{j\to i}(x_i).$$

The integral in the message update is often intractable, especially in high-dimensional settings (?). But in special cases such as **Gaussian belief propagation**, it can be computed analytically. (solution for non-Gaussian case: expectation propagation) (no convergence guarantee if non-tree structure).

4.4.3 Gaussian Belief Propagation

Product of Two Gaussian

The product of two Gaussian distributions results in another Gaussian distribution:

$$\mathcal{N}(x|a,A)\mathcal{N}(x|b,B) = \mathcal{N}(x|c,C),$$

where

$$c = C(A^{-1}a + B^{-1}b), \quad C = (A^{-1} + B^{-1})^{-1}.$$

This allows up to compute the message in closed form, by setting **both the potential and the message** to be Gaussian: **Message Update:**

$$M_{j\to i}(x_i) = \int_{\mathbb{R}^d} \psi_{ij}(x_i, x_j)\mathcal{N}(x_j|a, A)\, dx_j.$$

State Update (as a product of message):

$$p(x_i) = \mathcal{N}(x_i|\mu_i, \Sigma_i).$$

Integral of Gaussians is Gaussian

The integral of two Gaussian functions remains Gaussian:

1. $$\int_{\mathbb{R}^d} \mathcal{N}(x|Hx', R)\mathcal{N}(x'|a, A)\, dx' = \mathcal{N}(x|Ha, HAH^T + R).$$

2. $$\int_{\mathbb{R}^d} \mathcal{N}(x|Hx', R)\mathcal{N}(x|a, A)\, dx = \mathcal{N}(Hx'|a, A + R).$$

By choosing the pairwise potential as a form given above, based also on the previous property, we can write the whole integral as a Gaussian: **Message Update:**

$$M_{j\to i}(x_i) = \mathcal{N}(x_i|\mu_{j\to i}, \Sigma_{j\to i}).$$

State Update:

$$p(x_i) = \mathcal{N}(x_i|\mu_i, \Sigma_i).$$

4.4.4 Example: Time-series Modeling

Belief propagation can be applied to Time-series models, such as the **linear state-space model**:

$$z_{n+1} = Mz_n + e_n, \quad e_n \sim \mathcal{N}(0, Q),$$
$$x_n = Hz_n + \eta_n, \quad \eta_n \sim \mathcal{N}(0, R).$$

This can be re-expressed as conditional distribution and as factor graph:

$$p(z_{n+1}|z_n) = \mathcal{N}(z_{n+1}|Mz_n, Q),$$

$$p(x_n|z_n) = \mathcal{N}(x_n|Hz_n, R).$$

Running only the forward pass of BP is equivalent to the **Kalman filter**(computes the $p(z_n)$ given all observations till current time) , while running both forward and backward passes results in the **Rauch-Tung-Striebel smoother**.

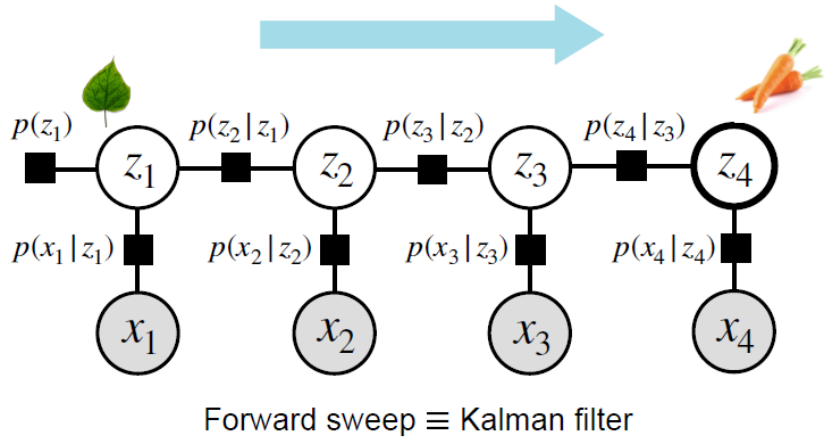


Figure 4.37: **The forward-backward sweep is equivalent to the Kalman filter.** The final latent variable z_4 is root node and we run a forward sweep from leaf node z_1.

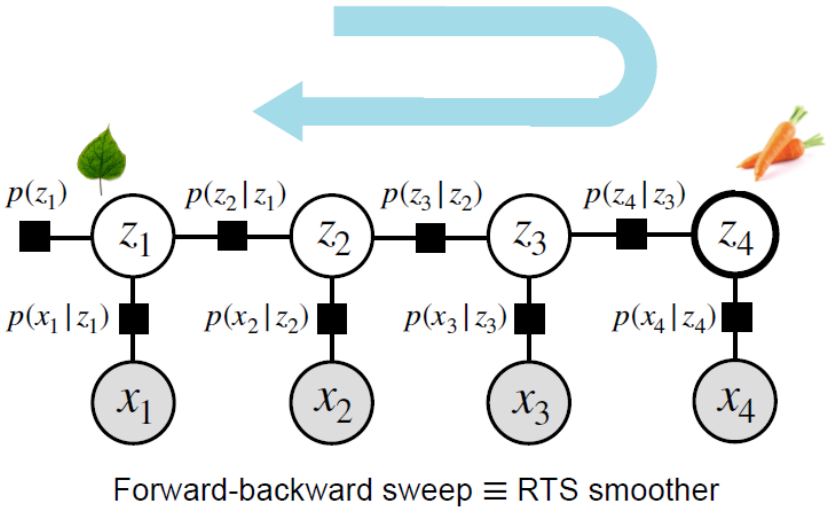


Figure 4.38: **The forward-backward sweep is equivalent to the RTS smoother.** It computes the $p(z_1)$ conditioning on all the observations from x_1 to x_4.

4.4.5 Extension 2: Max-Product Algorithm

By replacing the **sum in the message** update step with a **max operator**, we obtain the **max-product algorithm** to find the **mode**:

Message update:

$$M_{j\to i}(x_i) = \max_{x_j\in\{1,\dots,K\}} \psi_{ij}(x_i,x_j)\psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j).$$

After iterating from leaf nodes to the root, we obtain the mode:

$$\max_x p(x) = \max_{x_{\text{root}}\in\{1,\dots,K\}} \frac{1}{Z} \prod_{j\to \text{root}} M_{j\to \text{root}}(x_{\text{root}}).$$

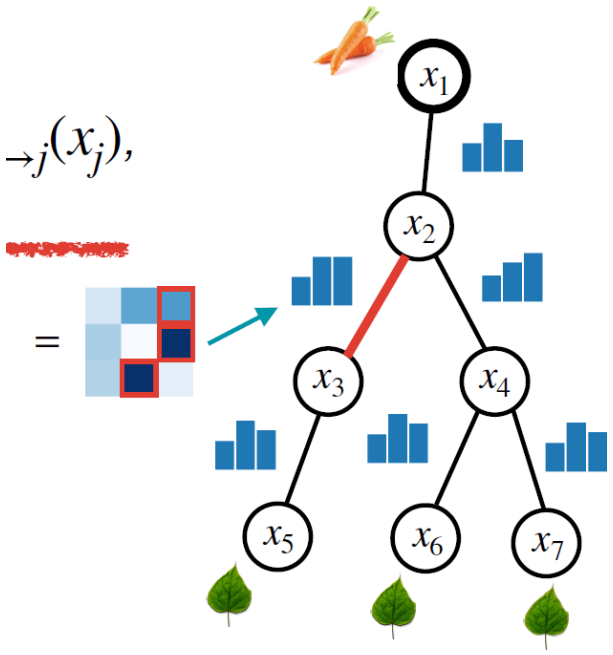


Figure 4.39: **Max-product algorithm**: finding the most probable state configuration using message passing. **The bar graphs represent messages, not states!**

Backtracking to Find the Mode

Once the root node is found, we backtrack to recover the mode:

1. Compute at the root:

$$x^*_{\text{root}} = \arg\max_{x_{\text{root}}} \frac{1}{Z} \prod_{j \to \text{root}} M_{j \to \text{root}}(x_{\text{root}}).$$

2. Propagate from the root back to the leaf nodes:

$$x^*_j = \arg\max_{x_j} \psi_{ij}(x^*_i, x_j)\psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \to j}(x_j), \quad j \sim i.$$

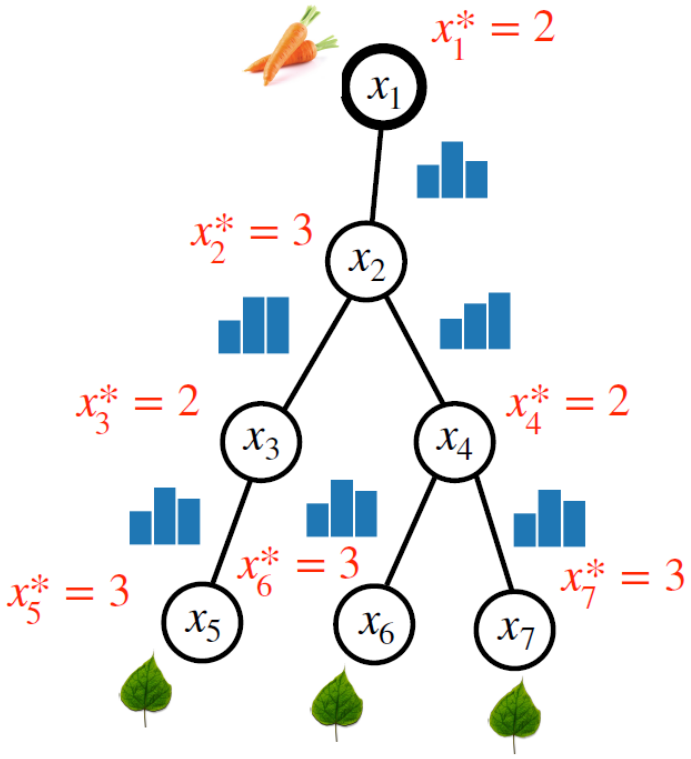


Figure 4.40: Backtracking to determine the most likely state configuration. Red denotes th most likely state (highest probability) of the nodes.

4.4.6 Extension 3: Polytrees and Other Graphs

A **polytree** is a **directed tree-like structure**. Belief propagation can be extended to polytrees and other graphical models by allowing more complex factorization:

- A polytree can be represented as a Markov random field (MRF). No longer necessarily a tree as may introduce cycles!
- Alternatively, it can be formulated as a factor graph. The factors are then no longer pairwise.

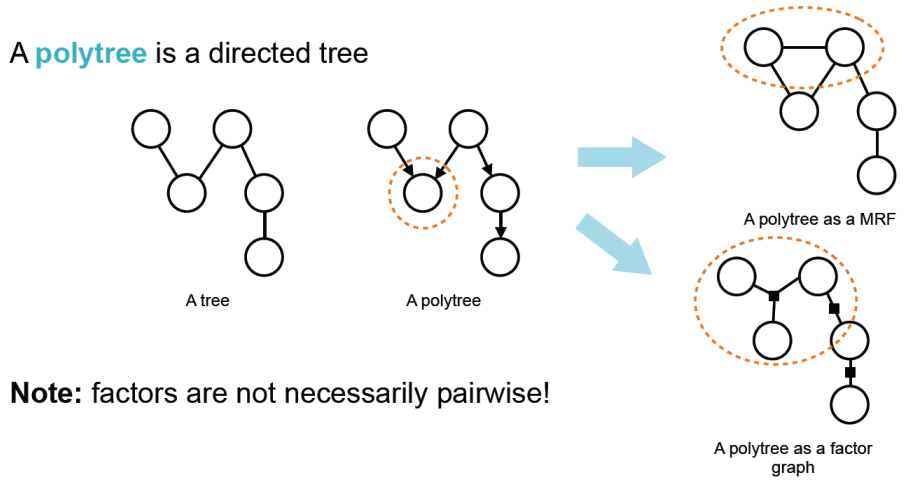


Figure 4.41: Polytrees: extending belief propagation beyond simple tree structures.

On trees, the message passing updates read:

$$M_{j\to i}(x_i) = \sum_{x_j\in\{1,\dots,K\}} \psi_{ij}(x_i,x_j)\psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j), \tag{4.1}$$

$$p(x_i) \propto \psi_i(x_i) \prod_{j\sim i} M_{j\to i}(x_i). \tag{4.2}$$

Factorized Message Updates

We can break down the message update into two sub-steps:

1. **Variable-to-Factor Message:**

$$\mu_{x_j\to\psi_{ij}}(x_j) = \psi_j(x_j) \prod_{k\sim j, k\neq i} M_{k\to j}(x_j). \tag{4.3}$$

Means aggregating all incoming messages to the source node x_j, which will be passed through the pairwise factor.

2. **Factor-to-Variable Message:**

$$\mu_{\psi_{ij}\to x_i}(x_i) = \sum_{x_j} \psi_{ij}(x_i, x_j)\mu_{x_j\to\psi_{ij}}(x_j). \tag{4.4}$$

Filter the variable-to-factor message through the pairwise potential, and convert it into a function in target node x_i.

This results in the final message update:

$$M_{j\to i}(x_i) = \mu_{\psi_{ij}\to x_i}(x_i). \tag{4.5}$$

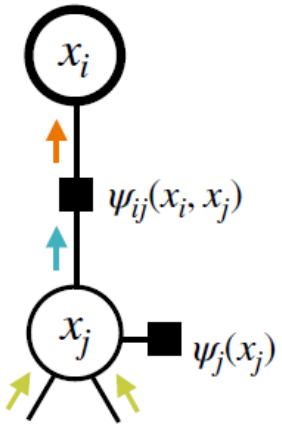


Figure 4.42

Extending to Polytrees

For polytrees, the update rules change as follows:

$$\mu_{x_j\to f_s}(x_j) = \prod_{l\in \mathrm{ne}(x_j)\setminus s} \mu_{f_l\to x_j}(x_j), \tag{4.6}$$

$$\mu_{f_s\to x_i}(x_i) = \sum_{x_{j_1},\ldots,x_{j_M}} f_s(x_i, x_{j_1}, \ldots, x_{j_M}) \prod_{k=1}^{M} \mu_{x_{j_k}\to f_s}(x_{j_k}). \tag{4.7}$$

Where the first step calculate the variable to factor message, which is aggregating all incoming messages to x_j . The second step calculates the factor to variable message, which filters all the messages through factor node f to produce a function of only variable x_i.

The state updates then read:

$$p(x_i) = \prod_{s \in \text{ne}(x_i)} \mu_{f_s \to x_i}(x_i). \tag{4.8}$$

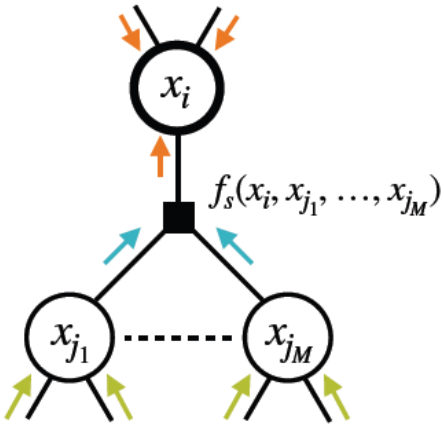


Figure 4.43

Loopy Belief Propagation (LBP)

For **more general graphs with loops** (before was tree/poly-tree) , we extend belief propagation to **Loopy Belief Propagation (LBP)**. The message updates remain the same, but LBP operates iteratively by:

- LBP start off and **initializes** messages as $\mu_{x \to f}(x) = 1$ for all variables x to factors f nodes.
- This is then passed to formula to calculate factor to variable message, then iterate.
- Updates are performed **in parallel(how?)** to speed up using the *flooding schedule*.

Implementation of Loopy Belief Propagation

During iterations of LBP:

- Approximate marginals are computed and refined over multiple iterations.
- True marginals converge as iterations progress.

4.4.7 Remarks

- LBP ***does not* have any convergence guarantee**. It can only give approximations to the true marginal distribution. The condition for convergence depends on the structure of the graph. Graphs with sparse connectivity or loops see more convergence. **(can re-weighting messages in trade-off of uncertainty estimate)**
- However, when it converges, results are often accurate.
- On **trees and polytrees**, convergence is guaranteed and we can get exact solutions .
- A notable variant of LBP is **Expectation Propagation (EP)**:
  - Approximates **intractable distributions** using a product of simpler ones.
  - Closeness is measured via **Kullback-Leibler (KL) divergence**. (moments - instead using ?)
  - EP generalizes LBP when applied to graphs.
- LBP is also related to Bethe free energy optimization(cost function to minimize).

4.5 Message Passing Neural Networks

4.5.1 Neural Networks

Neural networks have dominated machine learning in the past decade. They are:

- Extremely flexible for modeling
- Able to process complex data structures
- Composed of simple, parallelizable components e.g. matrix multiplication
- Automatically differentiable

The feedforward structure follows:

$$\begin{aligned} h^0 &= x \\ h^{l+1} &= \text{ReLU}(Wh^l + b), \quad t = 0, \ldots, L-1 \\ y &= \text{Softmax}(W^L h^L + b) \end{aligned}$$

4.5.2 Graphs in the Real World

Graphs appear in many real-world applications:

- Molecules as graphs (graph-level tasks)
- Citation networks (node-level tasks)
- Social networks
- Traffic networks

Example: Cora Dataset (node-level prediction task)

The Cora dataset contains:

- 2708 machine learning publications
- 5429 citation links
- Node feature size: 1433
- Seven classes

Task: Classify nodes according to topic.
Typically, MLP classification uses:

- Node features as inputs
- Seven topics as outputs

However, MLP may ignore relational information and the datasize is small and prone to overfitting without inductive bias.

Instead, we can use belief propagation. A Markov Random Field (MRF) is created with **pairwise potentials**:

$$\psi_{ij}(x_i, x_j) = \begin{cases} 0.9, & x_i = x_j \\ 0.0166..., & x_i \neq x_j \end{cases}$$

where $x_i = x_j$ means two neighbouring node belongs to same class.

Perform Loopy Belief Propagation (LBP) to compute $p(x_i|x^{obs})$. However:

- **This does not consider node features**
- **Pairwise potential is arbitrary**

4.5.3 Graph Neural Networks

Convolutional Neural Networks

CNNs have:

- Inductive bias of grid inputs
- Sparse connectivity due to local receptive fields
- Shared parameters

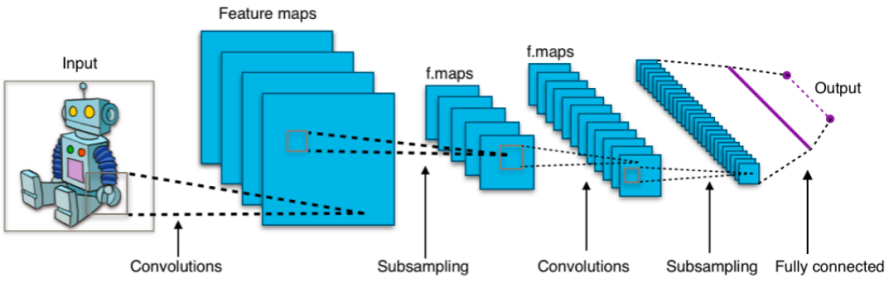


Figure 4.44: CNN Architecture.

Criteria for an Ideal Graph Neural Network

An ideal Graph Neural Network should:

- Have **computational and storage efficiency** (linear): $\mathcal{O}(|V| + |E|)$
- Have parameter size independent of input size (satisfied by CNN)
- Use only local information to construct hidden features, making the graph sparse
- Utilize edge features in addition to node features to make more informed predictions

Extending Convolutions to Graphs

CNNs are based on the discretization of the convolution operator:

$$\begin{aligned} f \star \psi_\theta(x) &= \int_{\mathbb{R}^2} f(y)\psi_\theta(x-y)dy \\ &\approx \sum_{y \in \mathbb{Z}^2} f(y)\psi_\theta(x-y) \end{aligned}$$

Can we define convolutions on graphs? Each node in graph can have different neighbors, making it hard to have uniform filter size like CNN.

4.5.4 Spectral Graph Convolution

Spectral graph convolution is based on the idea of applying the Fourier transform to graph signals, enabling a frequency-domain interpretation of graph convolutions.

Bruna et al. introduced **SpectralNe**t, leveraging the following property:

$$f \star \psi_\theta(x) = \mathcal{F}^{-1}(\mathcal{F}f \odot \mathcal{F}\psi_\theta)(x)$$

where $\mathcal{F}$ denotes the Fourier transform.

Graph Fourier Transform (GFT)

In the context of graphs, the Fourier transform is defined using the eigen-decomposition of the graph Laplacian:

1. **Graph Laplacian:**
$$L = D - A \tag{4.9}$$
where D is the degree matrix and A is the adjacency matrix.

2. **Spectral decomposition:**
$$L = U\Lambda U^T \tag{4.10}$$
where U contains the eigenvectors and Λ is the diagonal matrix of eigenvalues.

3. **Fourier transform and inverse:**
$$\mathcal{F}f := U^T f, \quad \mathcal{F}^{-1}\hat{f} := U\hat{f} \tag{4.11}$$

Applying these steps allows **spectral filtering** on graphs using convolution in the frequency domain.

Parameterized Spectral Filtering By parameterizing the filter response $\widehat{\psi_\theta}$ in the **Fourier domain**, we can simplify the spectral convolution as:

$$\widehat{\psi_\theta} = \theta \tag{4.12}$$

This means that instead of learning complex functions in the frequency domain, we directly learn a set of parameters θ that control the filter response.

The spectral graph convolution operation is equivalent to applying a learnable transformation in the frequency domain and then performing an inverse Fourier transform to map the filtered signal back to the original graph space.

Evaluating SpectralNet:

- Computational and storage efficiency: $\mathcal{O}(|V|^2)$ **quadratically in the size of Node.**
- Parameter size is $|V|$, thus **not independent of input size** as the parameter size scales linearly as the node size.
- Uses **non-local features due to Fourier domain representation** (diagonal features).
- **Cannot incorporate edge features** such as distance between nodes.

4.5.5 Graph Convolutional Networks (GCN)

Graph Convolutional Networks (GCNs) were introduced by Kipf and Welling as a spatial formulation of convolutional operations on graphs. The key idea behind GCNs is to **aggregate information from neighboring nodes**, enabling effective representation learning on graph-structured data.

GCN Update Rule

At each layer, node features are updated using a normalized sum of neighboring node features:

$$h_i^{l+1} = \text{ReLU}\left(\sum_{j \in \mathcal{N}_i} \frac{W^l}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}} h_j^l\right),$$

where:

- h_i^l is the feature representation of node i at layer l.
- $\mathcal{N}_i$ is the set of neighbors of node i.
- W^l is the learnable weight matrix at layer l.
- $|\mathcal{N}_i|$ and $|\mathcal{N}_j|$ are the degrees of nodes i and j, used for normalization.
- The ReLU activation function ensures non-linearity.

Adjacency Matrix Representation

The GCN update rule can be rewritten in matrix form using the adjacency matrix A and degree matrix D:

$$H^{l+1} = \text{ReLU}\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^l W^l\right),$$

where:

- $\widetilde{A} = A + I$ is the adjacency matrix with added self-loops.

- $\widetilde{D}$ is the degree matrix of $\widetilde{A}$.
- H^l is the matrix of node embeddings at layer l.
- W^l is the learnable weight matrix.

Evaluating Graph Convolutional Networks

1. **Computational and storage efficiency:**
   - Computational cost is linear $\mathcal{O}(|V|CF)$ (multiplication $h_j^l W^l$ performed $|V|$ times)
   - Storage cost is $\mathcal{O}(|E|)$ (to store adjacency matrix A)
2. **Parameter size**:
   - Parameter size is $\mathcal{O}(CF)$ per layer to store $W^l \in \mathbb{R}^{C \times F}$
3. **By construction, hidden features only depend on local neighbors**
4. **Vanilla GCN does not use edge features in the original formulation**

4.5.6 Semi-Supervised Learning with GCN

We apply such case when the number of labelled data-points are small. But relations between labelled and unlabelled data exist. The Cora dataset experiment:

- Only 140 nodes for training
- 1000 nodes for testing
- Train with cross-entropy loss over labelled data (i.e. training data):

$$L = - \sum_{(y,X) \in \mathcal{D}_L} y \log \text{GCN}(X)$$

GCN achieves 81.1% accuracy, compared to 57.1% for MLP and 48.6% for Loopy Belief Propagation. This tells us the importance of using both node features and relational information to make predictions, as done using GCN.

4.5.7 Message Passing Neural Networks (MPNN)

MPNNs was developed to predict properties of molecules. It introduces a general framework for learning features on graphs based on message passing, and it can handle graph data containing both node and edge features.

MPNNs generalize GCNs and many other architectures for message, state update, and readout(in the final layer):

$$
\begin{aligned}
M_{j\to i}^{l} &= M_{\theta}^{l}(h_i^l, h_j^l, e_{ij}) \\
h_i^{l+1} &= U_{\theta}^{l}(h_i^l, \square_{j\sim i} M_{j\to i}^{l}) \\
y &= R_{\theta}(\{h_i^L \mid i \in V\})
\end{aligned}
$$

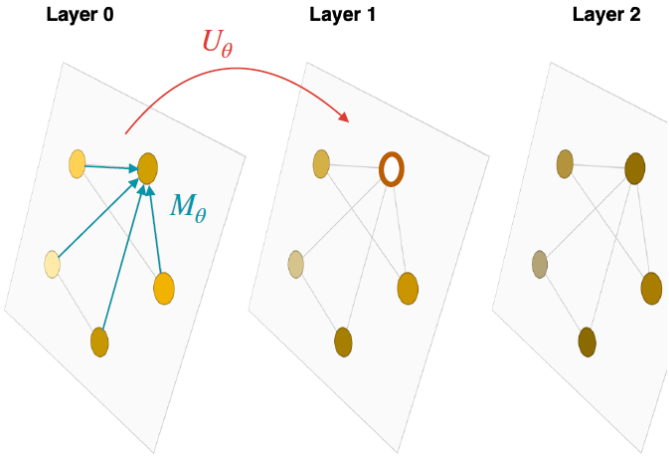


Figure 4.45: Message Passing Neural Network Layers.

GCN as MPNN

$$
\begin{aligned}
M_{\theta}^{l}(h_i^l, h_j^l, e_{ij}) &= \frac{1}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}} h_j^l \\
U_{\theta}^{l}(h_i^l, \square_{j\sim i} M_{j\to i}^{l}) &= \text{ReLU}\left(\frac{1}{|\mathcal{N}_i|} h_i^l + \sum_{j\sim i} M_{\theta}^{l}(h_i^l, h_j^l, e_{ij}) W^l\right)
\end{aligned}
$$

dimension of latent/input - unchanged? max pooling? - bottleneck (convolution cases) autoencoder/diffusion to graph cases (e.g. citation network, using set of nodes as input to generate graphs etc.)

MPNN by Gilmer

The original work of Gilmer et al. [5] used the following MPNN model:

- $M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}) = \text{MLP}(e_{ij})h_{v_j}^l$
- $U_\theta^l(h_{v_i}^l, \square_{j\sim i} M_{j\to i}^l) = \text{GRU}\left(h_{v_i}^l, \sum_{j\sim i} M_{j\to i}^l\right)$

to predict 13 quantum properties of molecules in the QM9 dataset. (GAT - discrete cases with bag of word embeddings)

Model performs extremely well with 11 out of 13 properties reaching “chemical accuracy”.

Transformers as MPNN

Transformers fit into MPNN:

$$M_\theta^l(h_i^l, h_j^l, e_{ij}) = \text{MultiheadAttention}(h_i^l, h_j^l)$$

The message function uses **Multi-head Attention**, meaning that instead of simple sum/mean/max aggregation, it **learns** importance weights w_{ij} for each neighboring node. This weight is computed using a **self-attention mechanism**, which allows the model to assign different importance to different neighbors.

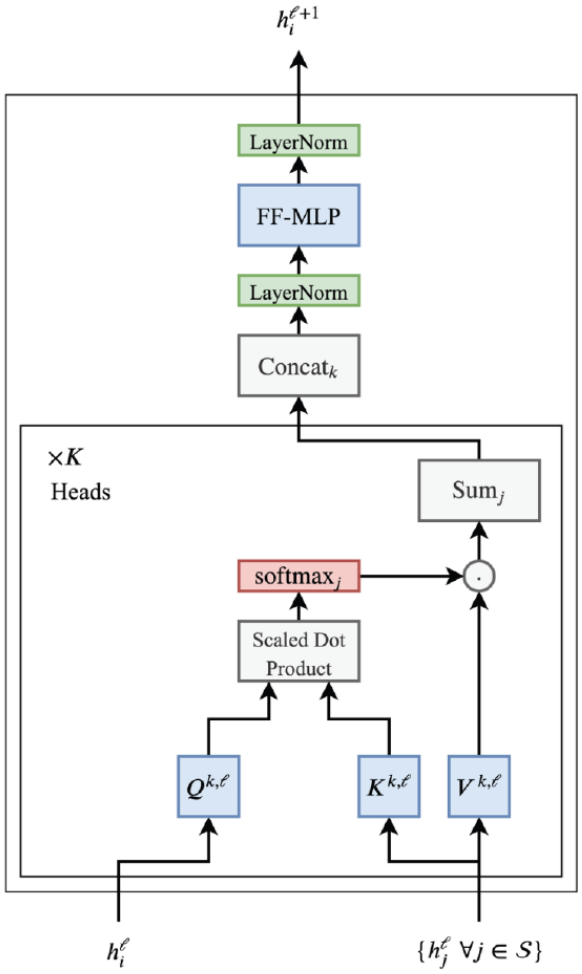


Figure 4.46: Transformer as MPNN.

4.5.8 Comparison of MPNN with LBP

LBP
Bayesian. Coupling between neighbours arise from prior knowledge of model. Message passing rule follows fro
Iterative. States are updated iteratively to obtain better estimates of marginals. (iteration analogy to receptive i
Interpretable. Prior assumptions are usually quite simple (only using relational information), making predictio

Table 4.1: Comparison between LBP and MPNN

Chapter 5

Meta Learning

5.1 Meta-Learning: Introduction

5.1.1 What is Meta-Learning?

Definition 5.1.1 (Meta-Learning)**.** Meta-learning is the process of learning how to learn. Instead of training a model for a single task, we train a model that can **adapt to new tasks efficiently**. Unlike traditional machine learning models, which require extensive training for each new task, meta-learning enables models to **generalize across different but related tasks**.

Meta-learning addresses questions such as:

- If we have already solved n problems, should solving the $(n+1)$-th problem be easier?
- How can learning algorithms accumulate experience over different but related problems?
- Can we automate the design of learning rules and representations instead of hand-crafting them?

5.1.2 Challenges in Machine Learning

Theorem 5.1.2 (Data Requirements)**.** *Deep learning models generally require **large amounts of labeled training data**. This poses limitations in domains where data is expensive or scarce, such as personalized medicine, personalized recommendations, and rare-language translation.*

Remark. *Human learning differs significantly from traditional machine learning: humans can generalize from **very few examples**, while deep learning models require massive datasets like thousands of even millions of examples.*

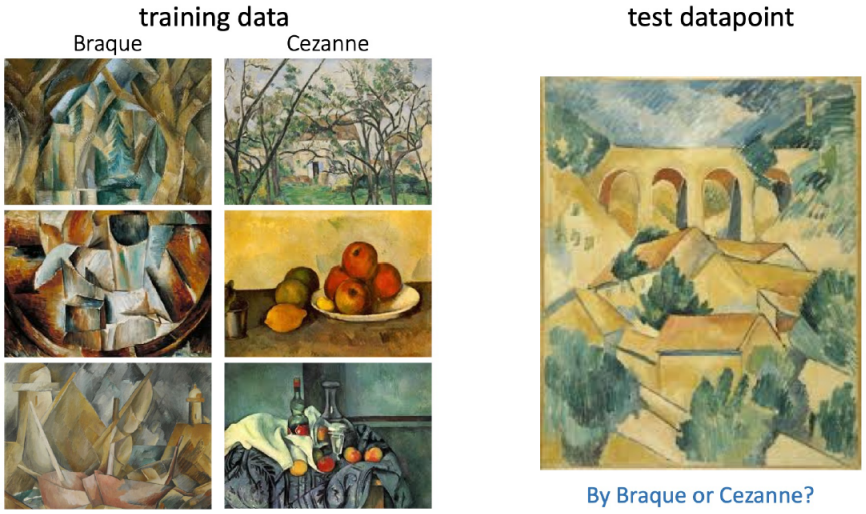


Figure 5.1: Human Learning Example.

5.2 Meta-Learning Framework

5.2.1 Learning to Learn: A General Framework

The goal in meta-learning is: more tasks, leading to better learning algorithms.

Definition 5.2.1 (Base Learner)**.** A base learner is a machine learning model trained to solve **a specific task** (e.g. a single classification problem) using an **inner learning algorithm**.

Definition 5.2.2 (Meta-Learner)**.** A meta-learning or outer learning algorithm improves performance over **multiple learning episodes** by training the inner learning algorithm. A **learning episode** consists of a base algorithm, a model, and a performance measure such as generalization error or convergence speed. Multiple learning episodes taken together provide the training data for the meta learning algorithm.

For example, if we have multiple classification tasks, the meta-learner tries to find a set of parameters or a strategy that will allow a model to learn *any* new classification task faster or with fewer samples.

5.2.2 Terminology and Cost per Episode

In our base learner, for each particular task or episode,

- we train on our **training set**, *[support set]*

- and evaluate on a **test set**. *[query set]*

For meta-learning a learning algorithm A or its parameters ω,

- we train on a **meta-training set**, composed of many different tasks, *[training set]*
- and evaluate on a **meta-test set** of unseen tasks or data. *[test set]*

Definition 5.2.3 (Meta-Learning Task)**.** A task T is defined as a dataset D and a loss function L, i.e., $T = \{D, L\}$.

5.2.3 Cost per episode

Theorem 5.2.4 (Meta Loss Minimization)**.** *Each episode's objective is to do well on that episode's test set after training on that episode's training set. An episode minimizes the loss L on a single dataset $D = \{D_{train}, D_{test}\}$. In a probabilistic model with base model parameters θ for a classification problem:*

$$L(D; \omega) = \frac{1}{|D_{test}|} \sum_{\{x_i, y_i\} \in D_{test}} -\log p(y_i | x_i, \theta^*) \tag{5.1}$$

where θ^* for a particular episode is obtained by running the base learning algorithm A with meta-learner parameters ω:

$$\theta^* = A(L, D_{train}; \omega) \tag{5.2}$$

θ^* is then tested on D_{test} for that task. The meta-learner updates w based on performance across many such episodes.

5.2.4 Distributions of tasks

One way to formalize meta-learning is to learn an algorithm that performs well across a distribution of tasks, which we denote $p(\mathcal{T})$.

- We can then write the overall meta-learning objective as

$$\min_{\omega} \mathcal{L}^{\text{meta}}(\omega) = \mathbb{E}_{\{\mathcal{L}, \mathcal{D}\} \sim p(\mathcal{T})} \big[\mathcal{L}(\mathcal{D}; \omega) \big].$$

- If we have M initial example tasks $\mathcal{T}^1, \ldots, \mathcal{T}^M \sim p(\mathcal{T})$, then our target optimal ω^* is

$$\omega^* = \arg\min_{\omega} \sum_{m=1}^{M} \mathcal{L}^m(\mathcal{D}^m; \omega).$$

- In the meta-test phase, we can then make predictions on new training sets $\mathcal{D}'_{\text{train}}$ for a new task by using ω^*, with

$$\theta^* = A(\mathcal{L}', \mathcal{D}'_{\text{train}}; \omega^*).$$

In practice, this often corresponds to solving a two-level optimization problem.

Example of different tasks

- Image classi
  cation, but on **previously unseen classes**
- Image classi
  cation, but in **new lighting conditions** (or weather conditions, etc)
- Same objective, but on new data: e.g. customized to a **different person**
- Same conceptual goal, but **different loss** or objective function
- Machine translation, but on **new languages** or with **new vocabulary**

Different tasks need to share at least some structure. But actually, many things, we might want to do using machine learning have a lot of shared structure such as distribution of images.

(In real-life setting, the distribution of data in different tasks can have modalities differences which need to be resolved.)

5.3 Meta-Learning-Adjacent Fields

A lot of different subareas in machine learning are similar to meta-learning.

5.3.1 Multi-Task Learning

Multi-task learning also considers a dataset of many different tasks at training time. However, it aims to learn a single set of parameters θ which are **appropriate for all tasks** (instead of learning meta-parameters ω). It is also generally assumed that all tasks are **simultaneously accessible**. It's about a single model that tries to handle multiple tasks simultaneously without a 'learned learning process'.

5.3.2 Transfer Learning

Transfer learning aims to take knowledge from previous tasks and use it to **accelerate future tasks**. This typically involves fitting θ to previous task(s), and then using those parameters as initialization on new tasks and fine-tune on the new target tasks.

Remark. *The main difference of transfer learning from meta-learning is that there is* ***no meta-objective*** *— the previous tasks are not trained in a way that is aware that* θ ***will be used on new data later****.*

I.e. Transfer learning typically does not optimize for the eventual adaptation. It just pre-trains on a larget dataset and then fine-tunes. Meta-learning explicitly trains with the knowledge that it will adapt to new tasks later.

5.3.3 AutoML

AutoML aims to automate large parts of the machine learning pipeline — from **algorithm selection to dataset transformations to neural network architectures**. It is generally a broader topic than meta-learning, but meta-learning can be useful.

For example, searching for neural architectures is slow and expensive, so it is desirable to find architectures that are generally helpful across a wide variety of problems, and to leverage knowledge that existing architectures (and algorithms) are useful on particular datasets.

5.4 Key Aspects of Meta-Learning Papers

Hospedales et al. (2020) set out the following useful taxonomy for meta-learning:

5.4.1 Three Key Components

- **Meta-representation: What** is being meta-learned? This could be (for example) an **initial value of model parameters**, but could be anything useful for learning subsequent tasks.
- **Meta-optimizer: How** do we do the outer-level optimization of ω? This might be **gradient descent through the inner optimization of** θ, or could be something else entirely (RL, random search, . . .).
- **Meta-objective: Why** are we doing any of this? The meta-objective corresponds to the **choice of task distribution** and meta-loss.

5.5 Amortization

One other aspect to keep an eye out for is the amount of **amortization**.

- Idea of **amortizing runtime costs**: pay a large computation or learning cost upfront, in exchange for fast and cheaper computation (e.g. inference, adaptation) later.
- Popular, for example, in running Bayesian inference repeatedly using the same model across different datasets: *we "amortize" the cost of doing inference repeatedly by learning a neural network that can directly produce approximate posterior distributions for new tasks.*
- Different meta-learning methods have different degrees of amortization
  - **Fully amortized**: A "black-box" method that *directly* outputs model parameters for a new task in one shot (very little adaptation cost).
  - **Less amortized**: Methods like MAML, which still require a **gradient-based fine-tuning step** for each new task, but they've "learned" a good starting point.

5.6 Case Study: Few-Shot Learning

5.6.1 Few-Shot Learning as Meta-Learning

Definition 5.6.1 (Few-Shot Learning)**.** Few-shot learning is the process of training models that can generalize to new tasks using only a handful of training examples (e.g. 1-shot or 5-shot).

If only having 1 or 5 examples per class, classical ML solution tend to overfit. Meta-learning solutions can use prior knowledge from other tasks to adapt effectively.

Example 5.6.2. Suppose we want to classify an image with very few examples (maybe only one!). Our objective is to learn an algorithm A that outputs optimal parameters θ for a model M given a small dataset D.

If we collect multiple few-shot learning tasks, we can train the algorithm A directly on data. However, this requires an appropriate meta-objective.

5.6.2 Real-world Few-shot Data

Example 5.6.3 (Omniglot Dataset)**.** The Omniglot dataset consists of:

- 1,623 classes.
- Spread across 50 different alphabets.
- Only 20 images per class.

This is often referred to as "transposed MNIST" since it presents a high-class but low-instance setting. If your meta-learning method can handle 1- or 5-shot classification on Omniglot, it shows promise in truly data-scarce scenarios.

5.6.3 Example Task: N-Way K-Shot Classification

Example 5.6.4 (5-Way 1-Shot Classification)**.** We are given:

- 5 images, 5 classes, each with one labeled training example, as a training set.
- A test set containing two new samples, and they are from classes we've never seen before.
- The model must predict the correct class for each test example.

Since these test images belong to previously unseen classes, standard supervised learning fails, requiring a meta-learning approach.

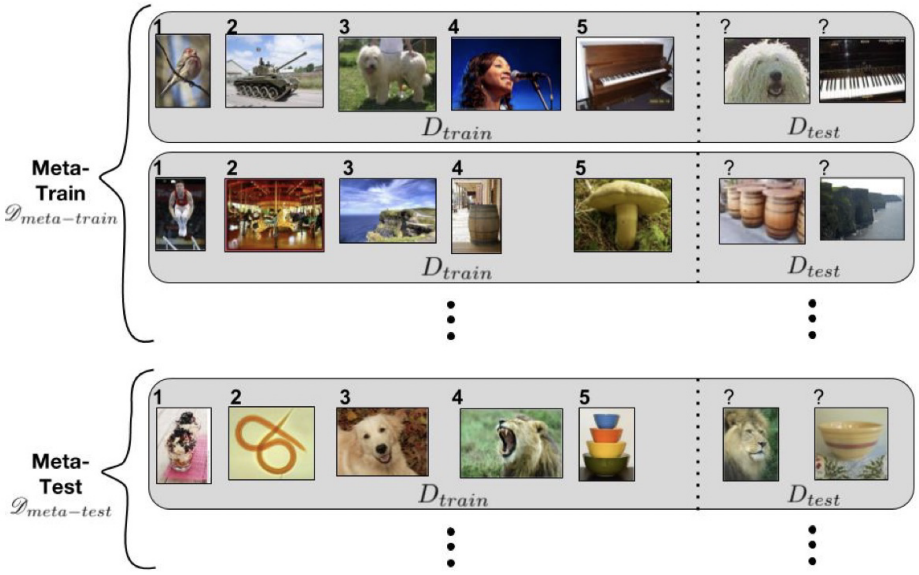


Figure 5.2: Mini-ImageNet Examples.

5.6.4 Meta-Representations for Few-Shot Learning

One approach is to start from an existing machine learning algorithm, add extra parameters like neural embeddings that are learned meta-parametrically. We will look at three of these base learner + meta learner combination for classification:

- **kNN or kernel classifier** $\rightarrow$ **Matching Network**s: Uses an attention mechanism to compare new examples to stored instances
- **Gaussian classifier** $\rightarrow$ **Prototypical Networks**: Embeds data into a metric space and classifies based on nearest prototypes.
- **Gradient descent** $\rightarrow$ **Model-Agnostic Meta Learning (MAML)**: Learns a general initialization from which models can quickly adapt to new tasks.

It's also possible to take a completely black-box approach, where the entire algorithm is learned by a neural network:

- **Memory-augmented neural networks**, Santoro et al. (2016)
- **SNAIL**, Mishra et al. (2018)

5.6.5 Matching Networks

Definition 5.6.5 (Matching Networks)**.** A non-parametric method that classifies new examples based on similarity to a few labeled examples using an attention mechanism.

New point x' is a weighted sum of the labels of the few support points x_i, y_i:

$$\hat{y} = \sum_{i=1}^{M} a(x', x_i) y_i, \tag{5.3}$$

where x_i are labeled examples in D_{train}, and $a(x', x_i)$ is a learned similarity function. (x_i, y_i) are the training set.

Using an attention weight to compute via a similarity measure (often cosine) in a learned embedding space:

$$a(x', x_i) = \frac{\exp\{c(f(x'), g(x_i))\}}{\sum_{j=1}^{M} \exp\{c(f(x'), g(x_j))\}}, \tag{5.4}$$

where f and g are learnable functions, and $c(f, g)$ is the cosine similarity:

$$c(f, g) = 1 - \frac{f^\top g}{\|f\|\|g\|}. \tag{5.5}$$

Advantages of Matching Networks

- Non-parametric Method
- Can recover nearest neighbors and kernel density estimators by choosing a particular form of a
- Can also function as an associative memory that points to nearest training examples

Trick in MN

One trick: instead of just $f_\theta(x')$ and $g_\theta(x_i)$, Matching Networks can also incorporate the entire training set inito the embedding function e.g. by passinig them through an LSTM:

- $f_\theta(x') = f_\theta(x', D_{\text{train}})$
- $g_\theta(x_i) = g_\theta(x_i, D_{\text{train}})$

where both are parameterized by sequence models like LSTM. This can capture richer relationships among all support examples.

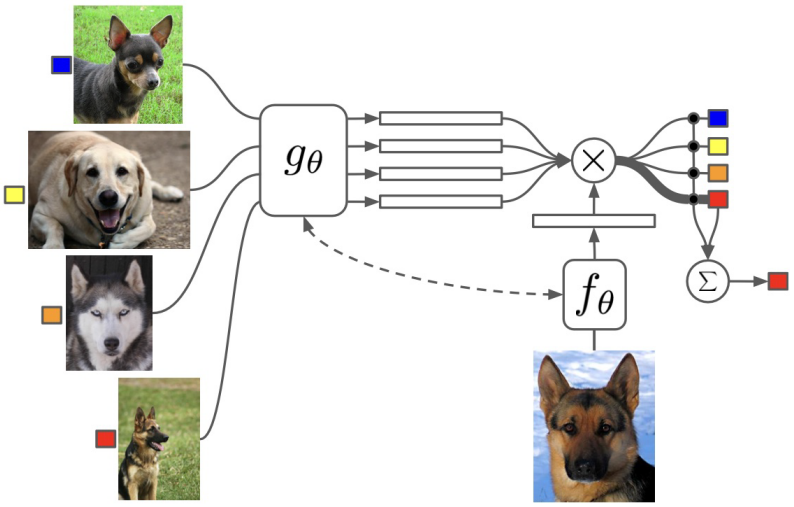


Figure 5.3: Left: Input to embedding $g(\theta)$

5.6.6 Prototypical Networks

Definition 5.6.6 (Prototypical Networks)**.** A method that learns an embedding function $f_\theta(x)$ so that each class can be summarized by the mean of its support poiints in the embedding space (the "prototype")

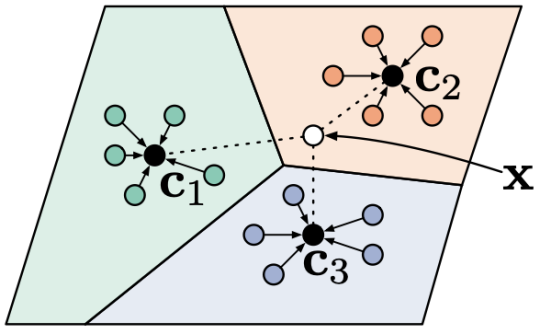


Figure 5.4: Prototypical Networks.

Class centroids are computed as:

$$c_k = \frac{1}{|S_k|} \sum_{(x_i, y_i) \in S_k} f_\theta(x_i), \tag{5.6}$$

where S_k contains training points with class label $y_i = k$.

Classification for new example x' is given as the nearest prototype in that embedding space:

$$p(y = k|x') = \frac{\exp\{-d(f_\theta(x'), c_k)\}}{\sum_{j=1}^{K} \exp\{-d(f_\theta(x'), c_j)\}}, \tag{5.7}$$

where $d(\cdot, \cdot)$ is a Euclidean distance function.

Embedding Interpretability

Since class predictions only depend on the **means of the embeddings** c_k, this is much simpler than matching networks — there is **no need for a complex embedding of the training sets**.

- With a Euclidean distance $d(\cdot, \cdot) = \|f_\theta(x') - c_k\|^2$, this is interpretable as learning a linear model in the embedding space $f_\theta(x)$:

$$-\|f_\theta(x') - c_k\|^2 = -f_\theta(x')^\top f_\theta(x') + 2c_k^\top f_\theta(x') - c_k^\top c_k \tag{5.8}$$

$$= \underbrace{(2c_k)^\top}_{w_k} f_\theta(x') - \underbrace{c_k^\top c_k}_{b_k} + \text{const} \tag{5.9}$$

- The learned network f_θ produces, in a feed-forward manner, the embedding space as well as the weights of the corresponding linear classifier.

Example Embedding for an Omniglot Meta-Test Task

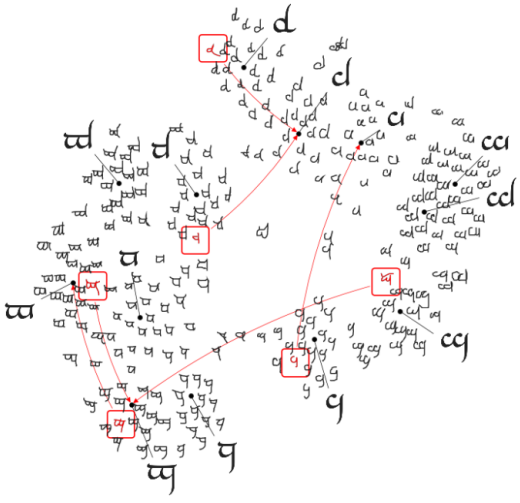


Figure 5.5: Omniglot Meta-Test Task.

Example 5.6.7 (t-SNE Visualization)**.** An example embedding for an Omniglot meta-test task:

- **T-SNE visualization**: Helps visualize high-dimensional embeddings.
- **Black highlights**: Represent class centers.
- **Red highlights**: Indicate misclassification errors.

5.6.7 Model-Agnostic Meta Learning (MAML)

Definition 5.6.8 (MAML)**.** MN and PN are both mainly for specialized classification setting. MAML learns a general initialization for model parameters θ that can quickly fine-tuned to new tasks using a few gradient steps.

Unlike transfer learning, where fine-tuning starts from a pre-trained model trained on unrelated tasks, MAML **explicitly optimizes for fast adaptation**. Given task-specific data D_i, the inner update fine-tunes from that initialization with gradient descent on a few exmaples of the new task:

$$\theta_i' = \theta - \alpha \nabla_\theta L(D_i; \theta). \tag{5.10}$$

And the "outer loop" updates the initilization itself to facilitate such quick adaptation:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(\mathcal{T})} \mathcal{L}_{T_i}(f_{\theta_i'}) \tag{5.11}$$

The meta-objective then optimizes across tasks:

$$\min_\theta \sum_i L(D_i; \theta_i'). \tag{5.12}$$

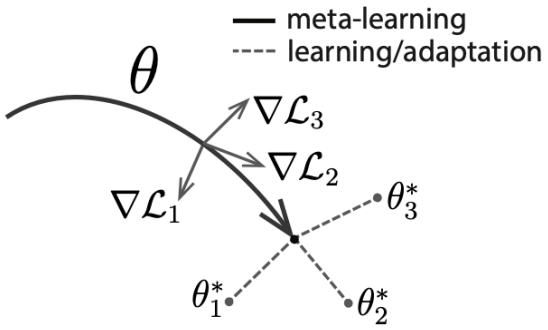


Figure 5.6: MAML

Remark. *MAML is highly versatile and "model-agnostic", as it applies to any model optimized via gradient descent. You can plug in convolutional nets, recurrent nets, etc., as long as you can backprop through them.*

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
0: Randomly initialize θ
0: **while** not done **do**
0: Sample batch of tasks $T_i \sim p(\mathcal{T})$
0: **for** each task T_i **do**
0: Evaluate $\nabla_\theta \mathcal{L}_{T_i}(f_\theta)$ with respect to K examples
0: Compute adapted parameters with gradient descent:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{T_i}(f_\theta)$$

0: **end for**
0: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(\mathcal{T})} \mathcal{L}_{T_i}(f_{\theta_i'})$
0: **end while**=0

5.6.8 Black-box Approach

Train a neural architecture to directly output predictions **at the "next" task**.

1. Train a network which directly map an entire dataset D^i_{train} to model parameters w_i for task T_i:

$$\omega_i = f_\theta(D^i_{\text{train}}) \tag{5.13}$$

2. Make predictions on test data with

$$y = g_{\omega_i}(x_{i,\text{test}}) \tag{5.14}$$

which is effectively a single forward pass: "Given this dataset, produce a classifier".

Clear training objective — this is now just supervised learning. However, while direct, it can be computationally heavy and typically needs sophisticated neural architectures to handle variable-size datasets.

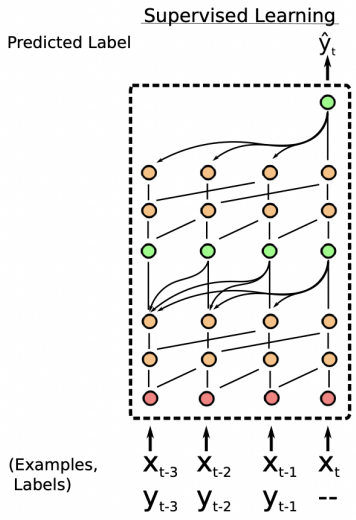


Figure 5.7: Black-box Approach

5.6.9 Other Things You Can Meta-Learn

- **Neural network architectures:** Many architectures (e.g., convnets, resnets, LSTMs, GNNs) have strong inductive biases. Can we learn architectures (or building blocks) that generalize well across many tasks?
- **Optimizers:** An optimizer is a function that takes a current value $\theta^{(t)}$ and a gradient g, returning a new value $\theta^{(t+1)}$. Instead of using a fixed algorithm like Adam or SGD, can we learn the update rule itself?
- **Losses or rewards:** Can we design alternative objective functions for inner learning tasks, such as loss functions robust to label noise? Can these generalize to other datasets?
- **Data augmentation:** Given many label-preserving transformations (rotations, flips, color shifts etc.), which should we include during training? Can we learn a data augmentation policy?
- **Domain Adaptation, Continual Learning, and Defense Against Adversarial Attacks...**