

# Gaussian Processes

Marc Deisenroth

Centre for Artificial Intelligence

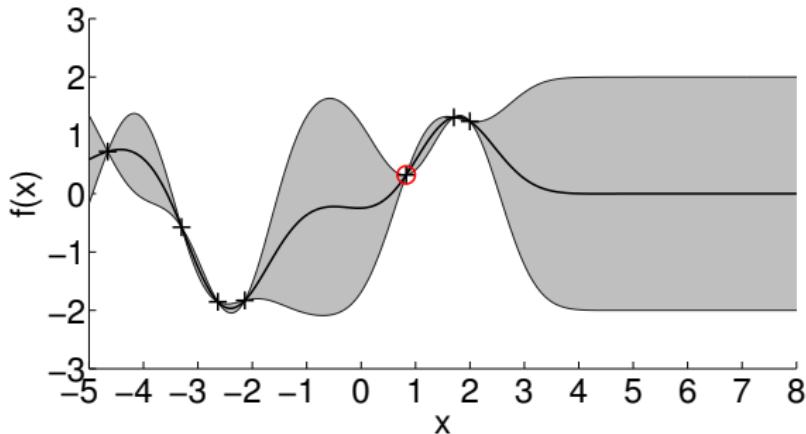
Department of Computer Science

University College London

 @mpd37

m.deisenroth@ucl.ac.uk

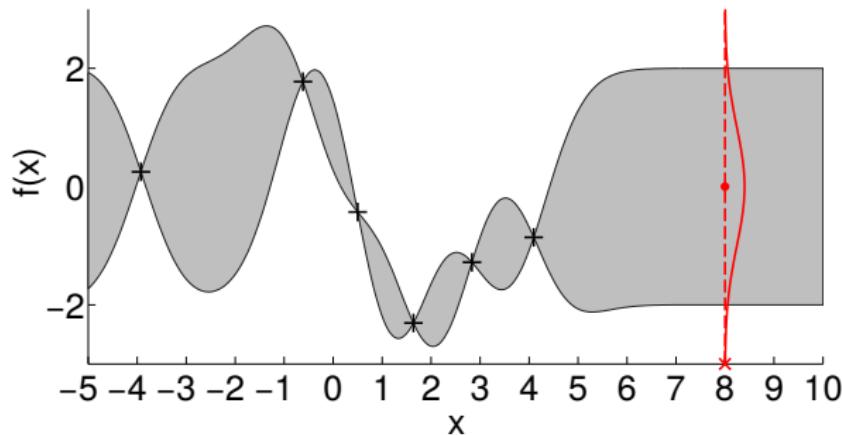
<https://deisenroth.cc>



## Objective

For a set of observations  $y_i = f(\mathbf{x}_i) + \varepsilon$ ,  $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$ , find a distribution over functions  $p(f)$  that explains the data

- ▶ Probabilistic regression problem

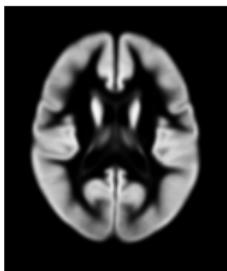
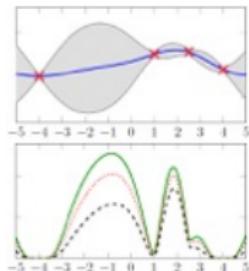
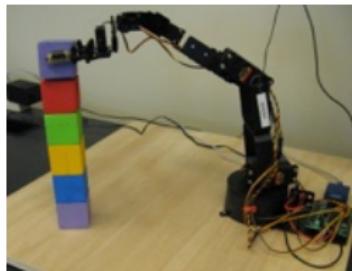


## Objective

For a set of observations  $y_i = f(\mathbf{x}_i) + \varepsilon$ ,  $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$ , find a distribution over functions  $p(f)$  that explains the data

► Probabilistic regression problem

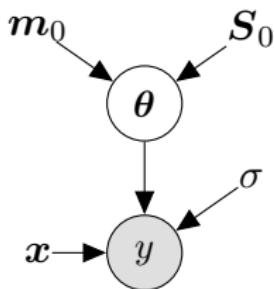
# Some Application Areas



- Reinforcement learning and robotics
- Bayesian optimization (experimental design)
- Geostatistics
- Sensor networks
- Time-series modeling and forecasting
- High-energy physics
- Medical applications

$$\text{Prior} \quad p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{m}_0, \boldsymbol{S}_0)$$

$$\begin{aligned} \text{Likelihood} \quad & p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}(y | \boldsymbol{\phi}^\top(\boldsymbol{x})\boldsymbol{\theta}, \sigma_n^2) \\ & \implies y = \boldsymbol{\phi}^\top(\boldsymbol{x})\boldsymbol{\theta} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_n^2) \end{aligned}$$

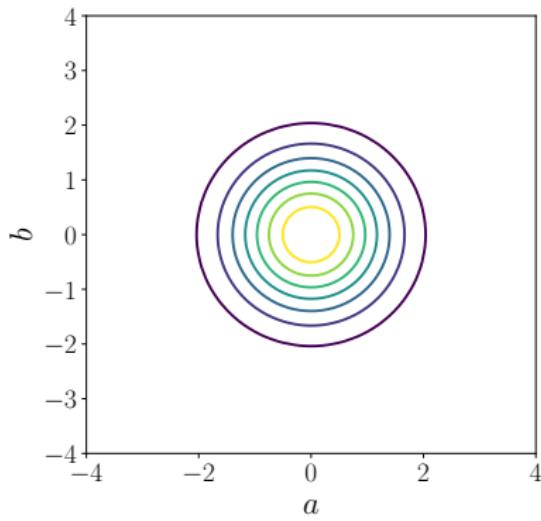


- Parameter  $\boldsymbol{\theta}$  becomes a latent (random) variable
- Distribution  $p(\boldsymbol{\theta})$  induces a **distribution over plausible functions**
- Choose a conjugate Gaussian prior
  - Gaussian posterior  $p(\boldsymbol{\theta}|X, y) = \mathcal{N}(\boldsymbol{\theta} | \boldsymbol{m}_N, \boldsymbol{S}_N)$
  - Closed-form computations (e.g., predictions, marginal likelihood)

# Distribution over Functions

Consider a linear regression setting

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



Consider a linear regression setting

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

$$p(a, b) = \mathcal{N}(\mathbf{0}, \mathbf{I})$$

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

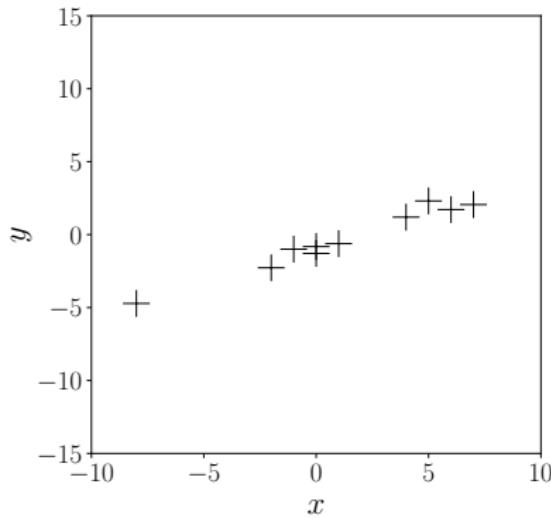
# Sampling from the Posterior over Functions

Consider a linear regression setting

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

$$p(a, b) = \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$\mathbf{X} = [x_1, \dots, x_N], \mathbf{y} = [y_1, \dots, y_N]$  Training data



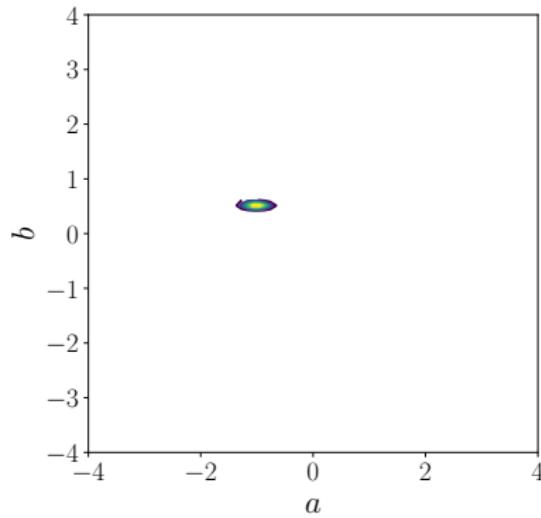
# Sampling from the Posterior over Functions

Consider a linear regression setting

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

$$p(a, b) = \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$p(a, b | \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{m}_N, \mathbf{S}_N) \quad \text{Posterior}$$



- Instead of sampling parameters, which induce a distribution over functions, **sample functions directly**
  - ▶ Place a prior on functions
  - ▶ Make assumptions on the distribution of functions
- Intuition: function = infinitely long vector of function values
  - ▶ Make assumptions on the distribution of function values
    - ▶ **Gaussian process**

# Overview

1 Gaussian Process: Definition

2 Regression as Inference

- GP Prior

- Likelihood

- Marginal Likelihood

- Posterior

- Predictions

3 Model Selection

- GP Training

- Hyper-Parameters

- Inspection of the Marginal Likelihood

- Covariance Function

4 Limitations and Guidelines

5 Application Areas

## Gaussian Process: Definition

- We will place a distribution  $p(f)$  on functions  $f$
- Informally, a function can be considered an infinitely long vector of function values  $f = [f_1, f_2, f_3, \dots]$
- A Gaussian process is a generalization of a multivariate Gaussian distribution to infinitely many variables.

## Definition (Rasmussen & Williams, 2006)

A **Gaussian process** (GP) is a collection of random variables  $f_1, f_2, \dots$ , any finite number of which is Gaussian distributed.

- A Gaussian distribution is specified by a mean vector  $\mu$  and a covariance matrix  $\Sigma$
- A Gaussian process is specified by a **mean function**  $m(\cdot)$  and a **covariance function (kernel)**  $k(\cdot, \cdot)$  ► More on this later

# Regression as Inference

## Objective

For a set of observations  $y_i = f(\mathbf{x}_i) + \epsilon$ ,  $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$ , find a (posterior) distribution over functions  $p(f(\cdot)|\mathbf{X}, \mathbf{y})$  that explains the data. Here:  $\mathbf{X}$  training inputs,  $\mathbf{y}$  training targets

Training data:  $\mathbf{X}, \mathbf{y}$ . Bayes' theorem yields

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

Prior:  $p(f(\cdot)) = GP(m, k)$  ➤ Specify mean  $m$  function and kernel  $k$ .

Likelihood (noise model):  $p(\mathbf{y}|f(\mathbf{X})) = \mathcal{N}(f(\mathbf{X}), \sigma_n^2 \mathbf{I})$

Marginal likelihood (evidence):  $p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|f(\cdot), \mathbf{X}) p(f(\cdot)|\mathbf{X}) df$

Posterior:  $p(f(\cdot)|\mathbf{y}, \mathbf{X}) = GP(m_{\text{post}}, k_{\text{post}})$

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

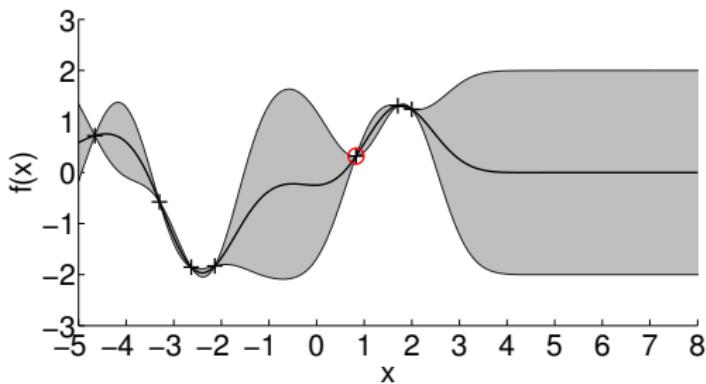
Bayesian linear regression:

- Prior  $p(\boldsymbol{\theta})$  on the parameters  $\boldsymbol{\theta}$  allows us to encode some properties of the parameters (e.g., range, reasonable values, ...)
- Every sample  $\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta})$  induces a function  $f_i(\cdot) := \boldsymbol{\theta}_i^\top \phi(\cdot)$

Gaussian process:

- GP prior:  $p(f(\cdot))$
- Function plays the role of the parameters
  - ▶ Every sample  $f_i(\cdot) \sim GP$  is a function

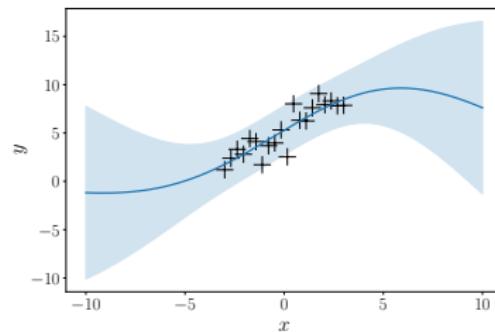
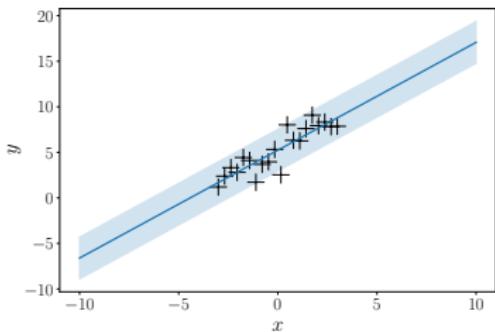
- Bayesian prior specifies assumptions on the quantity of interest
- What assumptions could we make on the underlying function?
- What characterizes the function we want to model?
  - Mean function
  - Covariance function



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

- The **average function** of the distribution over functions
- Allows us to **bias the model** (can make sense in application-specific settings)

# Mean Function (2)



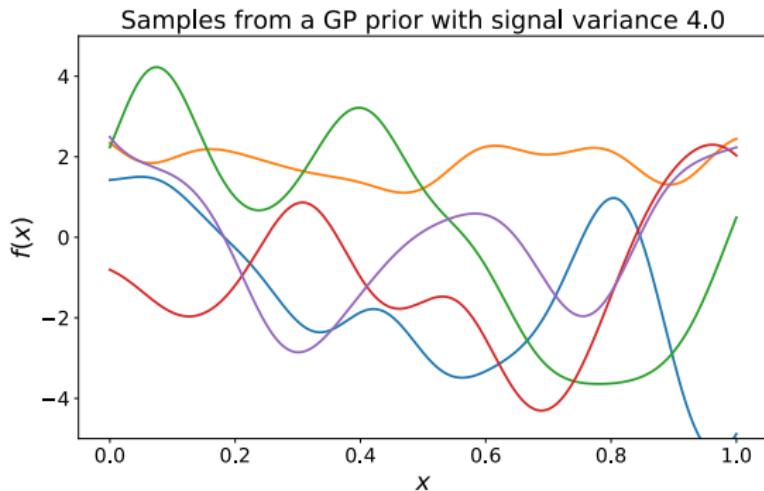
- Can be a parametrized function, e.g., linear, exponential, or neural network. Example:  $m_\theta(x) = \theta^\top \phi(x)$
- Prior mean function  $m_\theta$  can incorporate **problem-specific prior knowledge** (e.g., in robotics, natural sciences)
- Can simplify the learning problem
- Often: “Agnostic” mean function in the absence of data or prior knowledge:  $m(\cdot) \equiv 0$  everywhere (for symmetry reasons)

- Covariance function (kernel) is symmetric and positive semi-definite
- Compute covariances/correlations between (unknown) function values by just looking at the corresponding inputs:

$$\text{Cov}[f(\boldsymbol{x}_i), f(\boldsymbol{x}_j)] = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

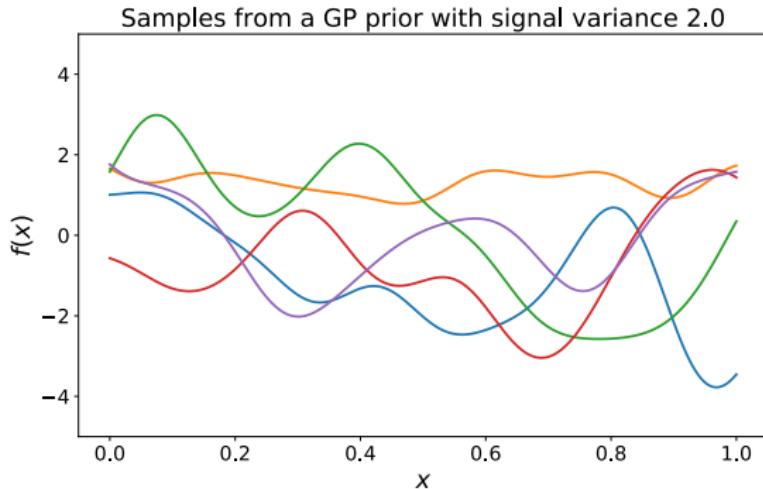
- ▶ Kernel trick (Schölkopf & Smola, 2002)
- Encodes high-level structural assumptions (e.g., smoothness, periodicity) of the function we want to model

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



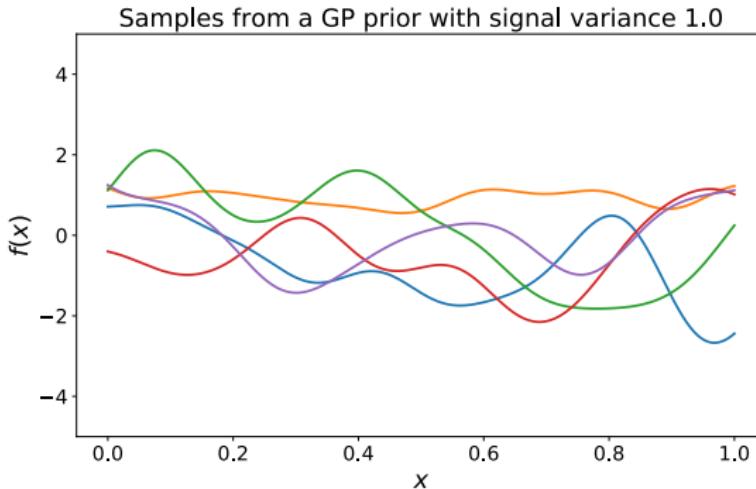
- Controls the amplitude (vertical magnitude) of the function we wish to model

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



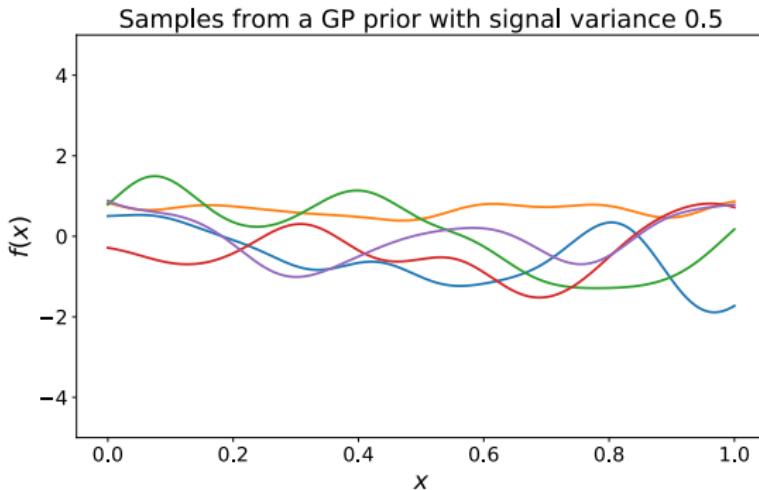
- Controls the amplitude (vertical magnitude) of the function we wish to model

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



- Controls the amplitude (vertical magnitude) of the function we wish to model

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$

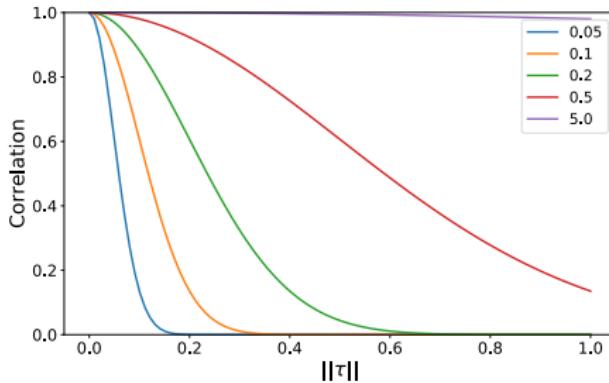


- Controls the amplitude (vertical magnitude) of the function we wish to model

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$

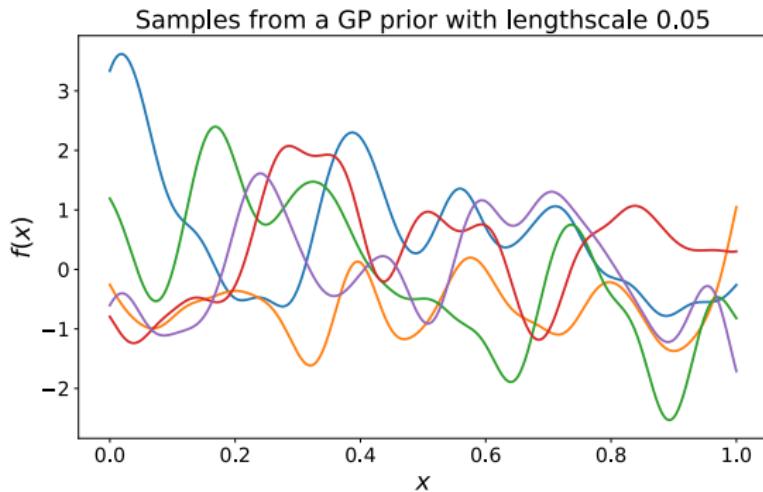
- How “wiggly” is the function?
- How much information we can transfer to other function values?
  - ▶ Correlation between function values
- How far do we have to move in input space from  $\mathbf{x}$  to  $\mathbf{x}'$  to make  $f(\mathbf{x})$  and  $f(\mathbf{x}')$  uncorrelated?

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



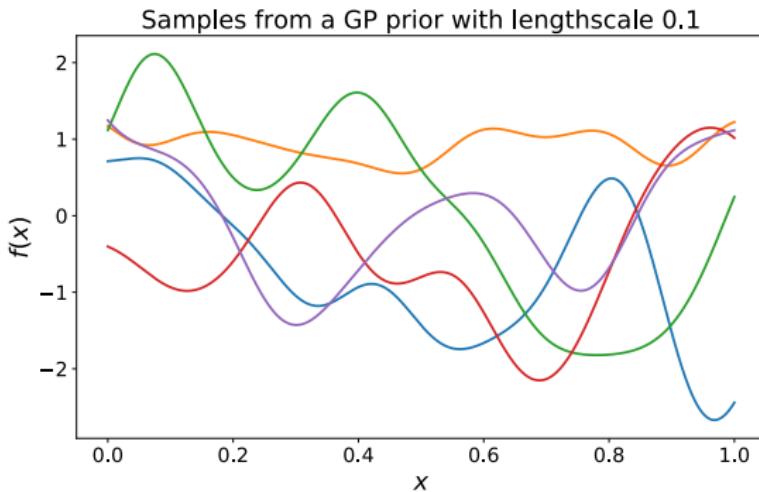
- Correlation between function values  $f(\mathbf{x})$  and  $f(\mathbf{x}')$  depends on the (scaled) distance  $\|\tau\|/\ell = \|\mathbf{x} - \mathbf{x}'\|/\ell$  of the corresponding inputs.
- What does a short/long length-scale  $\ell$  imply?

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



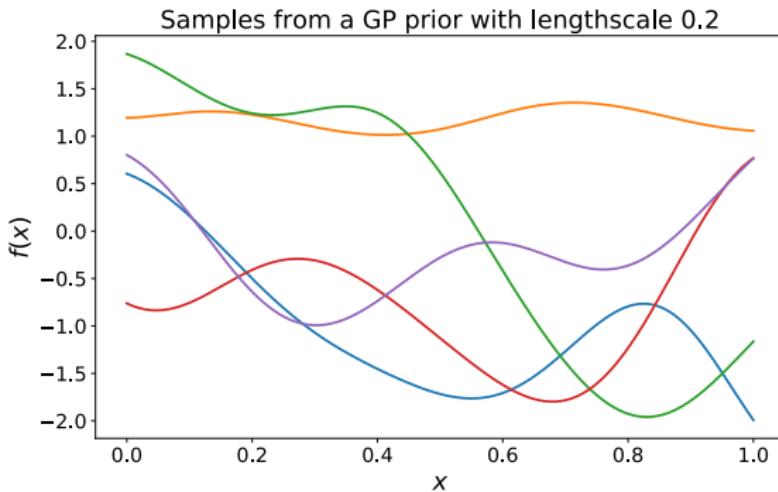
► Explore interactive diagrams at  
<https://drafts.distill.pub/gp/>

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



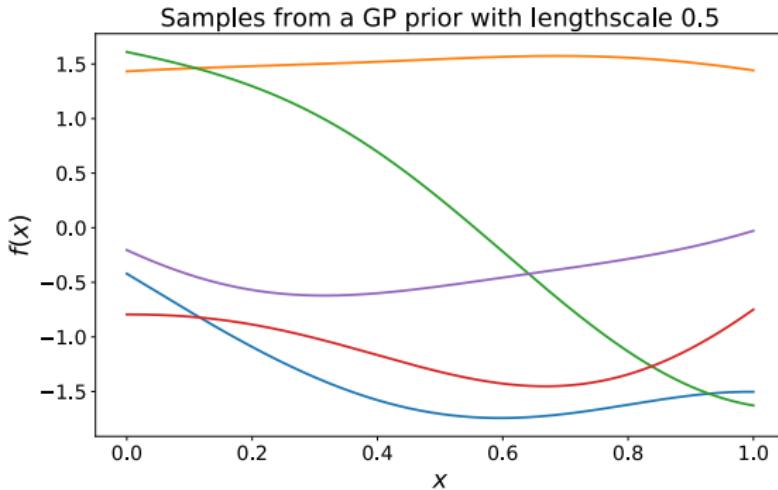
► Explore interactive diagrams at  
<https://drafts.distill.pub/gp/>

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$



► Explore interactive diagrams at  
<https://drafts.distill.pub/gp/>

$$k_{Gauss}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j)/\ell^2\right)$$

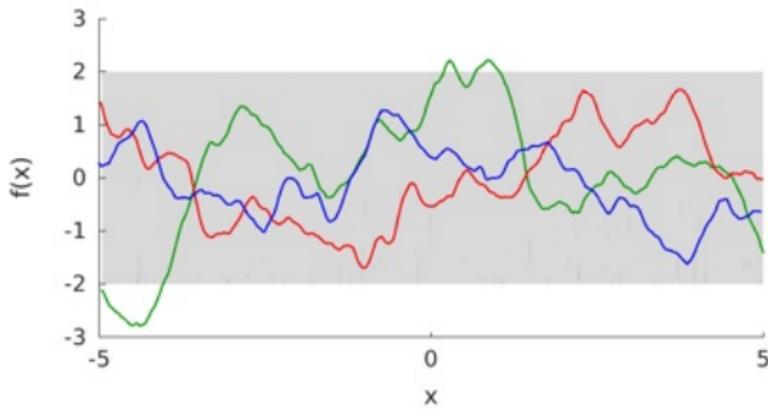


► Explore interactive diagrams at  
<https://drafts.distill.pub/gp/>

# Matérn Covariance Function

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

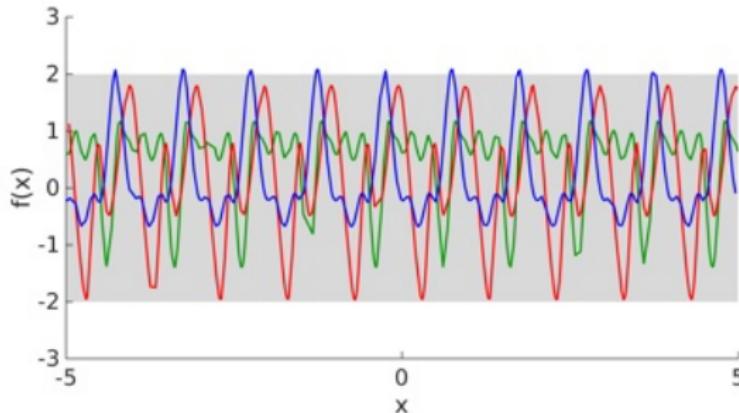
- Assumption on latent function: 1-times differentiable
- $\sigma_f$ : Amplitude of the latent function
- $\ell$ : Length-scale. How far do we have to move in input space before the function value changes significantly?



# Periodic Covariance Function

$$k_{per}(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{2 \sin^2\left(\frac{\kappa(x_i - x_j)}{2\pi}\right)}{\ell^2}\right)$$
$$= k_{Gauss}(\mathbf{u}(x_i), \mathbf{u}(x_j)), \quad \mathbf{u}(x) = \begin{bmatrix} \cos(\kappa x) \\ \sin(\kappa x) \end{bmatrix}$$

- Assumption on latent function: **periodic**
- **Periodicity parameter  $\kappa$**



Assume  $k_1$  and  $k_2$  are valid covariance functions and  $u(\cdot)$  is a (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 (MacKay, 1998)
  - ▶ Periodic covariance function
  - ▶ Manifold Gaussian process (Calandra et al., 2016)
  - ▶ Deep kernel learning (Wilson et al., 2016)
- ▶ Automatic Statistician (Lloyd et al., 2014)

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

Gaussian likelihood in linear regression:

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

- Function (not a distribution) of the parameters
- Describes how parameters and observed data are connected
- Tells us how to transform parameters into (noisy) data

Gaussian likelihood in Gaussian processes:

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

- Intuition: Parameters are the function  $f$  itself

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

Bayesian linear regression with a Gaussian prior  $p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ :

$$\begin{aligned} p(\mathbf{y} | \mathbf{X}) &= \int p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \\ &= \mathcal{N}(\mathbf{y} | \mathbf{0}, \Phi \Phi^\top + \sigma^2 \mathbf{I}) \\ &= \mathbb{E}_{\boldsymbol{\theta}}[p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta})] \end{aligned}$$

- Normalizes the posterior distribution
- Can be computed analytically
- Expected likelihood (under the parameter prior)
- Expected predictive distribution of the training targets  $\mathbf{y}$  (under the parameter prior)

Gaussian process marginal likelihood

$$\begin{aligned} p(\mathbf{y}|\mathbf{X}) &= \int p(\mathbf{y}|f(\mathbf{X}))p(f(\mathbf{X}))df \\ &= \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K} + \sigma^2 \mathbf{I}) \\ &= \mathbb{E}_f[p(\mathbf{y}|f(\mathbf{X}))] \end{aligned}$$

- Normalizes the posterior distribution
- Can be computed analytically
- Expected likelihood (under the GP prior)
- Expected predictive distribution of the training targets  $\mathbf{y}$  (under the GP prior)

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

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j), \quad i, j = 1, \dots, N$$

Posterior over functions (with training data  $\mathbf{X}, \mathbf{y}$ ):

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

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

$$\begin{aligned} p(\mathbf{y} | f(\mathbf{X})) p(f(\cdot)) &= \mathcal{N}(\mathbf{y} | f(\mathbf{X}), \sigma_n^2 \mathbf{I}) GP(m(\cdot), k(\cdot, \cdot)) \\ &= Z \times GP(m_{\text{post}}(\cdot), k_{\text{post}}(\cdot, \cdot)) \end{aligned}$$

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

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

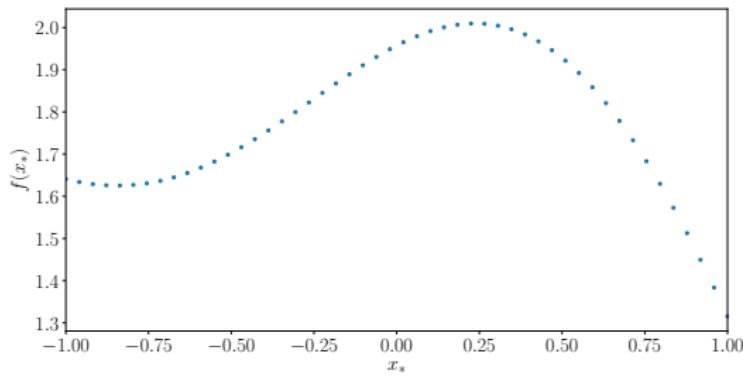
Marginal likelihood:

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

# Sampling from the GP Prior

- GP is a distribution over functions
  - ▶ A sample from a GP will be an entire function
- In practice, we cannot sample functions directly
- Instead: function = collection of function values
- Determine function values at a finite set of input locations

$$\mathbf{X}_* = [\mathbf{x}_*^{(1)}, \dots, \mathbf{x}_*^{(K)}]$$



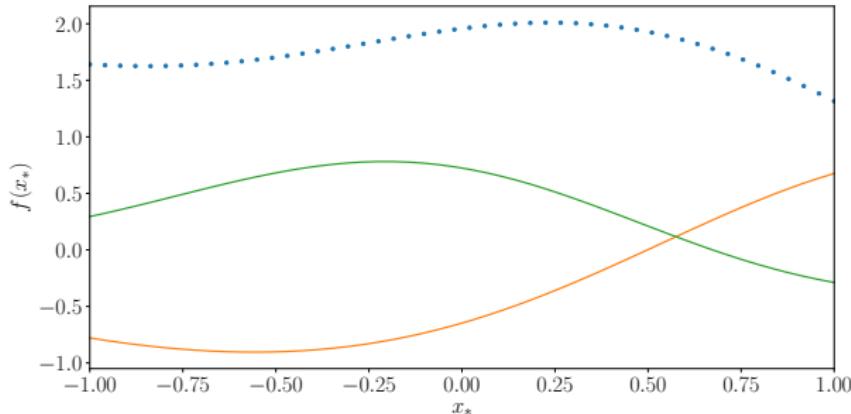
- Without any training data, the predictive distribution at test points  $\mathbf{X}_*$  is

$$\begin{aligned} p(\mathbf{f}(\mathbf{X}_*)|\mathbf{X}_*) &= \mathcal{N}\left(\mathbb{E}_f[f(\mathbf{X}_*)], \mathbb{V}_f[f(\mathbf{X}_*)]\right) \\ &= \mathcal{N}\left(m_{\text{prior}}(\mathbf{X}_*), k_{\text{prior}}(\mathbf{X}_*, \mathbf{X}_*)\right) \end{aligned}$$

- Exploited: Definition of GP that **all function values are jointly Gaussian distributed**
- Generate “function draws” (samples from the GP prior)

$$f_k(\mathbf{X}_*) \sim \mathcal{N}\left(m_{\text{prior}}(\mathbf{X}_*), k_{\text{prior}}(\mathbf{X}_*, \mathbf{X}_*)\right)$$

# Sampling from the GP Prior (3)



- Goal: Generate random functions  $f_k$ , so that

$$f_k(\mathbf{X}_*) \sim \mathcal{N}(m_{\text{prior}}(\mathbf{X}_*), k_{\text{prior}}(\mathbf{X}_*, \mathbf{X}_*))$$

- Define  $\mathbf{m}_* := m_{\text{prior}}(\mathbf{X}_*)$  and  $\mathbf{K}_{**} := k_{\text{prior}}(\mathbf{X}_*, \mathbf{X}_*)$ . Then

$$f_k(\mathbf{X}_*) \sim \mathcal{N}(\mathbf{m}_*, \mathbf{K}_{**})$$

► Sample from a multivariate Gaussian

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

- **Objective:** Find  $p(f(\mathbf{X}_*)|\mathbf{X}, \mathbf{y}, \mathbf{X}_*)$  for training data  $\mathbf{X}, \mathbf{y}$  and test inputs  $\mathbf{X}_*$ .
- GP prior at training inputs:  $p(f|\mathbf{X}) = \mathcal{N}(m(\mathbf{X}), \mathbf{K})$
- Gaussian Likelihood:  $p(\mathbf{y}|f, \mathbf{X}) = \mathcal{N}(f(\mathbf{X}), \sigma_n^2 \mathbf{I})$
- With  $f \sim GP$  it follows that  $f, f_*$  are jointly Gaussian distributed:

$$p(\mathbf{f}, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*) = \mathcal{N} \left( \begin{bmatrix} m(\mathbf{X}) \\ m(\mathbf{X}_*) \end{bmatrix}, \begin{bmatrix} \mathbf{K} & k(\mathbf{X}, \mathbf{X}_*) \\ k(\mathbf{X}_*, \mathbf{X}) & k(\mathbf{X}_*, \mathbf{X}_*) \end{bmatrix} \right)$$

- Due to the Gaussian likelihood, we also get ( $\mathbf{f}$  is unobserved)

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

Prior evaluated at  $\mathbf{X}, \mathbf{X}_*$ :

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

Posterior predictive distribution  $p(\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*)$  at test inputs  $\mathbf{X}_*$  obtained by Gaussian conditioning:

$$p(\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*) = \mathcal{N} \left( \mathbb{E}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*], \mathbb{V}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*] \right)$$

$$\mathbb{E}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*] = \underbrace{m(\mathbf{X}_*)}_{\text{prior mean}} + \underbrace{k(\mathbf{X}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1}}_{\text{"Kalman gain"}} \underbrace{(\mathbf{y} - m(\mathbf{X}))}_{\text{error}}$$

$$\mathbb{V}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*] = \underbrace{k(\mathbf{X}_*, \mathbf{X}_*)}_{\text{prior variance}} - \underbrace{k(\mathbf{X}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{X}_*)}_{\geq 0}$$

# Sanity Check

- GP posterior (from earlier):

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

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

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

- GP posterior predictions at  $\mathbf{X}_*$ :

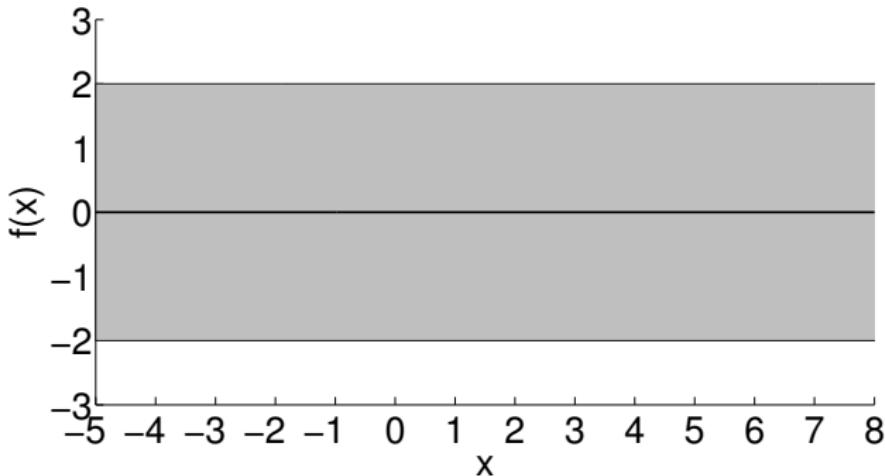
$$p(\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*) = \mathcal{N}(\mathbb{E}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*], \mathbb{V}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*])$$

$$\mathbb{E}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*] = m(\mathbf{X}_*) + k(\mathbf{X}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1}(\mathbf{y} - m(\mathbf{X}))$$

$$\mathbb{V}[\mathbf{f}_* | \mathbf{X}, \mathbf{y}, \mathbf{X}_*] = k(\mathbf{X}_*, \mathbf{X}_*) - k(\mathbf{X}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1}k(\mathbf{X}, \mathbf{X}_*)$$

## Predictions

Make predictions by evaluating the GP posterior mean and covariance function at a finite number of inputs  $\mathbf{X}_*$

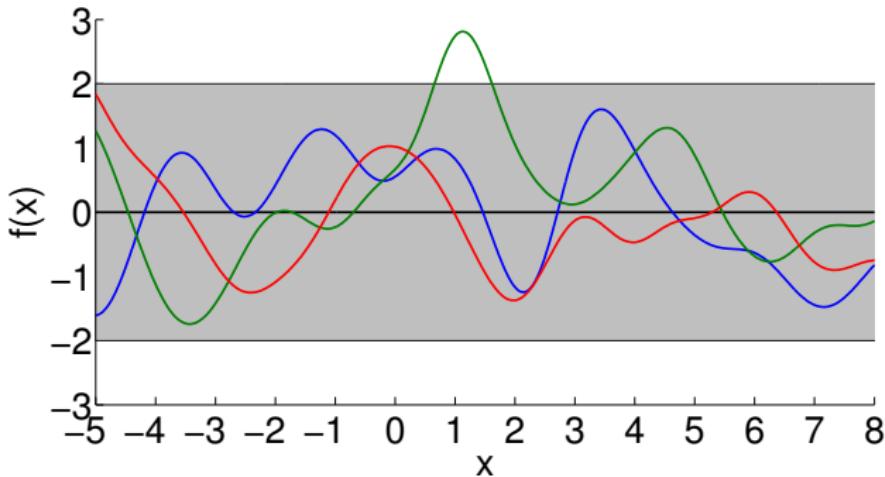


Prior belief about the function

Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\boldsymbol{x}_*) | \boldsymbol{x}_*, \emptyset] = m(\boldsymbol{x}_*) = 0$$

$$\mathbb{V}[f(\boldsymbol{x}_*) | \boldsymbol{x}_*, \emptyset] = \sigma^2(\boldsymbol{x}_*) = k(\boldsymbol{x}_*, \boldsymbol{x}_*)$$



Prior belief about the function

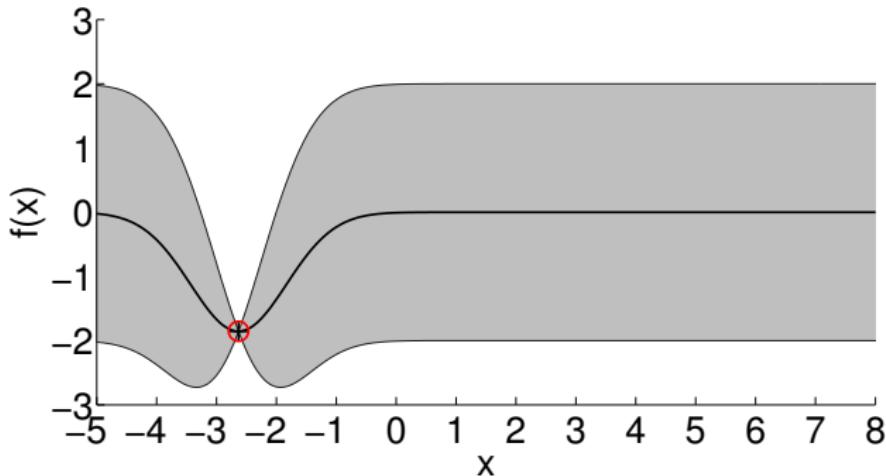
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*) | \mathbf{x}_*, \emptyset] = m(\mathbf{x}_*) = 0$$

$$\mathbb{V}[f(\mathbf{x}_*) | \mathbf{x}_*, \emptyset] = \sigma^2(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

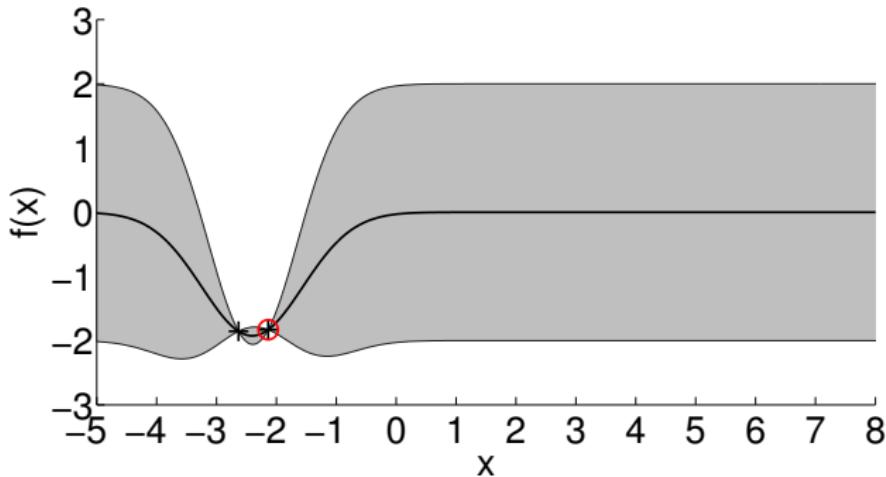
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

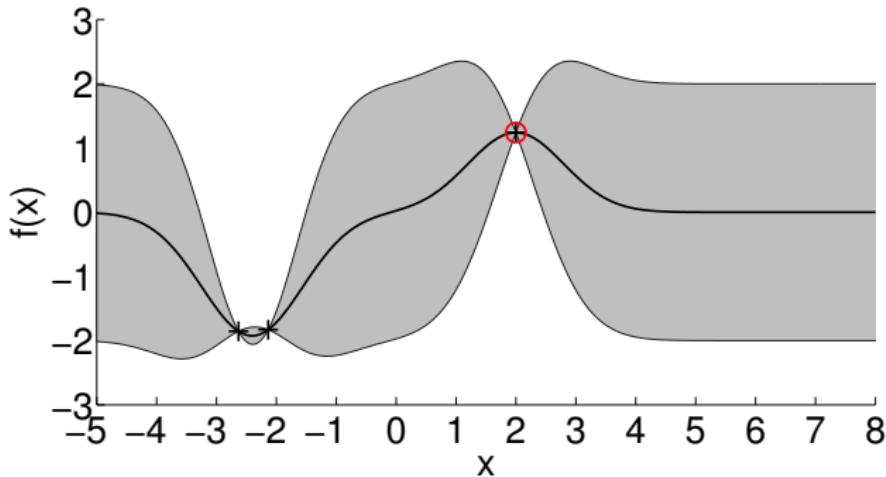
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

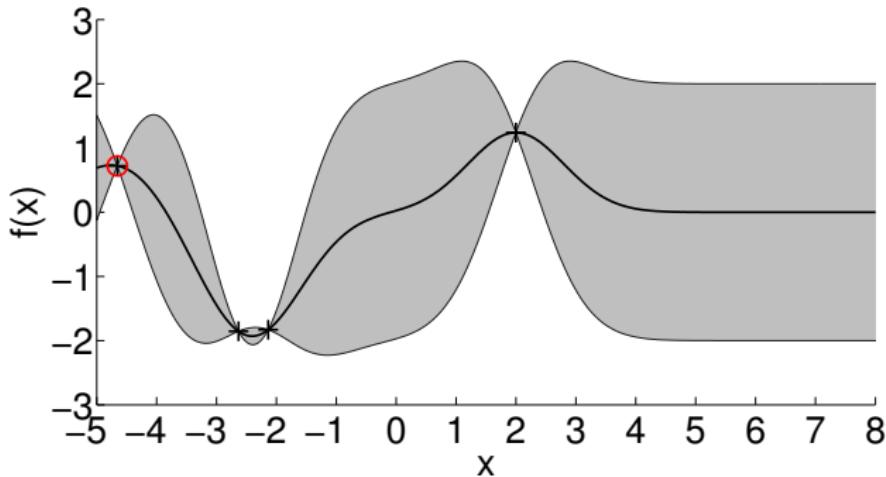
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

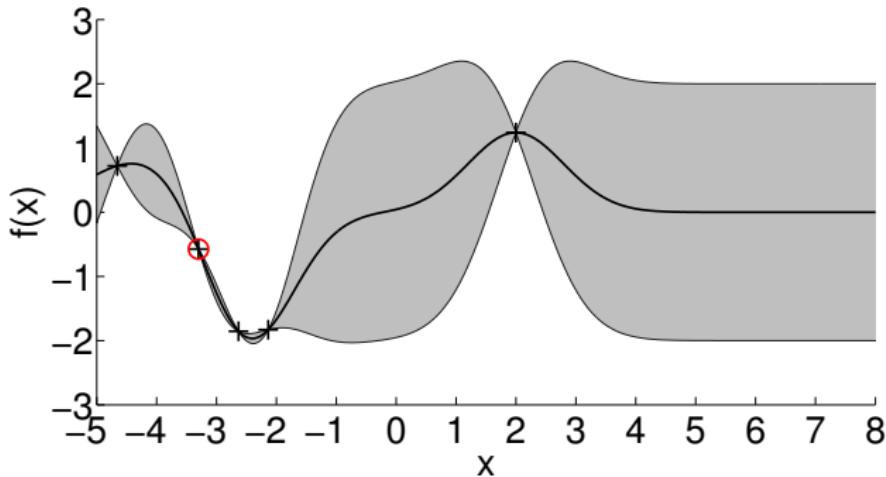
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

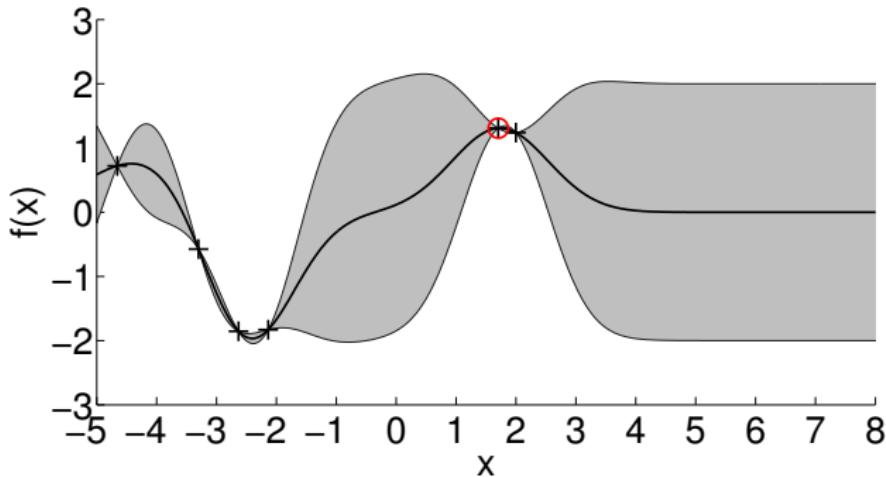
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

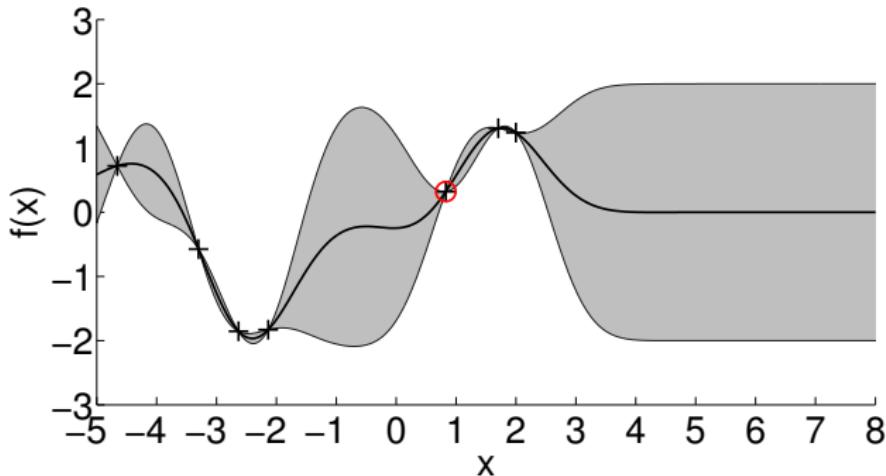
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL



Posterior belief about the function

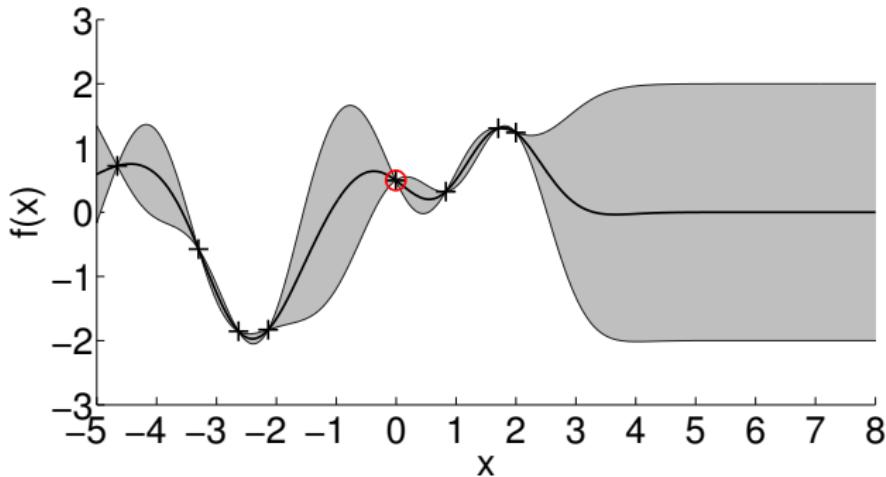
Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Illustration: Inference with Gaussian Processes

UCL

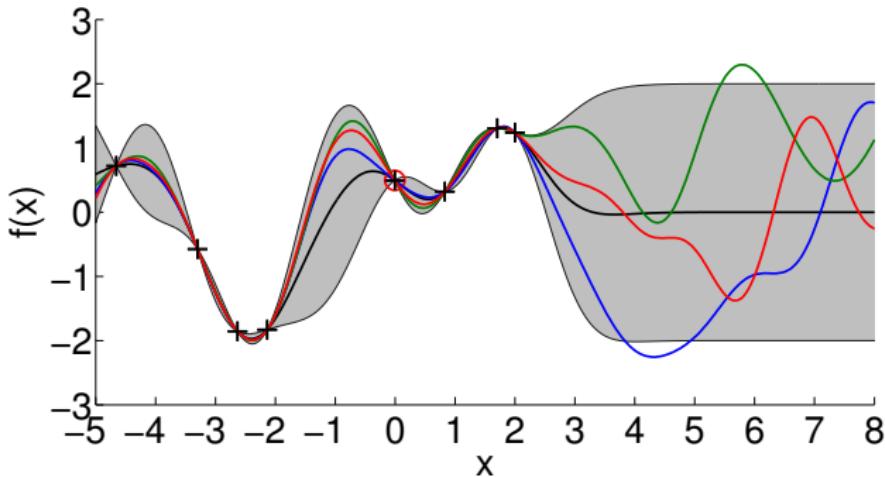


Posterior belief about the function

Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$



Posterior belief about the function

Predictive (marginal) mean and variance:

$$\mathbb{E}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = m(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbb{V}[f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*)$$

# Overview

1 Gaussian Process: Definition

2 Regression as Inference

- GP Prior

- Likelihood

- Marginal Likelihood

- Posterior

- Predictions

3 Model Selection

- GP Training

- Hyper-Parameters

- Inspection of the Marginal Likelihood

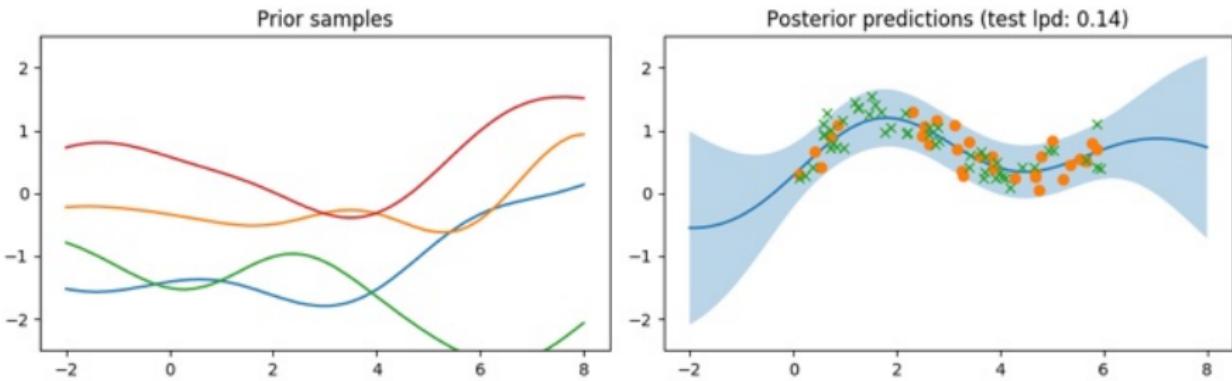
- Covariance Function

4 Limitations and Guidelines

5 Application Areas

# Model Selection

# Influence of Prior on Posterior

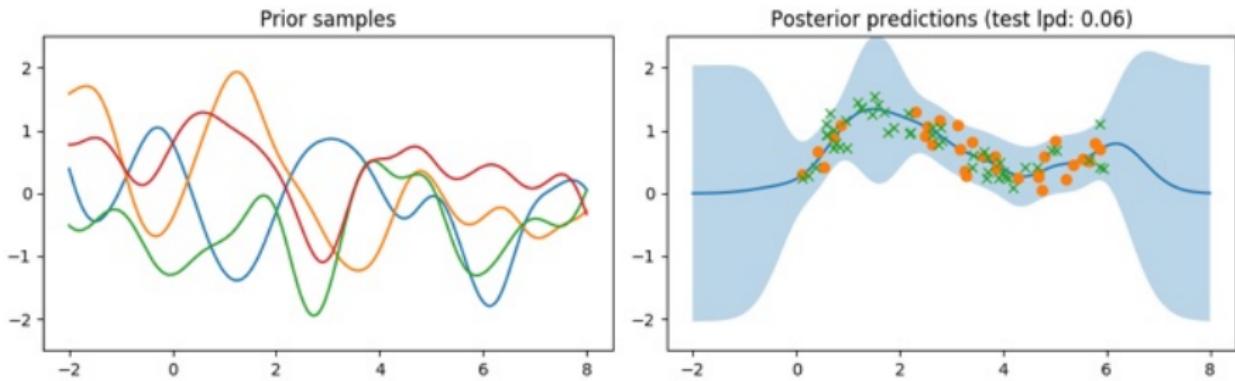


- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

# Influence of Prior on Posterior



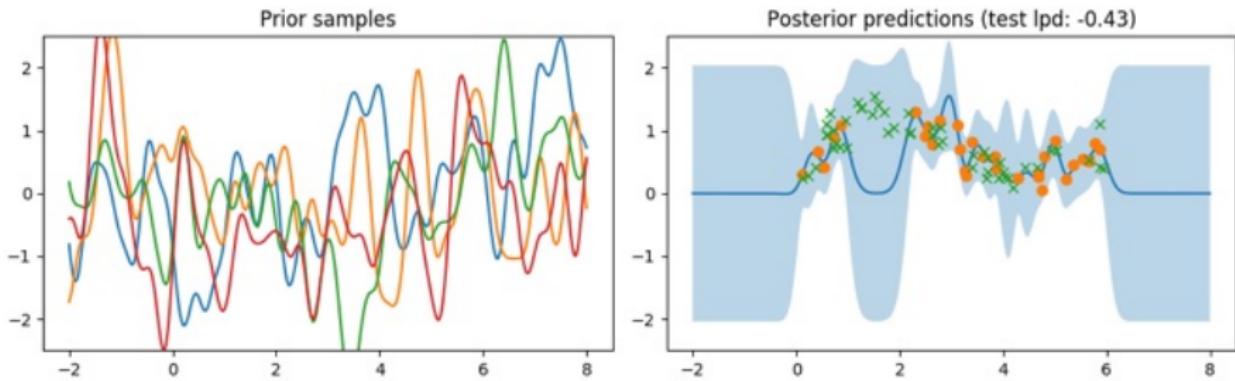
- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



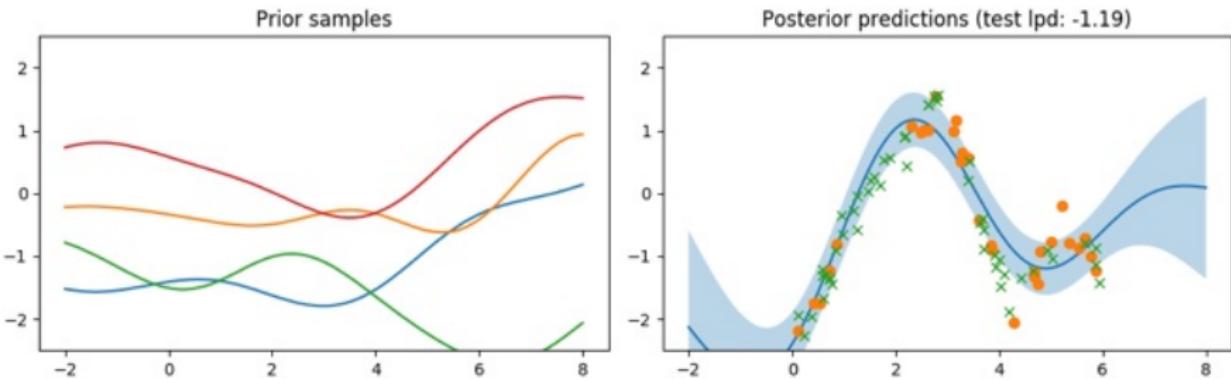
- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



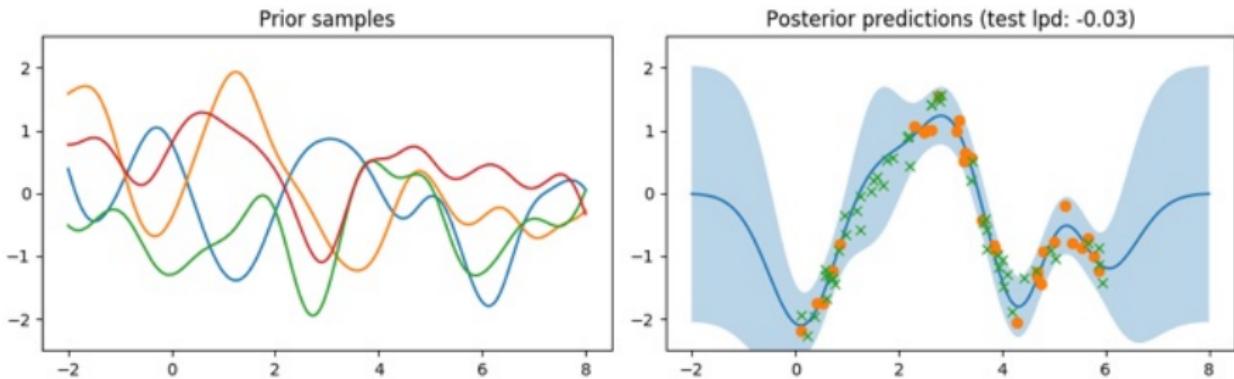
- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



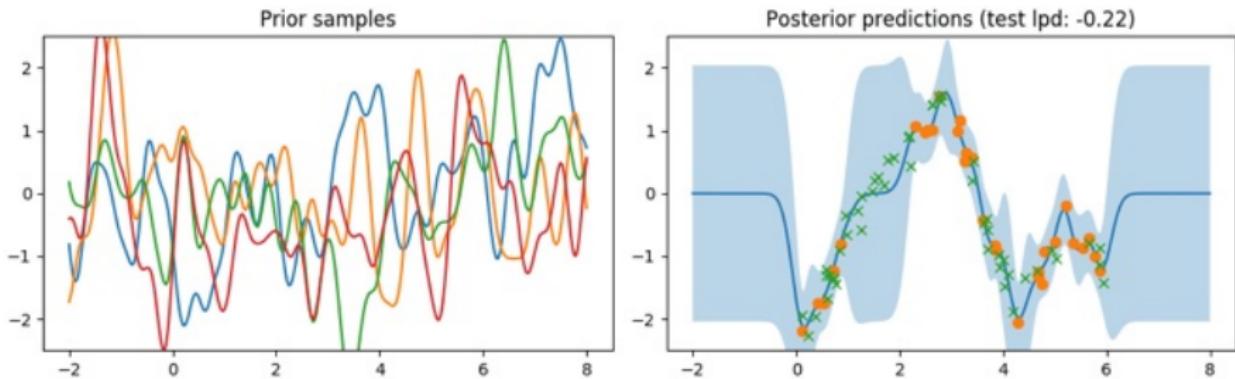
- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



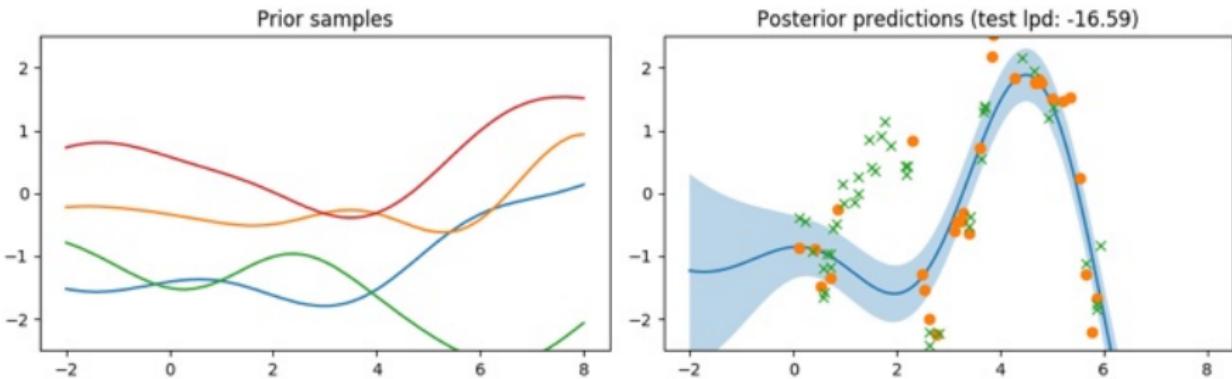
- Generalization error measured by **log-predictive density** (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



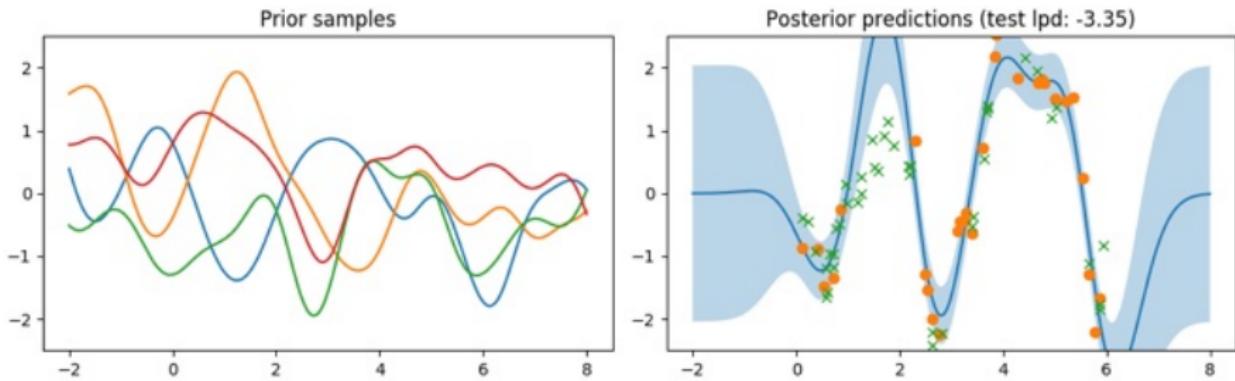
- Generalization error measured by **log-predictive density** (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



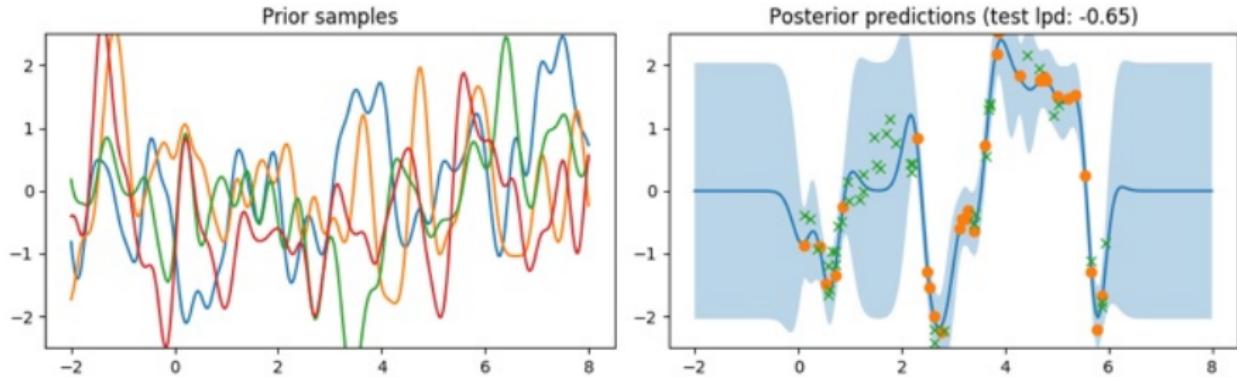
- Generalization error measured by log-predictive density (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

# Influence of Prior on Posterior



- Generalization error measured by **log-predictive density** (lpd)

$$\text{lpd} = \log p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}, \ell)$$

for different length-scales  $\ell$  and different datasets

- Shorter length-scale ➤ More flexible model ➤ Faster increase in uncertainty away from data ➤ Bad generalization properties

- Make predictions equipped with uncertainty
- Choice of prior (e.g., length-scale) influences predictions
- Different tasks require different priors

## How do we select a good prior?

### Model Selection in GPs

- ▶ Choose hyper-parameters of the GP
- ▶ Choose good mean function and kernel

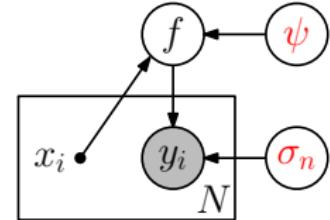
The GP possesses a set of **hyper-parameters**:

- Parameters of the mean function
  - Parameters of the covariance function (e.g., length-scales and signal variance)
  - Likelihood parameters (e.g., noise variance  $\sigma_n^2$ )
- Train a GP to find a good set of hyper-parameters
- Higher-level **model selection** to find good mean and covariance functions  
(can also be automated: Automatic Statistician (Lloyd et al., 2014))

## GP Training

Find good hyper-parameters  $\theta$  (kernel/mean function parameters  $\psi$ , noise variance  $\sigma_n^2$ )

- Place a prior  $p(\theta)$  on hyper-parameters
- Posterior over hyper-parameters:



$$p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y}) = \frac{p(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})}{p(\mathbf{y}|\mathbf{X})}$$

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{y}|f, \mathbf{X}) p(f|\mathbf{X}, \boldsymbol{\theta}) df$$

- Posterior over hyper-parameters:

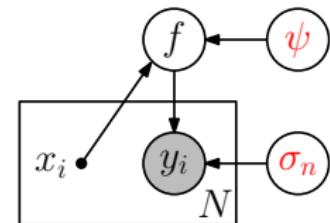
$$p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y}) = \frac{p(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})}{p(\mathbf{y}|\mathbf{X})}$$

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{y}|f(\mathbf{X})) p(f(\mathbf{X})|\boldsymbol{\theta}) df$$

- Choose hyper-parameters  $\boldsymbol{\theta}^*$ , such that

$$\boldsymbol{\theta}^* \in \arg \max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}) + \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

- ▶ Maximize marginal likelihood if  $p(\boldsymbol{\theta}) = \mathcal{U}$  (uniform prior)



## GP Training

Maximize the evidence/marginal likelihood (probability of the data given the hyper-parameters, where the unwieldy  $f$  has been integrated out) ➤ Also called Maximum Likelihood Type-II

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

$$\begin{aligned} p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) &= \int p(\mathbf{y}|f(\mathbf{X})) p(f(\mathbf{X})|\boldsymbol{\theta}) df \\ &= \int \mathcal{N}(\mathbf{y} | f(\mathbf{X}), \sigma_n^2 \mathbf{I}) \mathcal{N}(f(\mathbf{X}) | \mathbf{0}, \mathbf{K}) df \\ &= \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K} + \sigma_n^2 \mathbf{I}) \end{aligned}$$

Learning the GP hyper-parameters:

$$\boldsymbol{\theta}^* \in \arg \max_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

- Log-marginal likelihood:

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

$$\mathbf{K}_{\boldsymbol{\theta}} := \mathbf{K} + \sigma_n^2 \mathbf{I}$$

- Gradient-based optimization to get hyper-parameters  $\boldsymbol{\theta}^*$ :

$$\begin{aligned} \frac{\partial \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})}{\partial \theta_i} &= \frac{1}{2} \mathbf{y}^\top \mathbf{K}_{\boldsymbol{\theta}}^{-1} \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \theta_i} \mathbf{K}_{\boldsymbol{\theta}}^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_{\boldsymbol{\theta}}^{-1} \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \theta_i}) \\ &= \frac{1}{2} \text{tr}((\boldsymbol{\alpha} \boldsymbol{\alpha}^\top - \mathbf{K}_{\boldsymbol{\theta}}^{-1}) \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \theta_i}), \\ \boldsymbol{\alpha} &:= \mathbf{K}_{\boldsymbol{\theta}}^{-1} \mathbf{y} \end{aligned}$$

- “ELBO” refers to the log-marginal likelihood
- Data-fit term gets worse, but marginal likelihood increases

---

<sup>1</sup>Thanks to Mark van der Wilk

Log-marginal likelihood:

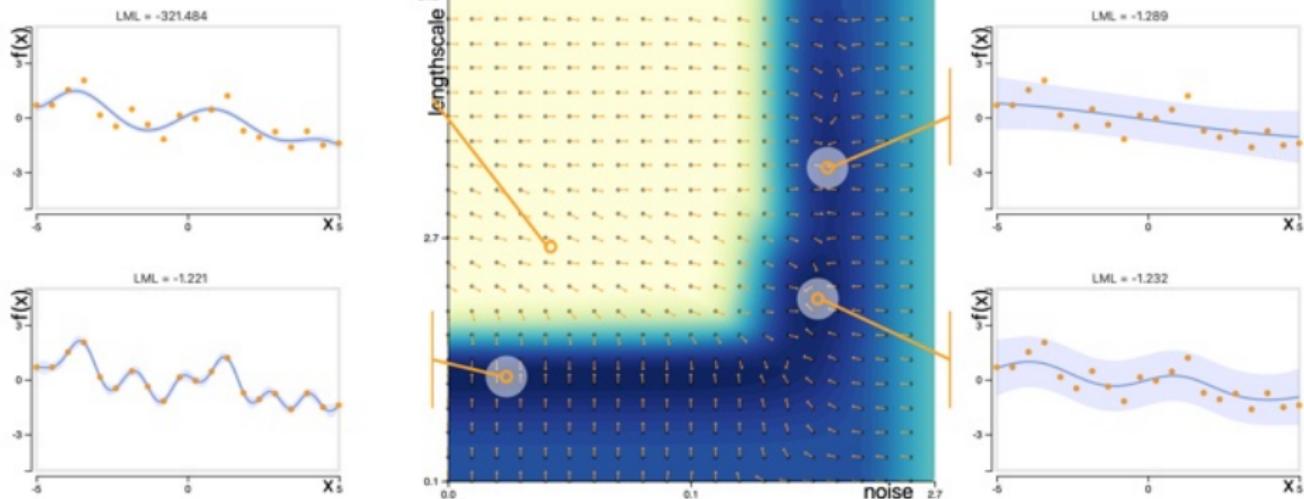
$$\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^\top \mathbf{K}_{\boldsymbol{\theta}}^{-1}\mathbf{y} - \frac{1}{2}\log |\mathbf{K}_{\boldsymbol{\theta}}| + \text{const}, \quad \mathbf{K}_{\boldsymbol{\theta}} := \mathbf{K} + \sigma_n^2 \mathbf{I}$$

- Quadratic term measures whether observation  $\mathbf{y}$  is within the variation allowed by the prior
- Determinant is the product of the variances of the prior (volume of the prior) ➤ Volume  $\approx$  richness of model class

## Marginal likelihood

➤ Automatic trade-off between data fit and model complexity

# Marginal Likelihood Surface



- Several plausible hyper-parameters (local optima)
- What do you expect to happen in each local optimum?

- The marginal likelihood is **non-convex**
- Especially in the very-small-data regime, a GP can end up in **three different situations** when optimizing the hyper-parameters:
  - Short length-scales, low noise (highly nonlinear mean function with little noise)
  - Long length-scales, high noise (everything is considered noise)
  - Hybrid
- **Re-start** hyper-parameter optimization from random initialization to mitigate the problem
- With increasing data set size the GP typically ends up in the “hybrid” mode. Other modes are unlikely.
- Ideally, we would integrate the hyper-parameters out  
**No closed-form solution** ➤ Markov chain Monte Carlo

- Overall goal: Good generalization performance on unseen test data
- Minimizing training error is not a good idea (e.g., maximum likelihood) ► Overfitting
- Just adding uncertainty does not help either if the model is wrong, but it makes predictions more cautious
- Marginal likelihood seems to find a good balance between fitting the data and finding a simple model (Occam's razor)

Why does the marginal likelihood lead to models that generalize well?

- “Probability of the training data” given the parameters
- General factorization (ignoring inputs  $X$ ):

$$p(\mathbf{y}|\boldsymbol{\theta}) = p(y_1, \dots, y_N|\boldsymbol{\theta})$$

$$= p(y_1|\boldsymbol{\theta})p(y_2|y_1, \boldsymbol{\theta})p(y_3|y_1, y_2, \boldsymbol{\theta}) \cdot \dots \cdot p(y_N|y_1, \dots, y_{N-1}, \boldsymbol{\theta})$$

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

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

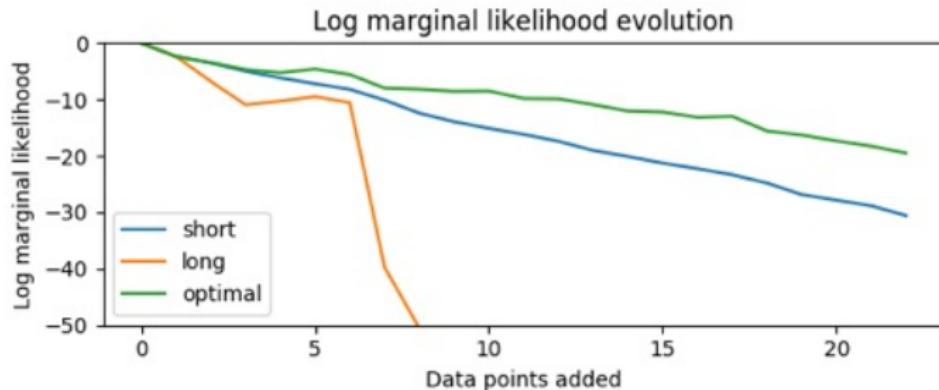
- If we think of this as a sequence model (where data arrives sequentially), the marginal likelihood predicts the  $n$ th training observation given all “previous” observations
- Predict training data  $y_n$  that has not been accounted for (we only condition on  $y_1, \dots, y_{n-1}$ ) ➤ Treat next data point as test data
- Intuition: If it continuously predicted well on all  $N$  previous points, it probably will do well next time
  - Proxy for generalization error on unseen test data

$$p(\mathbf{y}|\boldsymbol{\theta}) = p(y_1, \dots, y_N|\boldsymbol{\theta}) = p(y_1|\boldsymbol{\theta}) \prod_{n=2}^N p(y_n|y_1, \dots, y_{n-1}, \boldsymbol{\theta})$$

- Optimal length-scale

---

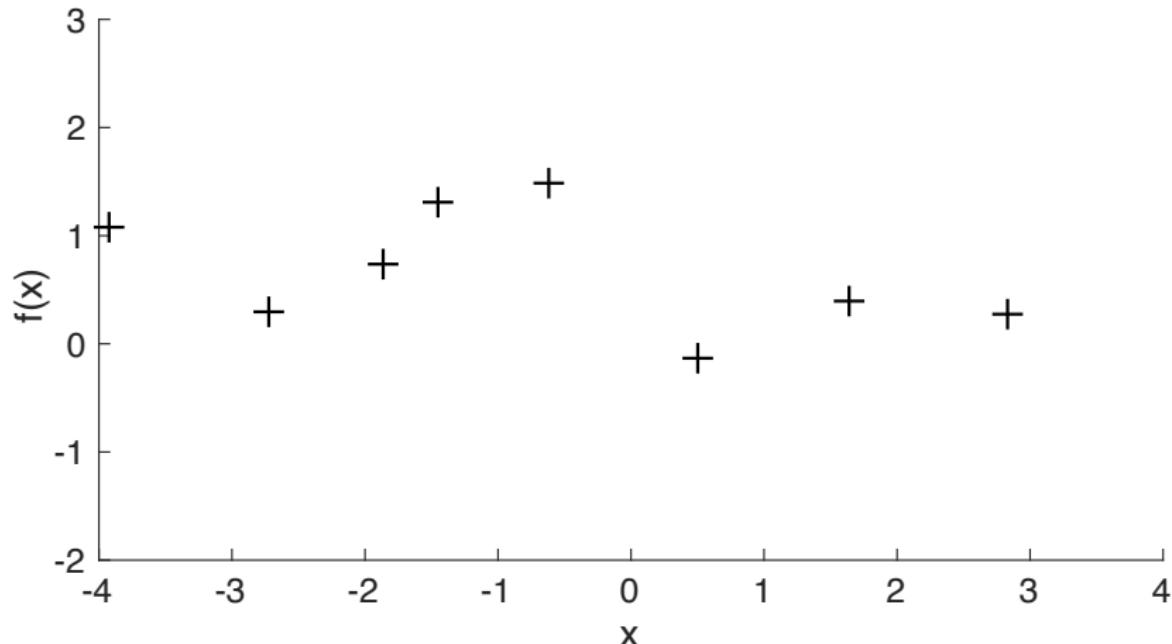
<sup>4</sup>Thanks to Mark van der Wilk



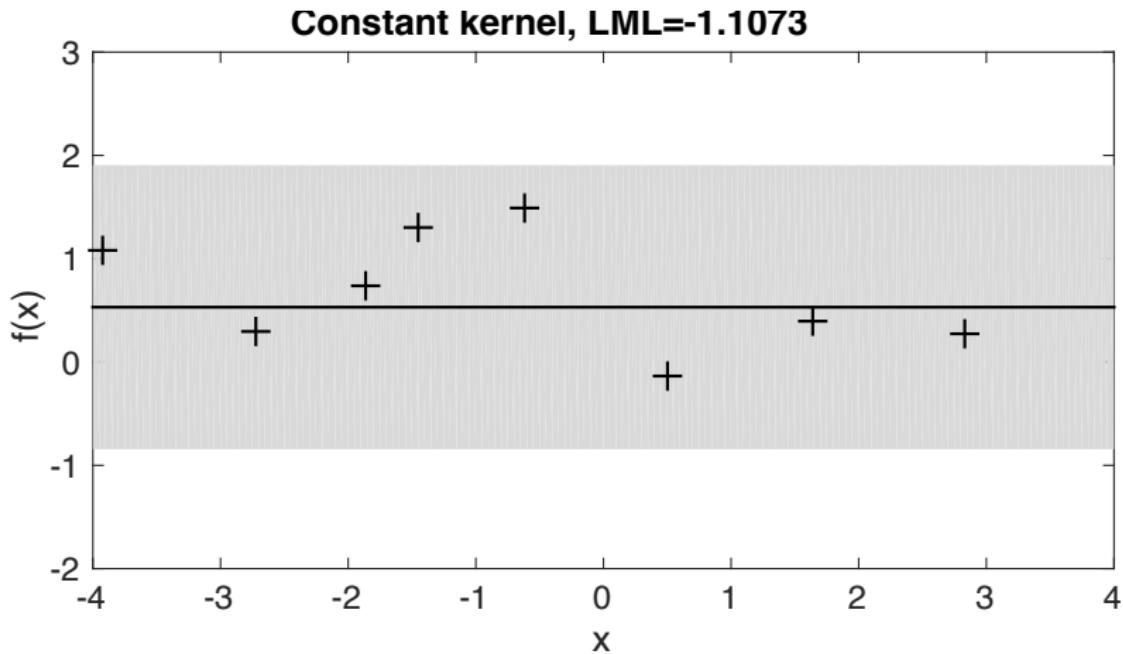
- Short lengthscale: consistently **overestimates variance**
  - ▶ No high density, even with observations inside the error bars
- Long lengthscale: consistently **underestimates variance**
  - ▶ Low density because observations are outside the error bars
- Optimal lengthscale: **trades off both behaviors reasonably well**

- Assume we have a finite set of models  $M_i$ , each one specifying a mean function  $m_i$  and a kernel  $k_i$ . How do we find the best one?
- Some options:
  - Cross validation
  - Bayesian Information Criterion, Akaike Information Criterion
  - Compare marginal likelihood values (assuming a uniform prior on the set of models)

# Example

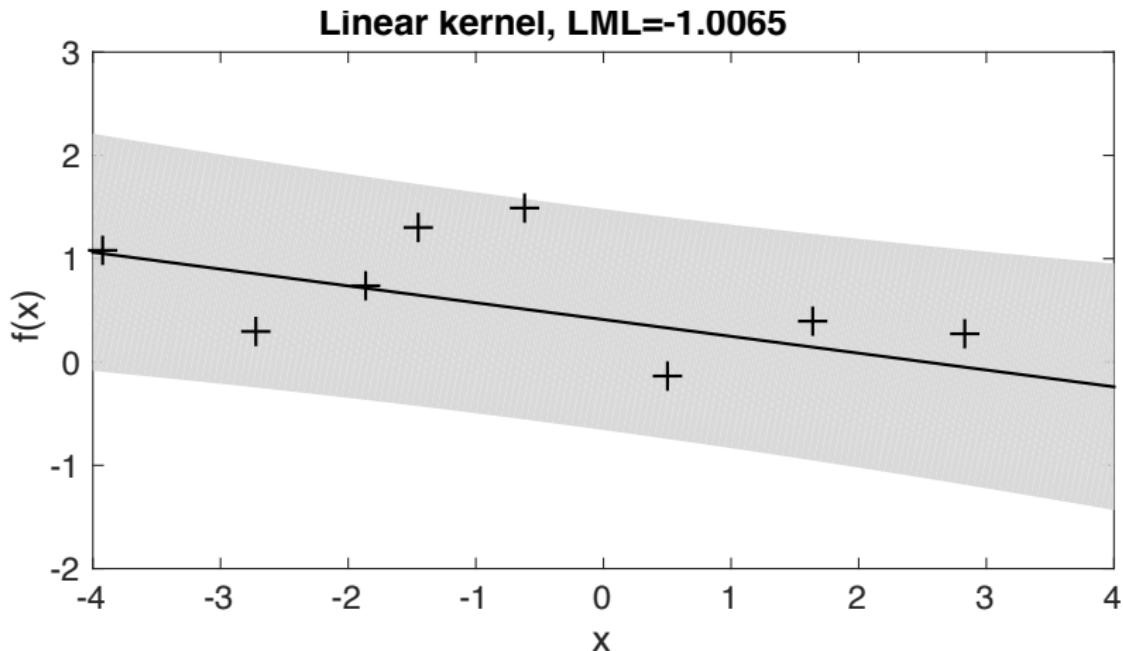


- Four different kernels (mean function fixed to  $m \equiv 0$ )
- MAP hyper-parameters for each kernel
- Log-marginal likelihood values for each (optimized) model

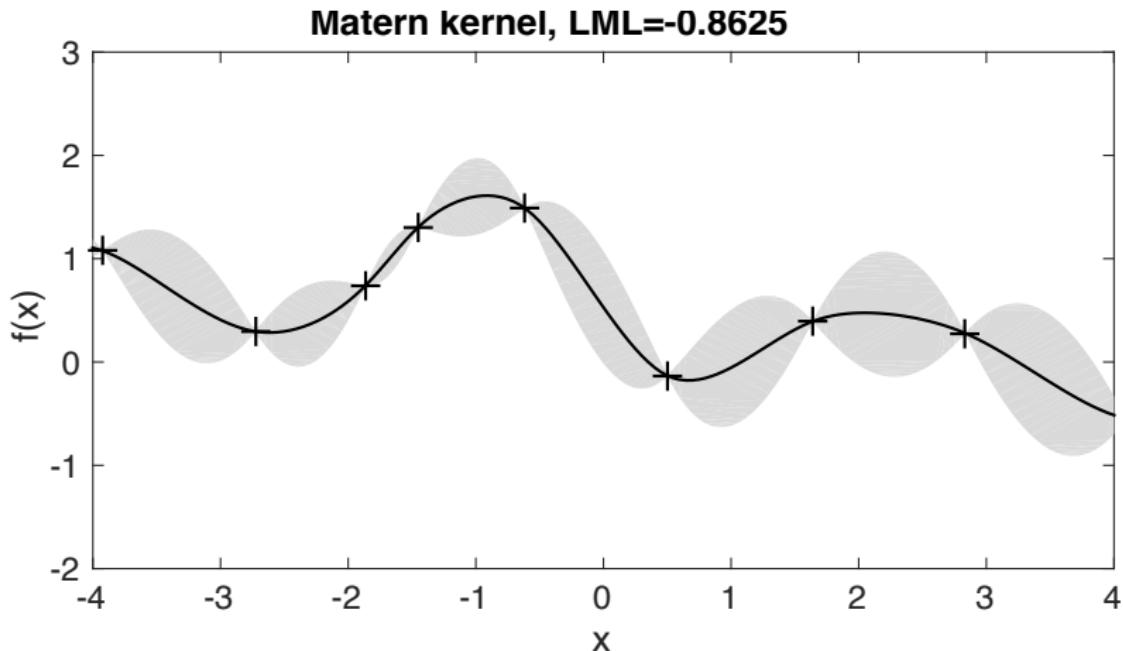


- Four different kernels (mean function fixed to  $m \equiv 0$ )
- MAP hyper-parameters for each kernel
- Log-marginal likelihood values for each (optimized) model

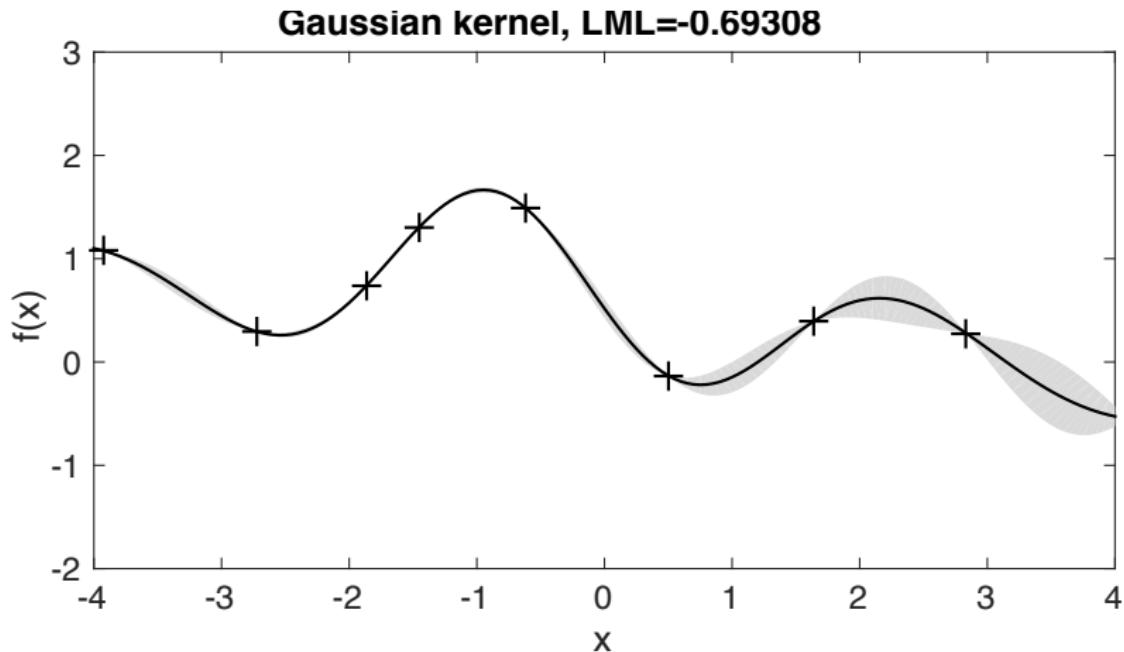
# Example



- Four different kernels (mean function fixed to  $m \equiv 0$ )
- MAP hyper-parameters for each kernel
- Log-marginal likelihood values for each (optimized) model



- Four different kernels (mean function fixed to  $m \equiv 0$ )
- MAP hyper-parameters for each kernel
- Log-marginal likelihood values for each (optimized) model



- Four different kernels (mean function fixed to  $m \equiv 0$ )
- MAP hyper-parameters for each kernel
- Log-marginal likelihood values for each (optimized) model

- Prior:  $f(\mathbf{x}) = \theta_s f_{\text{smooth}}(\mathbf{x}) + \theta_p f_{\text{periodic}}(\mathbf{x})$ , with smooth and periodic GP priors, respectively.
- Amount of periodicity vs. smoothness is automatically chosen by selecting hyper-parameters  $\theta_s, \theta_p$ .
- Marginal likelihood learns how to generalize, not just to fit the data

---

<sup>5</sup>Thanks to Mark van der Wilk

## Limitations and Guidelines

## Computational and memory complexity

Training set size:  $N$

- Training scales in  $\mathcal{O}(N^3)$
- Prediction (variances) scales in  $\mathcal{O}(N^2)$
- Memory requirement:  $\mathcal{O}(ND + N^2)$

► **Practical limit**  $N \approx 10,000$

Some solution approaches:

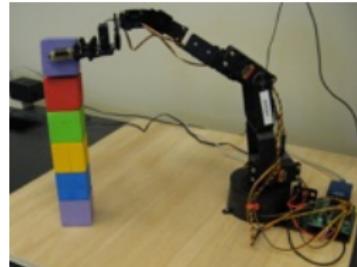
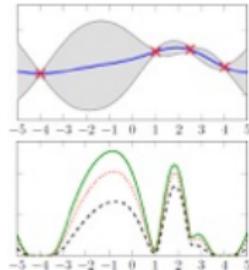
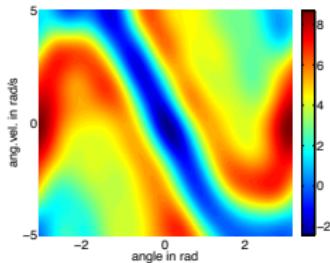
- Sparse GPs with **inducing variables** (e.g., Snelson & Ghahramani, 2006; Quiñonero-Candela & Rasmussen, 2005; Titsias 2009; Hensman et al., 2013; Matthews et al., 2016)
- Combination of **local GP expert models** (e.g., Tresp 2000; Cao & Fleet 2014; Deisenroth & Ng, 2015)
- **Variational Fourier features** (Hensman et al., 2018)

- To set initial hyper-parameters, use domain knowledge.
- Standardize input data and set initial length-scales  $\ell$  to  $\approx 0.5$ .
- 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 are advised.
- Mitigate the problem of numerical instability (Cholesky decomposition of  $\mathbf{K} + \sigma_n^2 \mathbf{I}$ ) by penalizing high signal-to-noise ratios  $\sigma_f/\sigma_n$

► <https://drafts.distill.pub/gp>

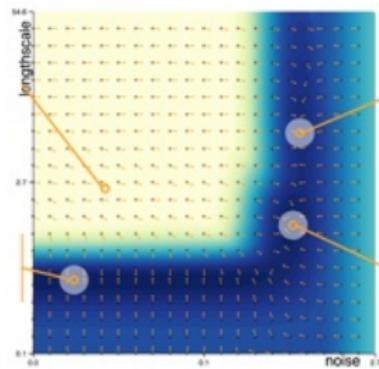
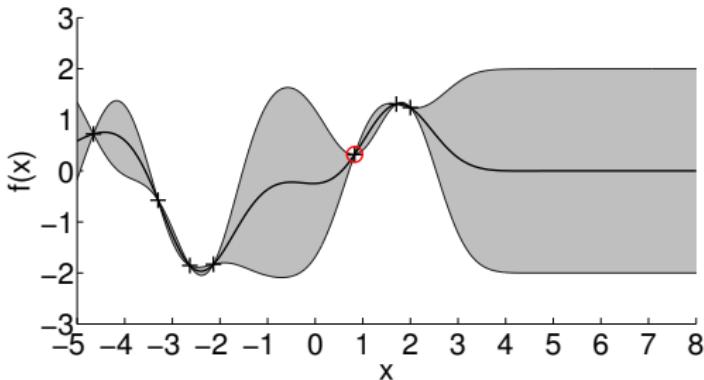
## Application Areas

# Application Areas



- Reinforcement learning and robotics
  - ▶ Model value functions and/or dynamics with GPs
- Bayesian optimization (Experimental Design)
  - ▶ Model unknown utility functions with GPs
- Geostatistics
  - ▶ Spatial modeling (e.g., landscapes, resources)
- Sensor networks
- Time-series modeling and forecasting

# Summary



- Gaussian processes are the **gold-standard** for regression
- Closely related to Bayesian linear regression
- Computations boil down to **manipulating multivariate Gaussian distributions**
- Marginal likelihood objective automatically trades off data fit and model complexity

# Bayesian Optimization

Marc Deisenroth

Centre for Artificial Intelligence

Department of Computer Science

University College London

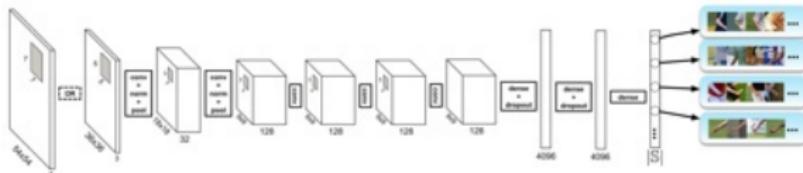
 @mpd37

m.deisenroth@ucl.ac.uk

<https://deisenroth.cc>

- Machine learning models are getting more and more complicated
  - ▶ Usually more parameters (e.g., deep neural networks)
- Non-convex and stochastic optimization methods have meta-parameters that are difficult to tune (learning rates, momentum parameters, ...)
  - ▶ Generally hard to apply modern techniques or reproduce results

# Example: Deep Neural Networks



Huge interest in large neural networks

- When well-tuned, very successful for visual object identification, speech recognition, computational biology, ...
- Huge investments by Google, Facebook, Microsoft, etc.
- **Many choices:** number of layers, weight regularization, layer size, which nonlinearity, batch size, learning rate schedule, stopping conditions

# Example: Online Latent Dirichlet Allocation

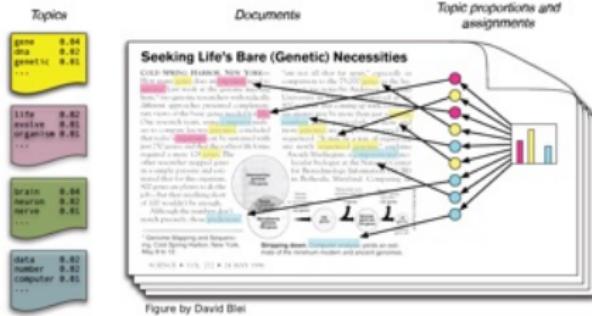
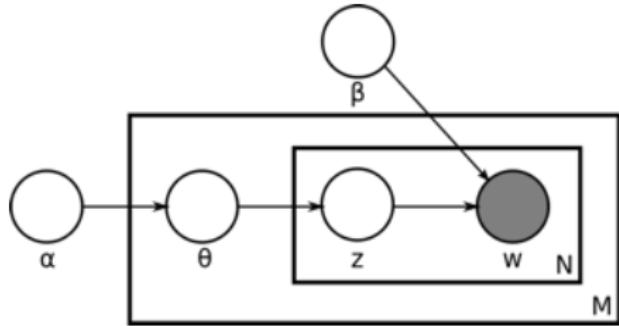
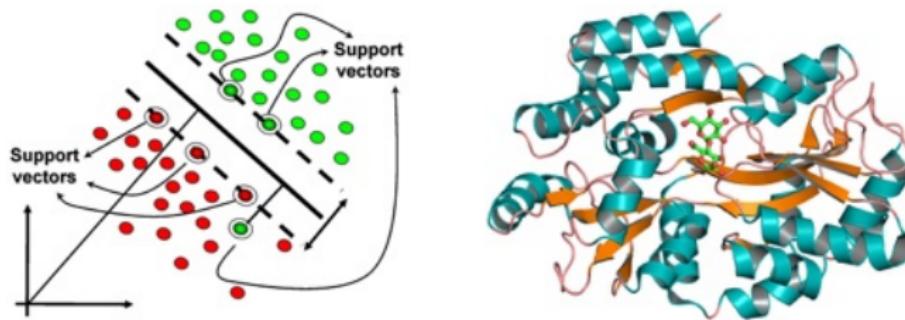


Figure by David Blei



- Hoffman et al. (2010): Approximate inference for **large-scale text analysis (topic modeling)** with Latent Dirichlet Allocation
- Good empirical results when well tuned
- **Hyper-parameters** tricky to set: Dirichlet parameters, number of topics, learning rate schedule, batch size, vocabulary size, ...

# Example: Classification of DNA Sequences



- Objective: Predict which DNA sequences will bind with which proteins
- Miller et al. (2012): [Latent Structural Support Vector Machine](#)
- **Hyper-parameters:** margin/slack parameter, entropy parameter, convergence criterion

- Define an objective function to evaluate the quality of the hyper-parameters
  - Usually, we care about generalization performance
  - Cross validation to measure parameter quality
- Standard search procedures:
  - Manual tuning
  - Grid search
  - Random search (very simple, works surprisingly well)
  - Black magic
- Painful:
  - Evaluating the quality of the objective may be very expensive (e.g., time or money)
    - ▶ Imagine we would need to run a GPU/TPU cluster for 2 weeks
  - Many training cycles
  - Possibly noisy

## Setting

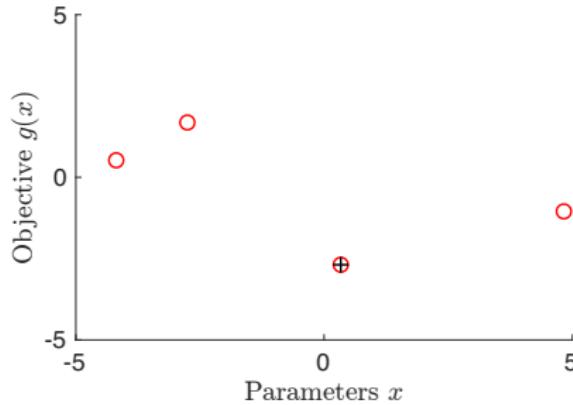
Globally optimize a black-box objective that is expensive to evaluate  
(e.g., cross-validation error for a massive neural network)

- Build a **probabilistic proxy model** for the objective using outcomes of past experiments as training data
- The proxy model is much **cheaper to evaluate** than the original objective
- **Optimize cheap proxy** function to determine where to evaluate the true objective next
- Standard proxy: **Gaussian process**

- Objective: Find global minimum of objective function  $g$ :

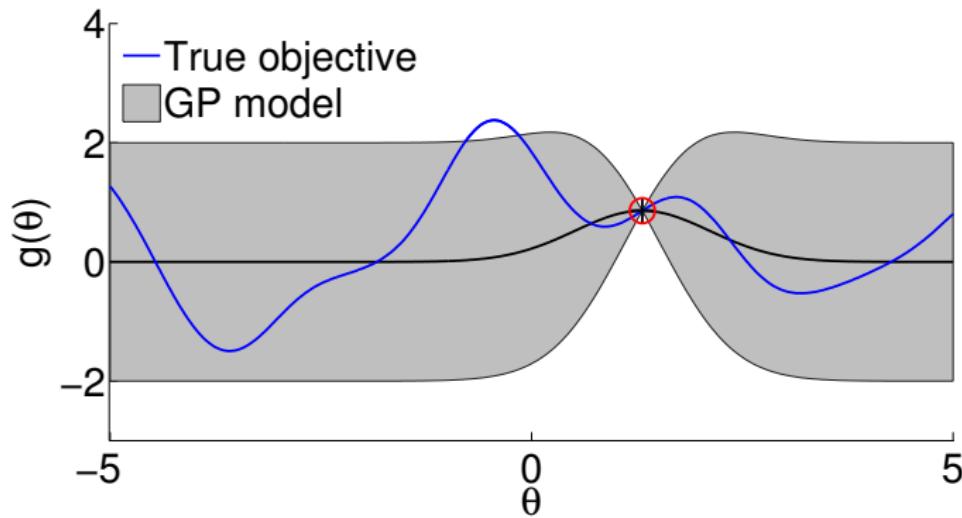
$$\mathbf{x}_* = \arg \min_{\mathbf{x}} g(\mathbf{x})$$

- We can evaluate the objective  $g$  pointwise, but do not have an easy functional form or gradients; observations may be noisy
- Evaluating  $g$  is costly (e.g., train a massive deep network)

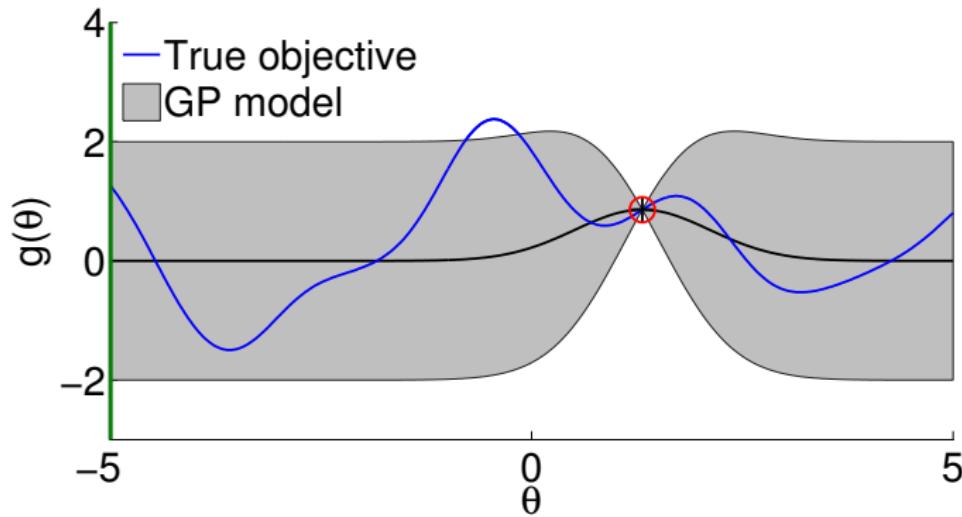


- To avoid evaluating  $g$  an excessive number of times, approximate it using a proxy function  $\tilde{g}$  (which is cheap to evaluate)
- Find a global optimum  $\tilde{g}(x_*)$  of proxy function  $\tilde{g}$
- Evaluate true objective  $g$  at  $x_*$
- Overall: Evaluate  $g$  only once
- Works well if  $\tilde{g} \approx g$ .
- Usually not the case ➤ Repeat this cycle and keep updating  $\tilde{g}$

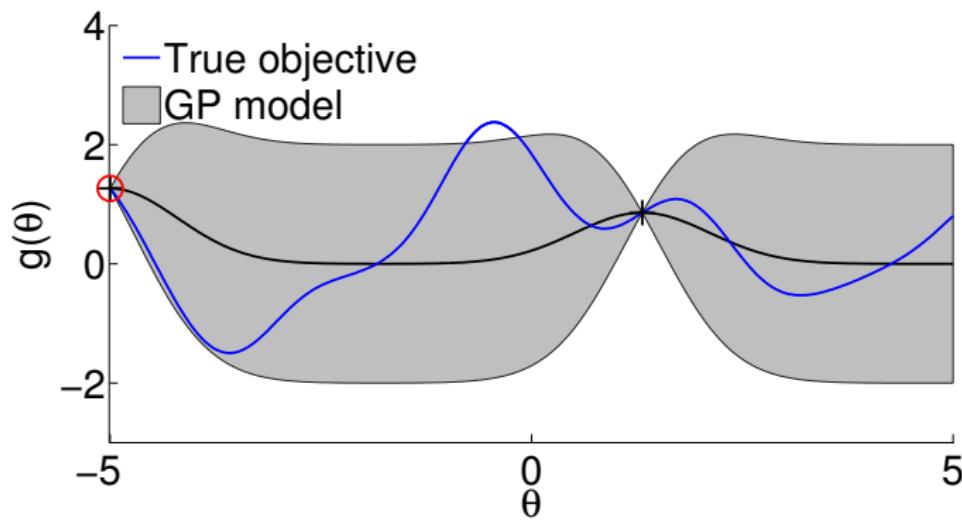
# Bayesian Optimization: Illustration



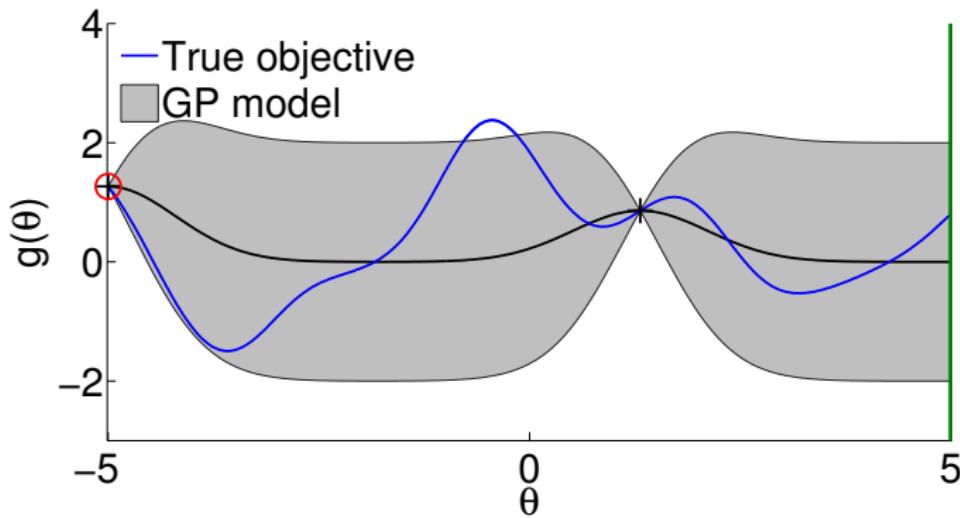
# Bayesian Optimization: Illustration



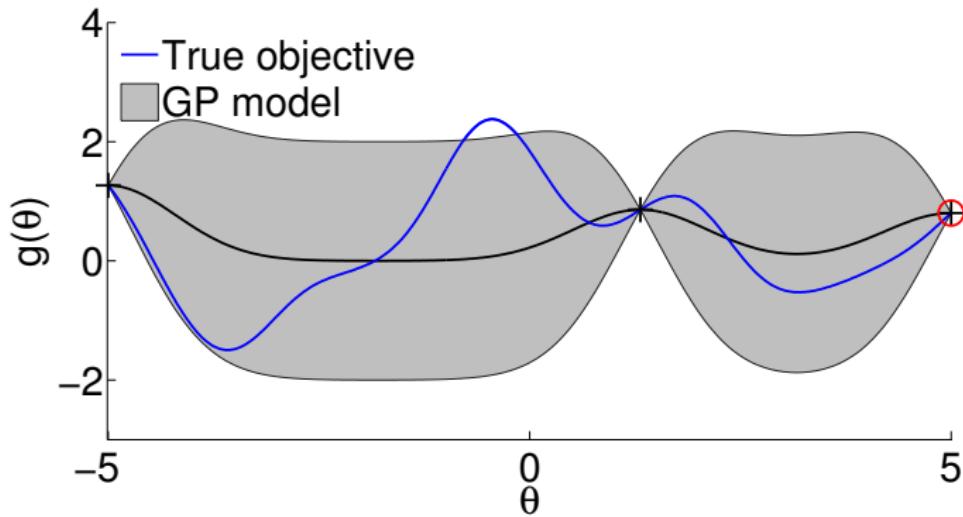
# Bayesian Optimization: Illustration



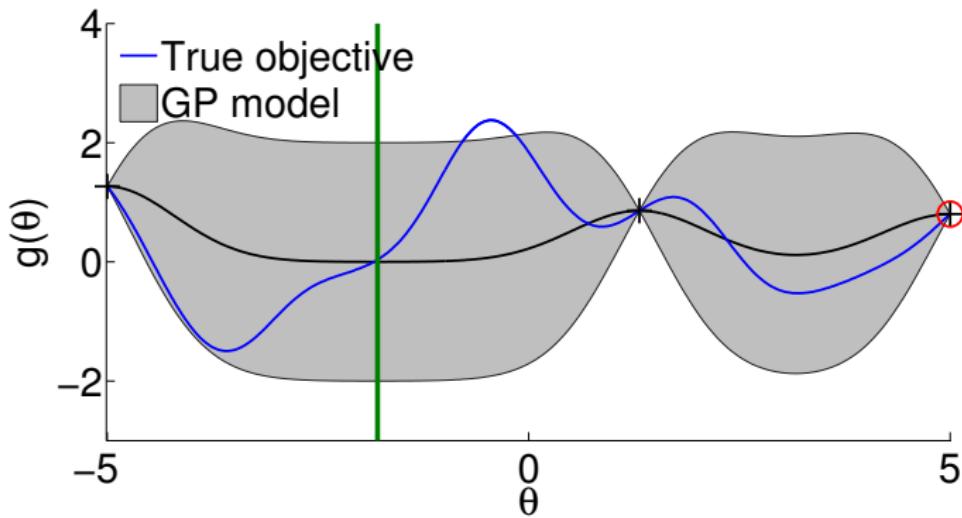
# Bayesian Optimization: Illustration



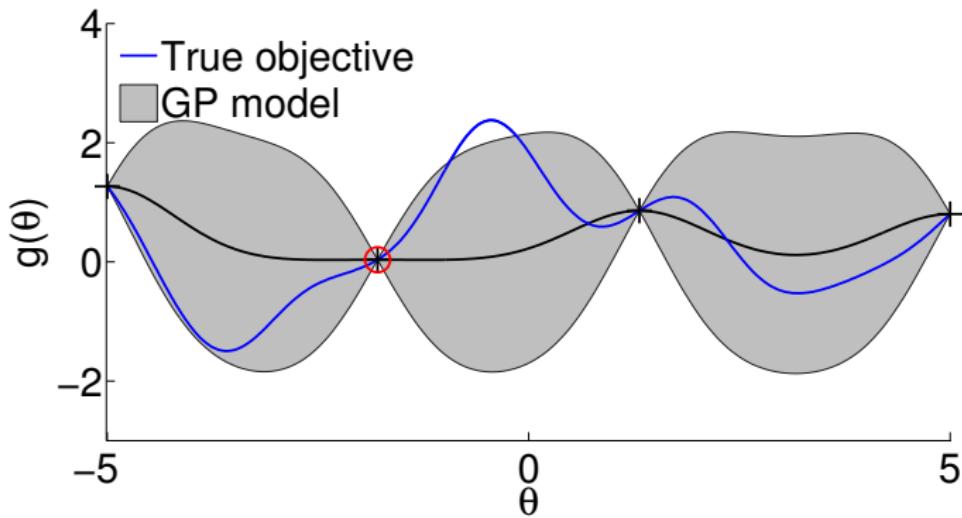
# Bayesian Optimization: Illustration



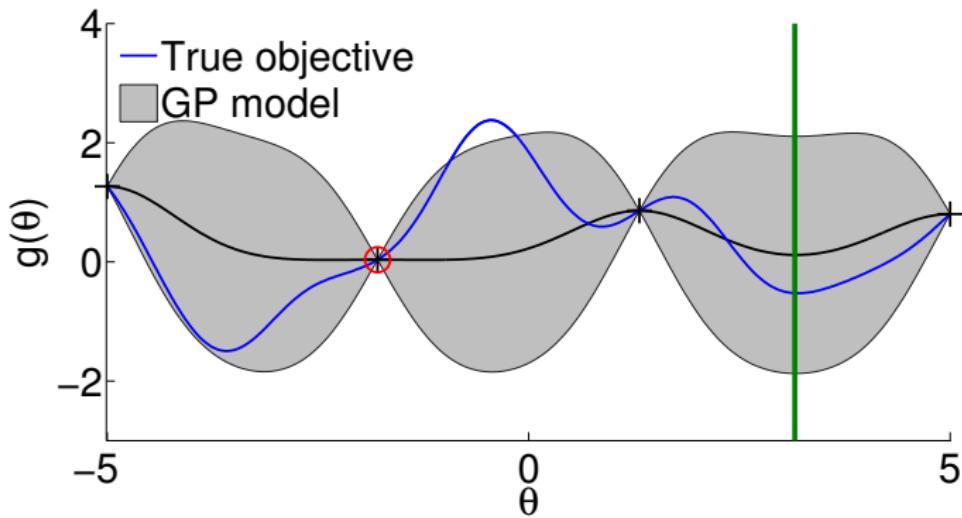
# Bayesian Optimization: Illustration



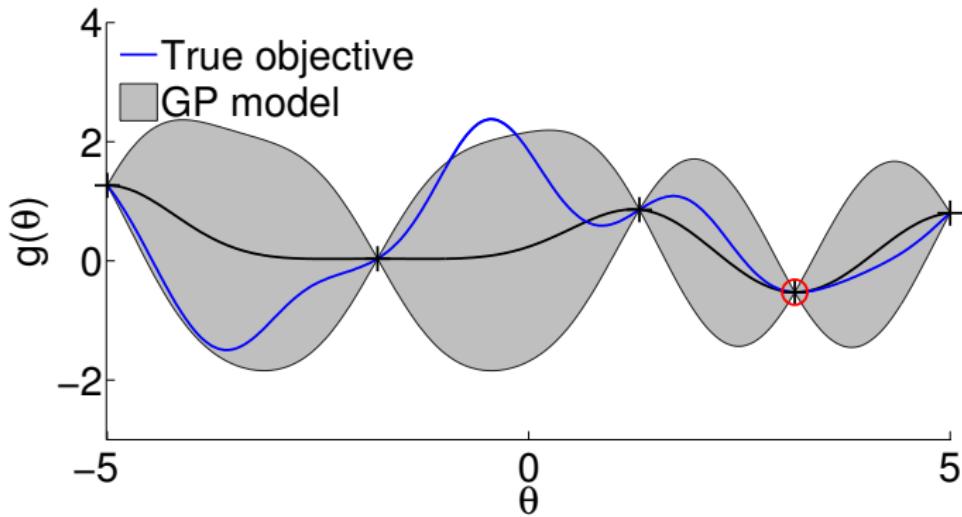
# Bayesian Optimization: Illustration



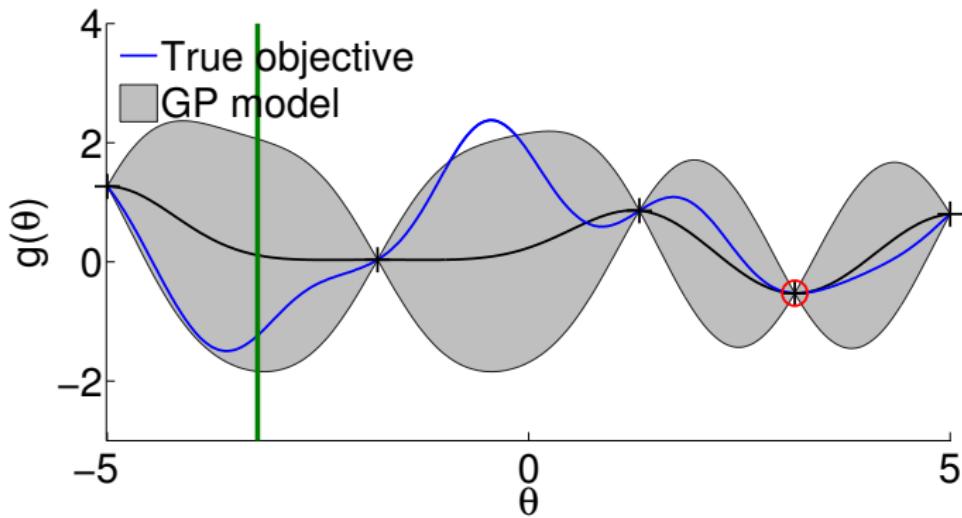
# Bayesian Optimization: Illustration



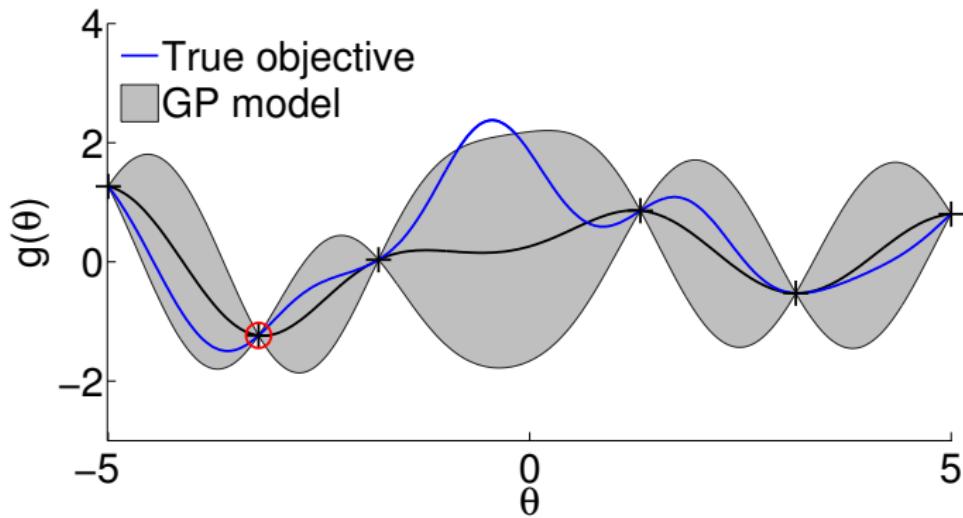
# Bayesian Optimization: Illustration



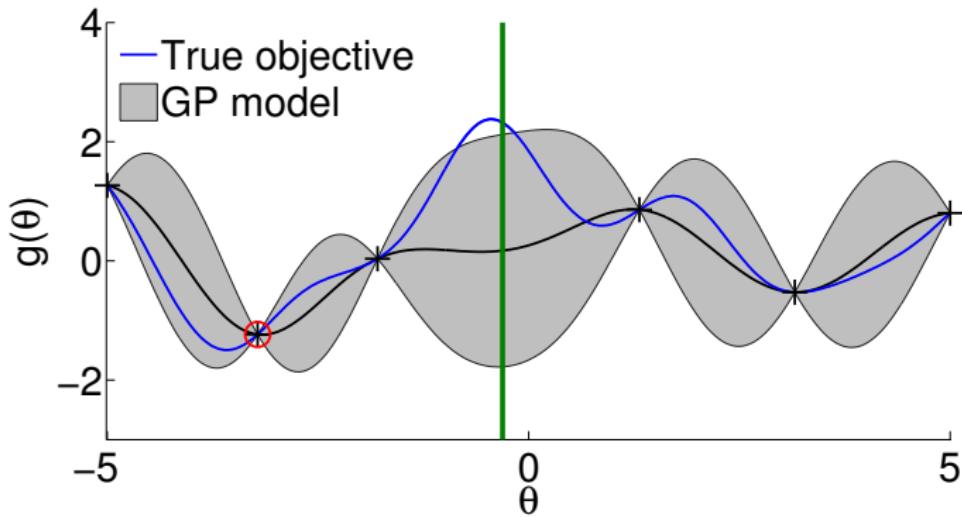
# Bayesian Optimization: Illustration



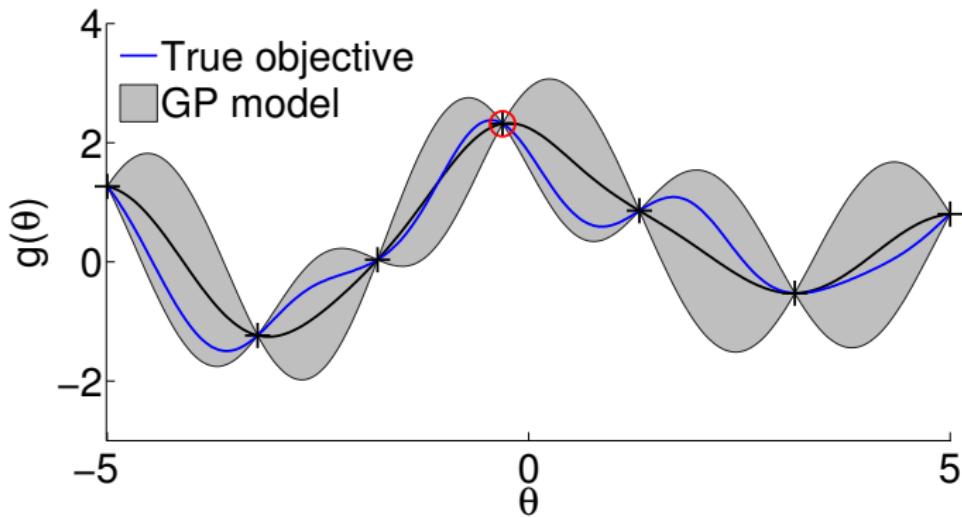
# Bayesian Optimization: Illustration



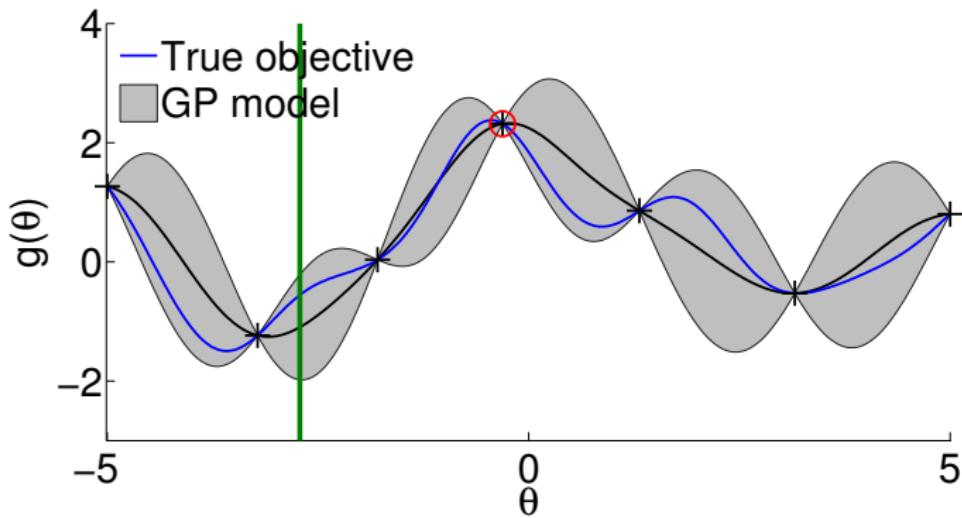
# Bayesian Optimization: Illustration



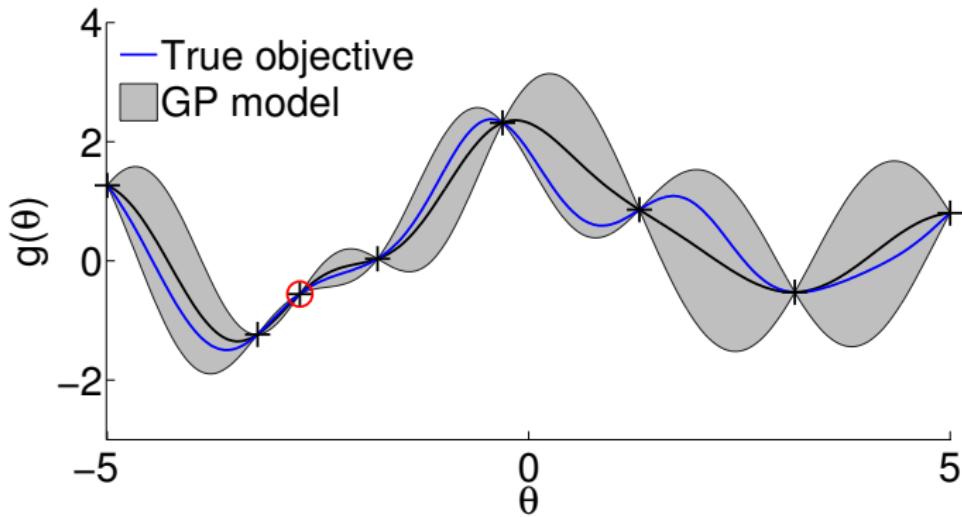
# Bayesian Optimization: Illustration



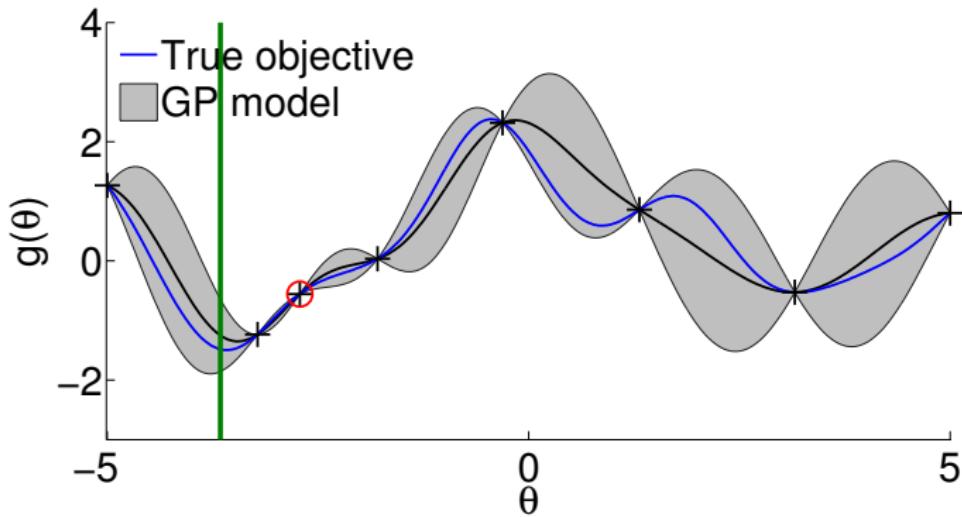
# Bayesian Optimization: Illustration



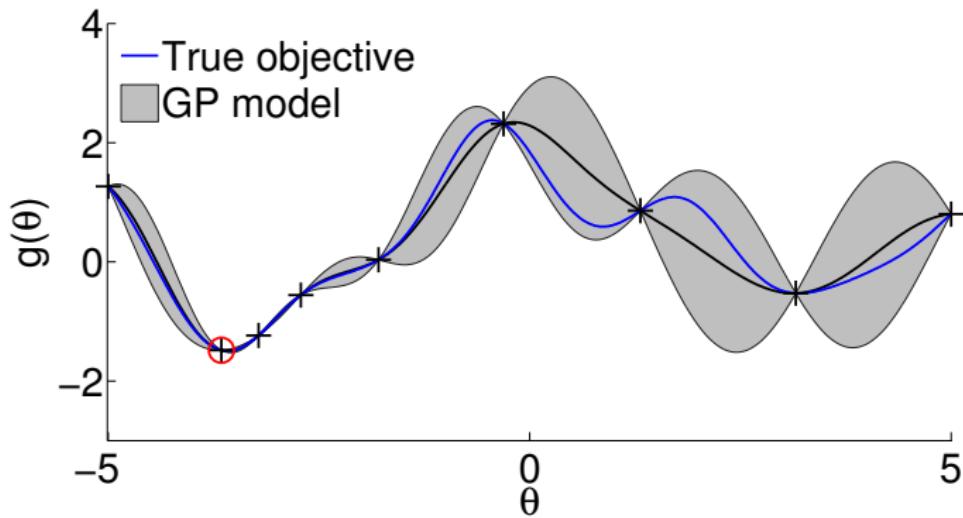
# Bayesian Optimization: Illustration



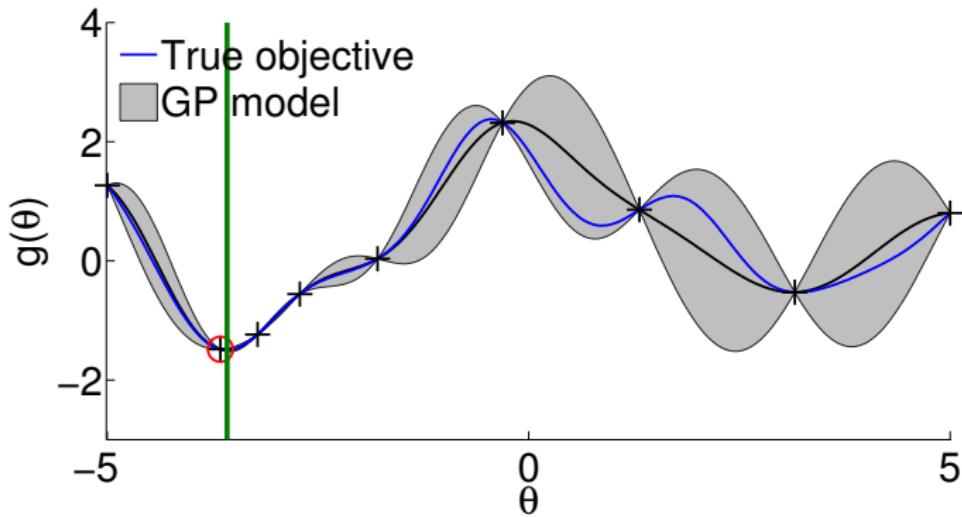
# Bayesian Optimization: Illustration



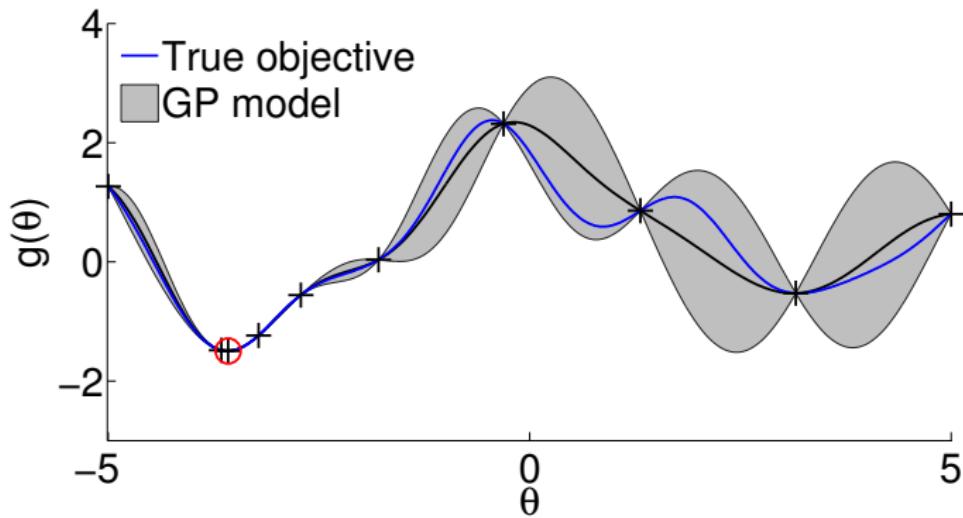
# Bayesian Optimization: Illustration



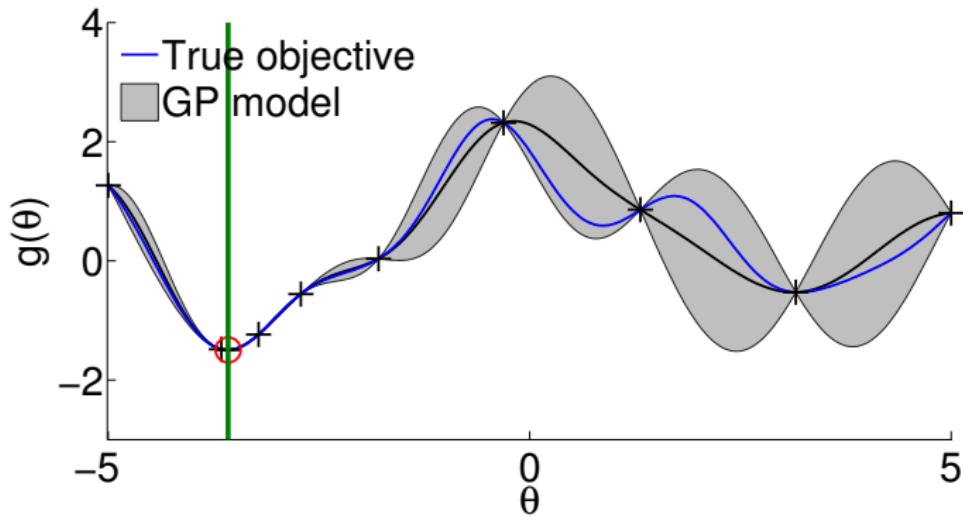
# Bayesian Optimization: Illustration



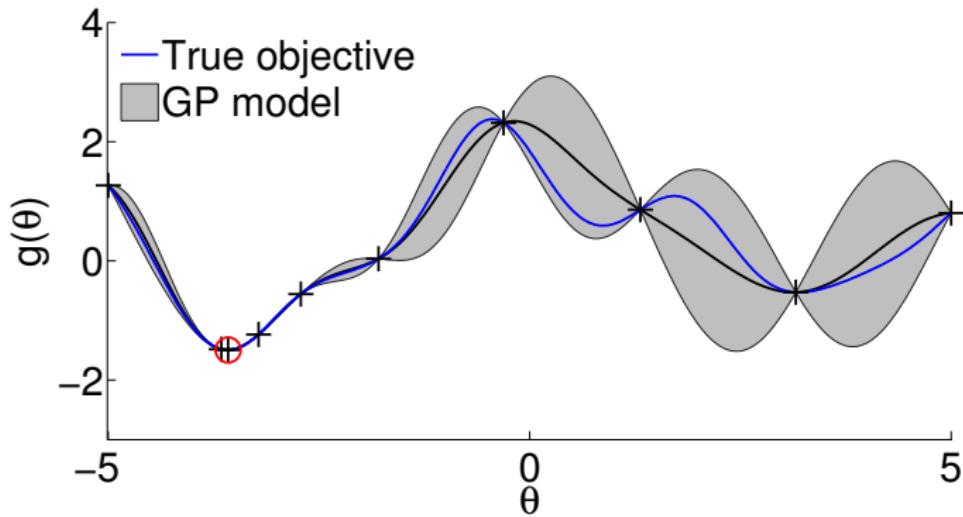
# Bayesian Optimization: Illustration



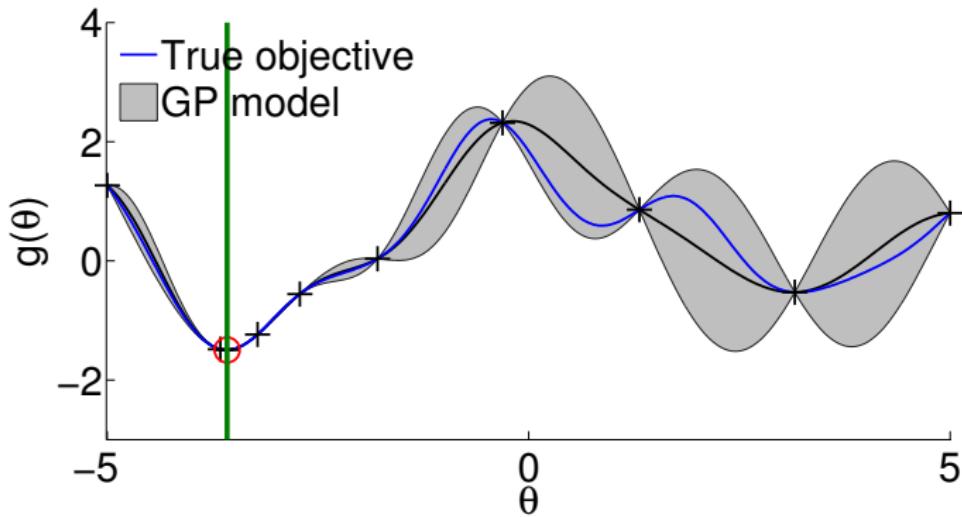
# Bayesian Optimization: Illustration



# Bayesian Optimization: Illustration

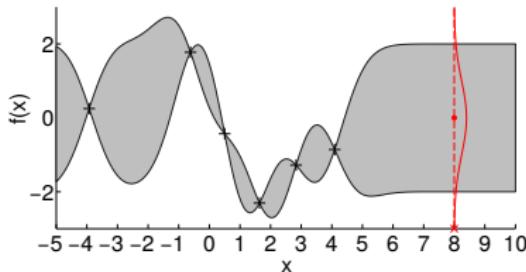


# Bayesian Optimization: Illustration



## Choosing the Next Point to Evaluate the True Objective: Acquisition Functions

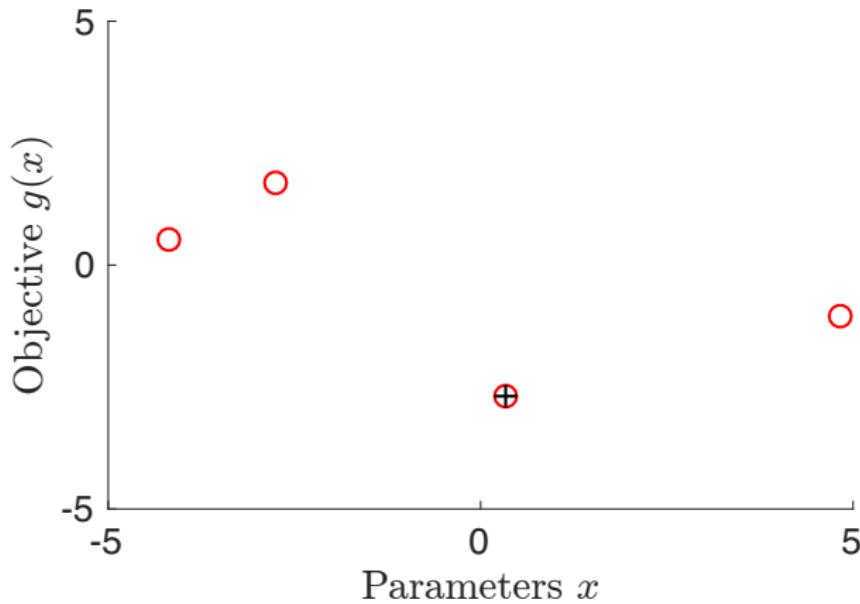
# Using Uncertainty in Global Optimization



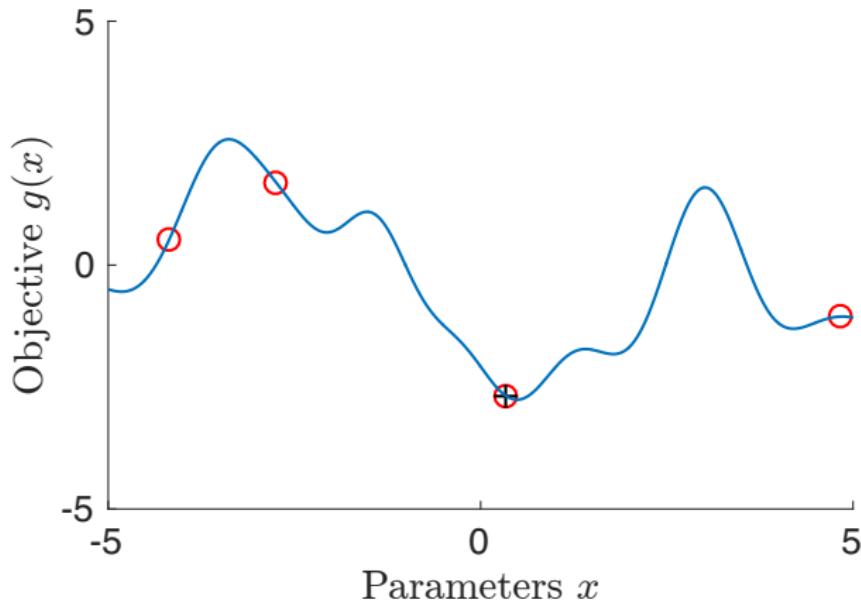
- Find a good (global) optimum
  - ▶ Need to get out of local optima
- Extrapolate from collected knowledge
- GP gives us closed-form means and variances
  - ▶ Trade off exploration and exploitation
    - **Exploration:** Seek places with high variance/uncertainty
    - **Exploitation:** Seek places with low mean
- **Acquisition function  $\alpha$**  trades off exploration and exploitation for our proxy optimization

```
1: Init: Data set  $\mathcal{D}_0 = \{\mathbf{X}_0, \mathbf{y}_0\}$ 
2: for iterations  $t = 1, 2, \dots$  do
3:   Update GP using data  $\mathcal{D}_{t-1}$ 
4:   Select  $\mathbf{x}_t = \arg \max_{\mathbf{x}} \alpha(\mathbf{x})$  by optimizing acquisition function
5:   Query true objective  $g$  at  $\mathbf{x}_t$ 
6:   Augment data set  $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(\mathbf{x}_t, y_t)\}$ 
7: end for
8: Return best input in data set:  $\mathbf{x}^* = \arg \min_{\mathbf{x}} y(\mathbf{x})$ 
```

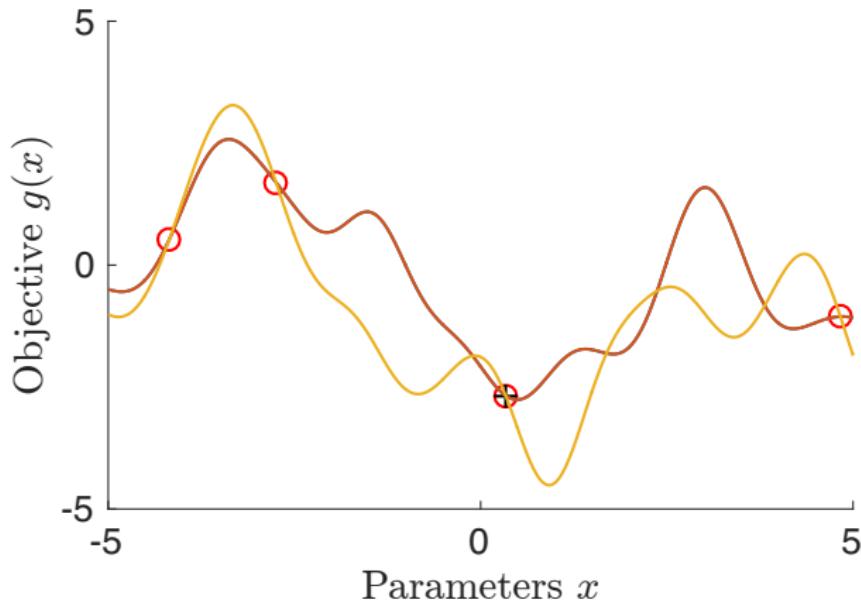
# Where to Evaluate Next?



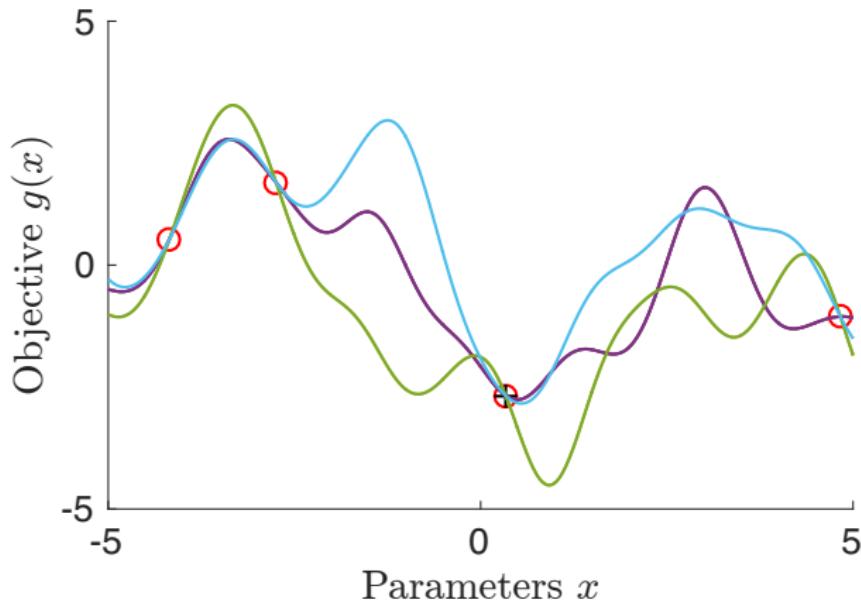
# Where to Evaluate Next?



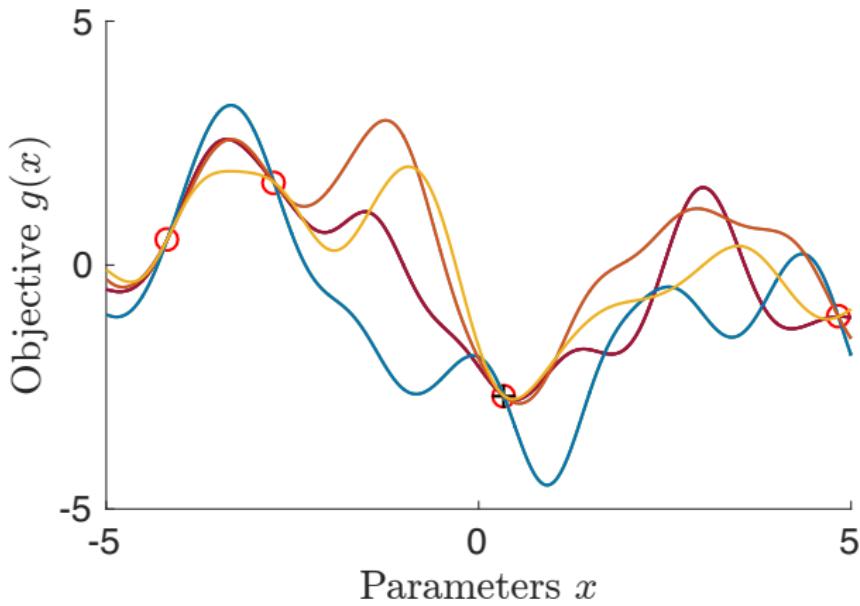
# Where to Evaluate Next?



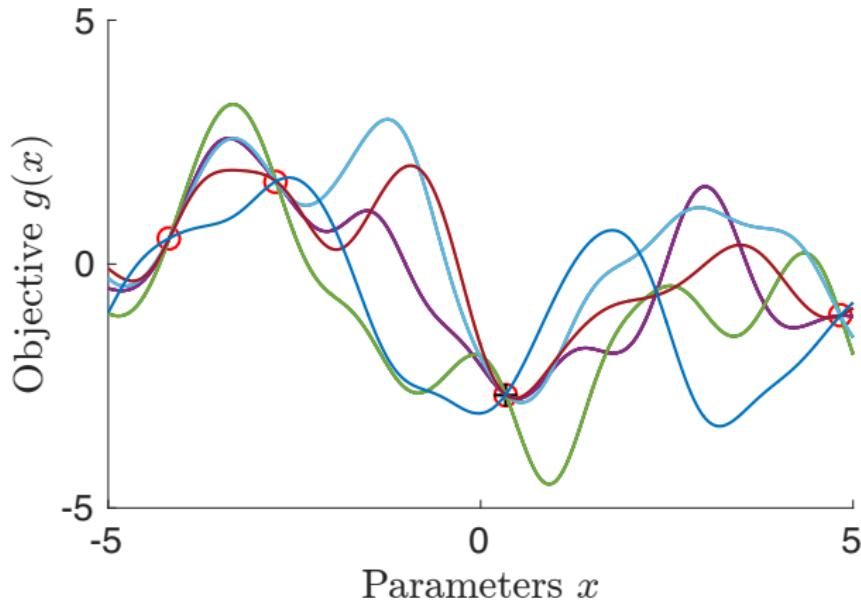
# Where to Evaluate Next?



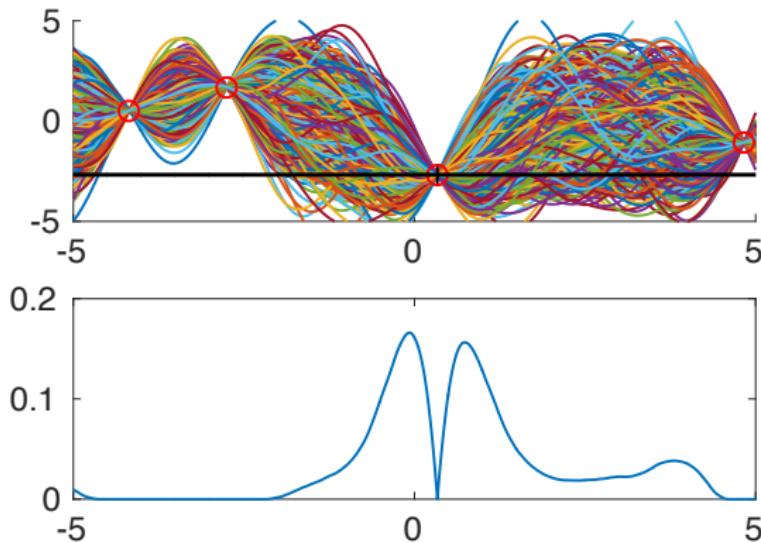
# Where to Evaluate Next?



# Where to Evaluate Next?



# Where to Evaluate Next to Improve Most?



- Upper panel: Samples from a probabilistic proxy  $\tilde{g}$
- Lower panel: Corresponding **expected improvement** over the best solution so far (black cross)
  - ▶ Evaluate  $g$  at the maximum of the expected improvement

- For all  $x \in \mathbb{R}^D$  the GP posterior gives a predictive mean  $\mu(x)$  variance  $\sigma^2(x)$  of  $g(x)$
- Define

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

- **Probability of Improvement (Kushner 1964):**

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

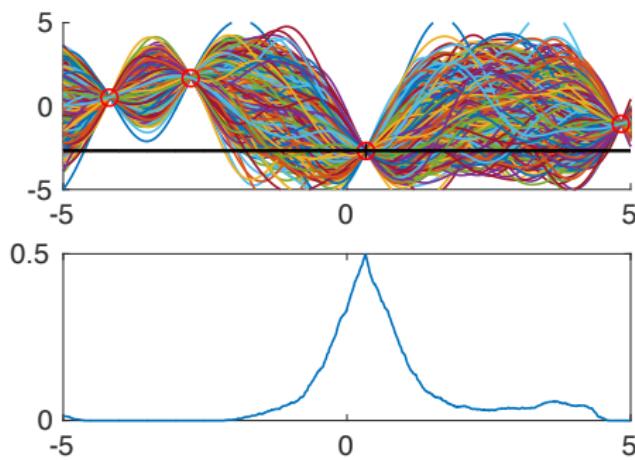
- **Expected Improvement (Mockus 1978):**

$$\alpha_{\text{EI}}(\mathbf{x}) = \sigma(\mathbf{x}) (\gamma(\mathbf{x}) \Phi(\gamma(\mathbf{x})) + \mathcal{N}(\gamma(\mathbf{x}) | 0, 1))$$

- **GP Lower Confidence Bound (Srinivas et al., 2010):**

$$\alpha_{\text{LCB}}(\mathbf{x}) = -(\mu(\mathbf{x}) - \kappa \sigma(\mathbf{x})), \quad \kappa > 0$$

# Probability of Improvement (1)

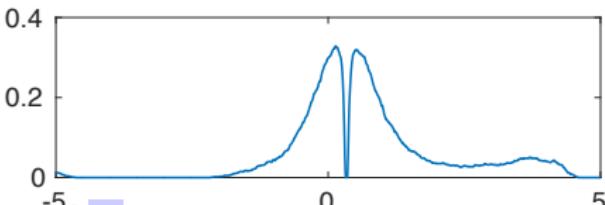
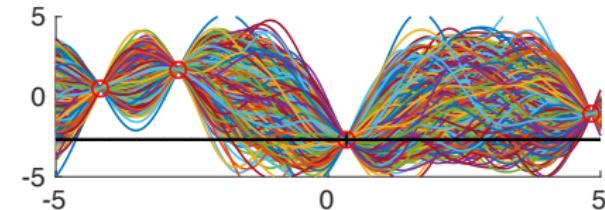
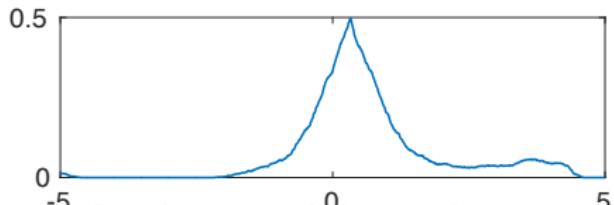
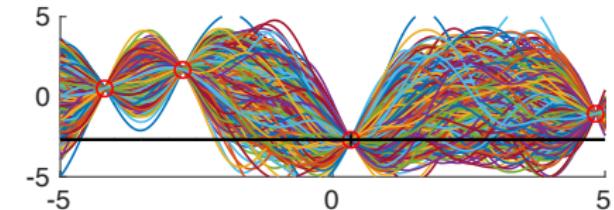


- **Idea:** Determine the probability that  $x_*$  leads to a better function value than the currently best one  $g(\mathbf{x}_{\text{best}})$
- **Sampling-based setting:** Sample  $N$  functions  $g_i$ ; at every input  $\mathbf{x}$  compute a Monte-Carlo estimate

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

- ▶ Can lead to continued exploitation in an  $\epsilon$ -region around  $\mathbf{x}_{\text{best}}$ .
- ▶ Introduce a “slack variable”  $\xi$  for more aggressive exploration

# Probability of Improvement (2)



- Look at a minimum improvement of  $\xi > 0$ :

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

- If  $f \sim GP$  and  $p(g(\mathbf{x})) = \mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))$ :

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

# Expected Improvement

- Idea: Quantify the amount of improvement

- Sampling-based scenario, where  $g_i \sim p(f)$ :

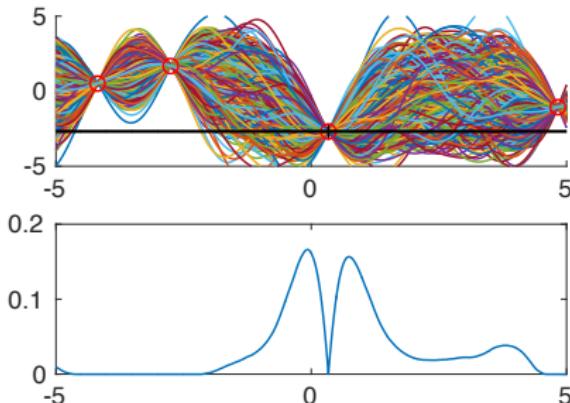
$$\alpha_{EI}(\mathbf{x}) = \mathbb{E}[\max\{0, g(\mathbf{x}_{\text{best}}) - g(\mathbf{x})\}]$$

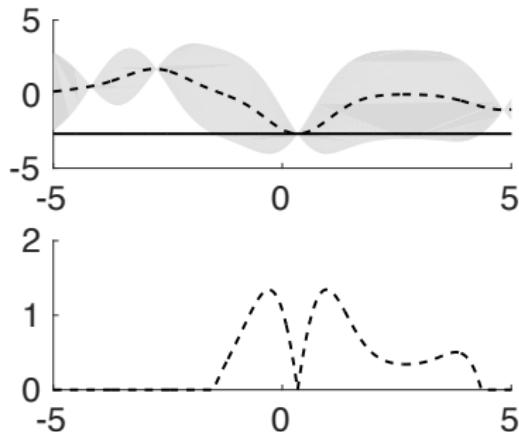
$$\approx \frac{1}{N} \sum_{i=1}^N \max\{0, g(\mathbf{x}_{\text{best}}) - g_i(\mathbf{x})\}$$

- If  $f \sim GP$ , we have a closed-form expression:

$$\alpha_{EI}(\mathbf{x}) = \sigma(\mathbf{x}) (\gamma(\mathbf{x}) \Phi(\gamma(\mathbf{x})) + \mathcal{N}(\gamma(\mathbf{x}) | 0, 1))$$

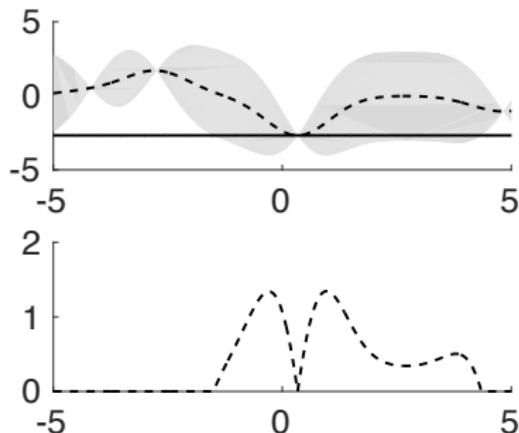
- Slack-variable approach also possible (similar to PI)





- Use the predictive mean  $\mu(x)$  and variance  $\sigma^2(x)$  of the GP prediction directly for targeted exploration by means of the acquisition function

$$\alpha_{\text{LCB}}(\mathbf{x}_t) = -(\mu(\mathbf{x}_t) - \sqrt{\kappa}\sigma(\mathbf{x}_t))$$



- More generally, we can get regret bounds for iteration-dependent  $\kappa$  (Srinivas et al., 2010)

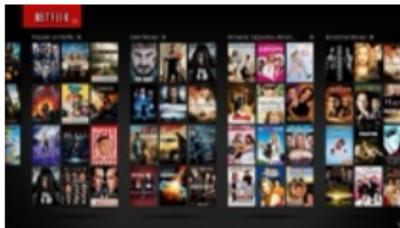
$$\alpha_{\text{LCB}}(\mathbf{x}_t) = -(\mu(\mathbf{x}_t) - \sqrt{\kappa_t} \sigma(\mathbf{x}_t))$$

where  $\kappa_t \in \mathcal{O}(\log t)$  grows with the iteration  $t$

► Continue exploration

# Optimizing the Acquisition Function

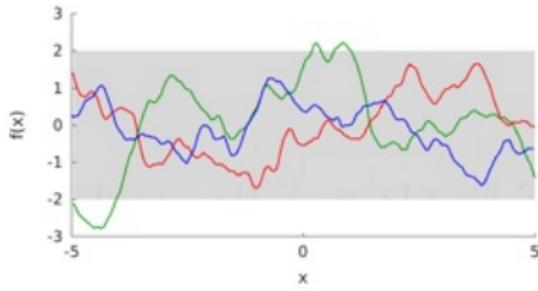
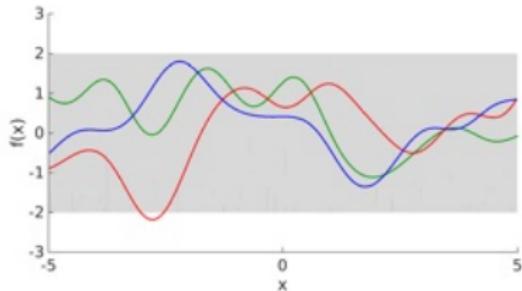
- Optimizing the acquisition function requires us to run a global optimizer inside Bayesian optimization
- What have we gained?
- Evaluating the acquisition function is cheap compared to evaluating the true objective
  - ▶ We can afford evaluating it many times



- Getting the function model (e.g., covariance function) wrong can be catastrophic
- Limited scalability in the number of dimensions and/or evaluations of the true objective function

Why?

# Poor Model Choice



- Covariance function selection is crucial for good performance
  - ▶ Choose a sufficiently flexible and adaptive kernel, e.g., Matérn (but not the squared exponential (Gaussian))
- Nice side-effect of Matérn: Exploration is more encouraged than with the Gaussian kernel

# Choosing Covariance Functions

- Structured SVM for Protein Motif Finding (Miller et al., 2012)
- Optimize hyper-parameters of SSVM using BO (Snoek et al., 2012)

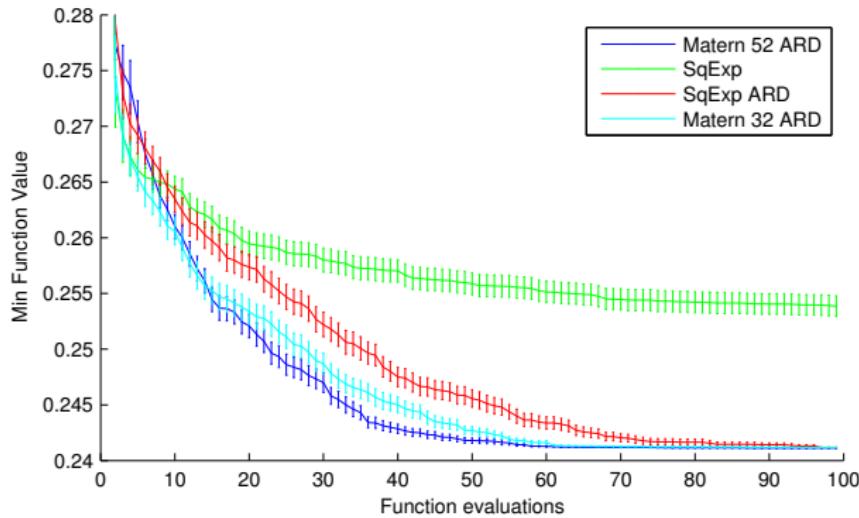


Figure: Figure from Snoek et al. (2012)

- Empirical Bayes (maximize the marginal likelihood) can fail horribly, especially in the early stages of Bayesian optimization when we have only a few data points
- Solution: Integrate out the GP hyper-parameters  $\theta$  by Markov Chain Monte Carlo (MCMC) sampling (e.g., slice sampling)
- Look at integrated acquisition function

$$\begin{aligned}\alpha(\mathbf{x}) &= \mathbb{E}_{\boldsymbol{\theta}}[\alpha(\mathbf{x}, \boldsymbol{\theta})] = \int \alpha(\mathbf{x}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \\ &\approx \frac{1}{K} \sum_{k=1}^K \alpha(\mathbf{x}, \boldsymbol{\theta}^{(k)}), \quad \boldsymbol{\theta}^{(k)} \sim \underbrace{p(\boldsymbol{\theta} | \mathbf{X}_n, \mathbf{y}_n)}_{\text{hyper-parameter posterior}}\end{aligned}$$

# Integrating out GP Hyper-parameters

- Online LDA (Hoffman et al., 2010) for topic modeling
- Two critical hyper-parameters that control the learning rate learned by BO (Snoek et al., 2012)

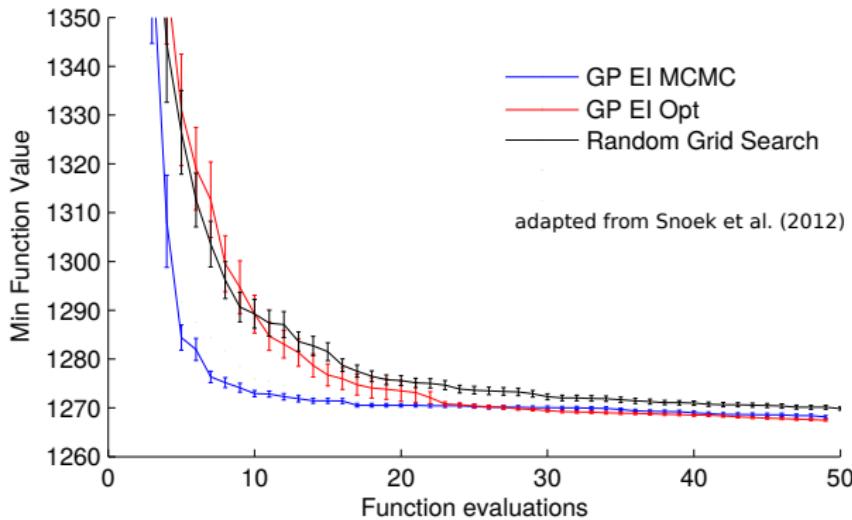
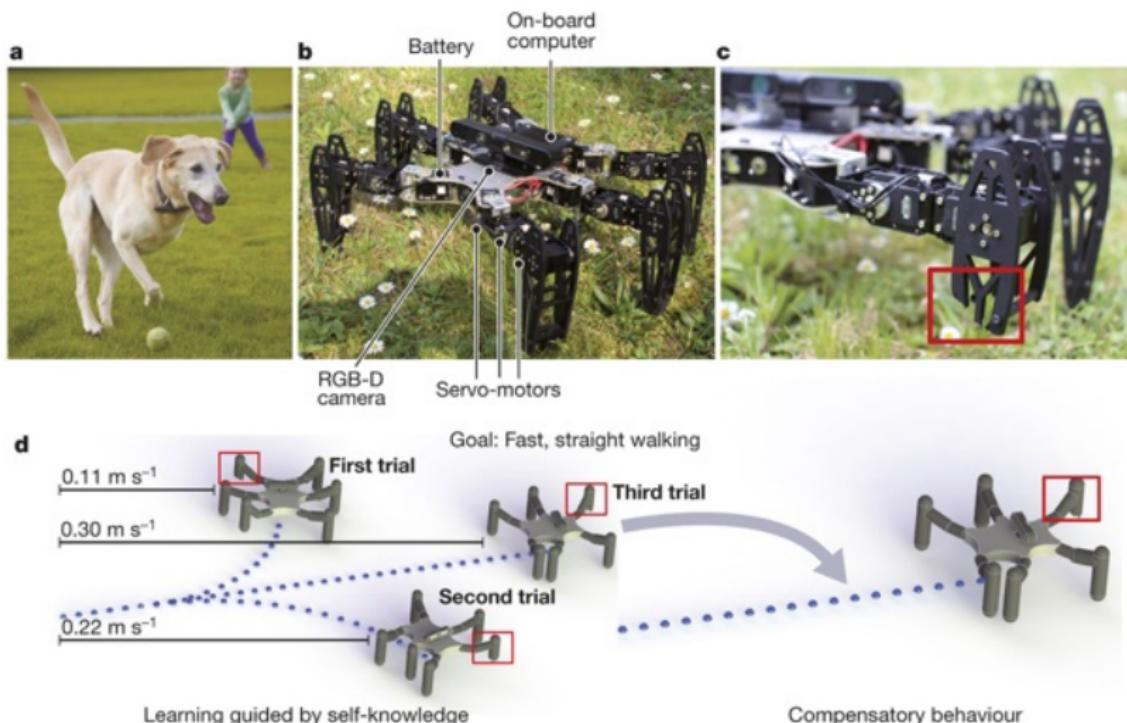


Figure: Figure from Snoek et al. (2012)

# Robots That Learn to Recover from Damage



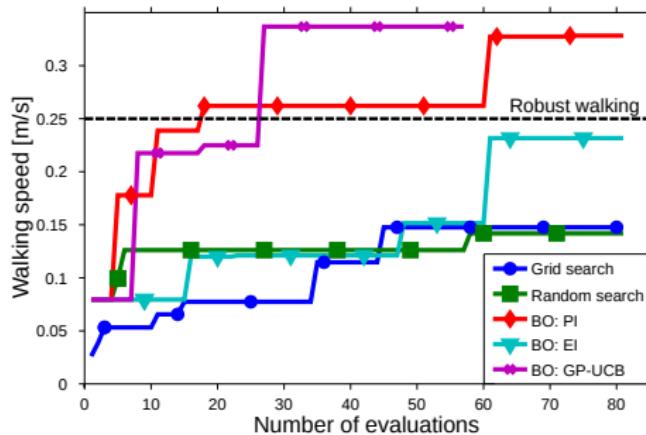
Cully et al. (2015)

- Fragile bipedal robot
  - ▶ Only few experiments feasible
- Maximize robustness and walking speed
- 4 motors:  
2 actuated hips + 2 actuated knees
- Controller implemented as a finite-state-machine (8 parameters)
- Good parameters found after 80–100 experiments
- Substantial speed-up compared to manual parameter search



Calandra et al. (2015)

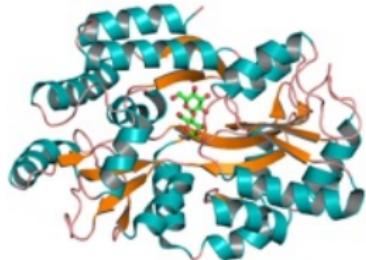
# Comparison



- Squared exponential covariance function
- Learned GP hyper-parameters (no MCMC for integrating them out)

- **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)

- **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)



- 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

# Integration

**Cheng Soon Ong**

Data61, CSIRO

[chengsoon.ong@anu.edu.au](mailto:chengsoon.ong@anu.edu.au)

 @ChengSoonOng

**Marc Peter Deisenroth**

University College London

[m.deisenroth@ucl.ac.uk](mailto:m.deisenroth@ucl.ac.uk)

 @mpd37

December 2020

# Integration problems

- ▶ Moment computation

$$M_k(x) = \int x^k p(x) dx$$

- ▶ Evidence (marginal likelihood)

$$p(\mathbf{X}) = \int p(\mathbf{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$$

- ▶ Relative entropy (KL divergence)

$$\text{KL}(p||q) = \int \log \frac{p(\mathbf{x})}{q(\mathbf{x})} p(\mathbf{x}) d\mathbf{x}$$

- ▶ Prediction in time-series models

$$p(\mathbf{x}_{t+1}) = \int p(\mathbf{x}_{t+1}|\mathbf{x}_t)p(\mathbf{x}_t)d\mathbf{x}_t$$

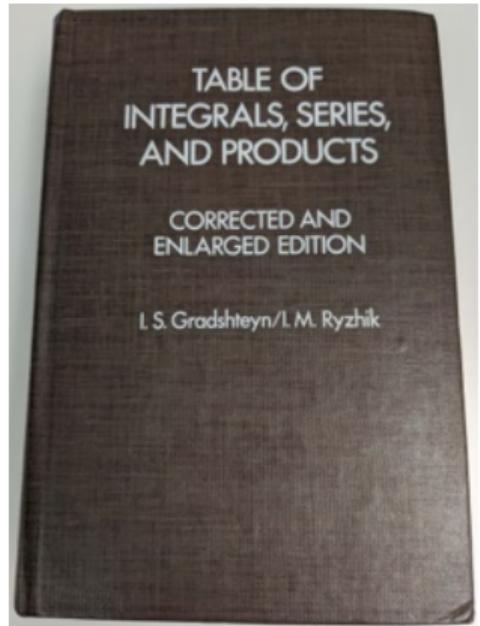
- ▶ Experimental design

$$\mathbb{E}_{\boldsymbol{\theta}}[U(\mathbf{x})] = \int U(\mathbf{x}; \boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$$

- ▶ Planning

$$J^\pi(\mathbf{x}(0)) = \int_0^T r(\mathbf{x}(t), \mathbf{u}(t))|\mathbf{x}(0)dt$$

# Exact integration



498                    DEFINITE INTEGRALS OF ELEMENTARY FUNCTIONS                    (3.95)

2. 
$$\int_{-\infty}^{\infty} x^n e^{-(ax^2+bx+c)} \cos(px+q) dx =$$
$$= \left( \frac{-1}{2a} \right)^n \sqrt{\frac{\pi}{a}} \exp\left(\frac{b^2-p^2}{4a}-c\right) \sum_{k=0}^{E\left(\frac{n}{2}\right)} \frac{n!}{(n-2k)! k!} a^k \times$$
$$\times \sum_{j=0}^{n-2k} \binom{n-2k}{j} b^{n-2k-j} p^j \cos\left(\frac{pb}{2a}-q+\frac{\pi}{2}j\right)$$
$$[a > 0]. \quad \text{GW ((337))(1a)}$$

3.959 
$$\int_0^{\infty} x e^{-px^2} \operatorname{tg} ax dx = \frac{a}{p^3} \sqrt{\frac{\pi}{a}} \sum_{k=1}^{\infty} (-1)^k k \exp\left(-\frac{a^2 k^2}{p^2}\right)$$
$$[p > 0]. \quad \text{BI ((362))(15)}$$

- ▶ Compute integrals analytically, if possible (Gradshteyn & Ryzhik, 2007)

# Approximate integration

Topic	Useful reference	Video
Numerical integration	Stoer & Bulirsch (2002)	Chapter 1
Bayesian quadrature	Rasmussen & Ghahramani (2003), Gunter et al. (2014)	Chapter 1
Monte-Carlo integration	MacKay (2003), Murray (2015)	Chapter 2
Normalizing flows	Weng (2018), Papamakarios et al. (2019), Kobyzev et al. (2020)	Chapter 3
Inference in time series	Julier & Uhlmann (2004), Särkkä (2013)	Chapter 4

# Monte-Carlo Estimation

Cheng Soon Ong  
Marc Peter Deisenroth

December 2020



# Setting: Computing expectations

$$\int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \mathbb{E}_{\mathbf{x} \sim p}[f(\mathbf{x})]$$

Moments of random variables

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

Marginal likelihood

“Average likelihood”

$$p(\mathbf{X}) = \int p(\mathbf{X}|\boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} = \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta})}[p(\mathbf{X}|\boldsymbol{\theta})]$$

Predictions in a Bayesian model

“Average predictive distribution”

$$\begin{aligned} p(\mathbf{x}_* | \mathbf{X}) &= \int p(\mathbf{x}_* | \boldsymbol{\theta}) p(\boldsymbol{\theta} | \mathbf{X}) d\boldsymbol{\theta} \\ &= \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta} | \mathbf{X})}[p(\mathbf{x}_* | \boldsymbol{\theta})] \end{aligned}$$

# Key idea

$$\int f(\mathbf{x})p(\mathbf{x})d\mathbf{x} = \mathbb{E}_{\mathbf{x} \sim p}[f(\mathbf{x})]$$



## Key idea

Make use of random numbers to approximate the expectation.

# How it works

## Key idea

Make use of random numbers to approximate an expectation.

- ▶ Compute expectations via statistical sampling:

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

- ▶ Example: Making predictions in a supervised setting (e.g., Bayesian logistic regression with training set  $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$  at test input  $\mathbf{x}_*$ )

$$p(y_* | \mathbf{x}_*, \mathcal{D}) = \int p(y_* | \boldsymbol{\theta}, \mathbf{x}_*) \underbrace{p(\boldsymbol{\theta} | \mathcal{D})}_{\text{parameter posterior}} d\boldsymbol{\theta}$$

# How it works

## Key idea

Make use of random numbers to approximate an expectation.

- ▶ Compute expectations via statistical sampling:

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

- ▶ Example: Making predictions in a supervised setting (e.g., Bayesian logistic regression with training set  $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$  at test input  $\mathbf{x}_*$ )

$$p(y_*|\mathbf{x}_*, \mathcal{D}) = \int p(y_*|\boldsymbol{\theta}, \mathbf{x}_*) \underbrace{p(\boldsymbol{\theta}|\mathcal{D})}_{\text{parameter posterior}} d\boldsymbol{\theta} \approx \frac{1}{S} \sum_{s=1}^S p(y_*|\boldsymbol{\theta}^{(s)}, \mathbf{x}_*), \quad \boldsymbol{\theta}^{(s)} \sim p(\boldsymbol{\theta}|\mathcal{D})$$

# Properties of Monte Carlo estimation

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

- Estimator is **unbiased** and **asymptotically consistent**, i.e.,

$$\lim_{S \rightarrow \infty} \frac{1}{S} \sum_{s=1}^S f(\mathbf{x}^{(s)}) = \mathbb{E}[f(\mathbf{x})] + \epsilon$$

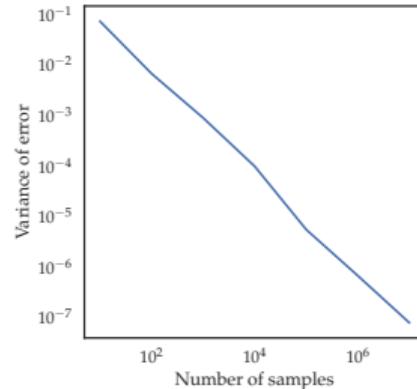
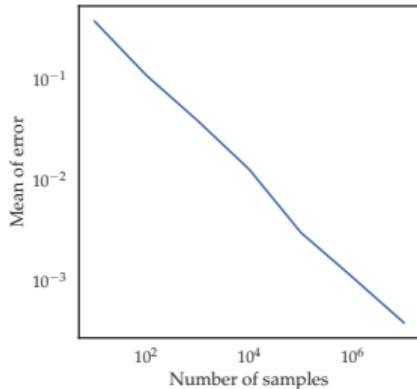
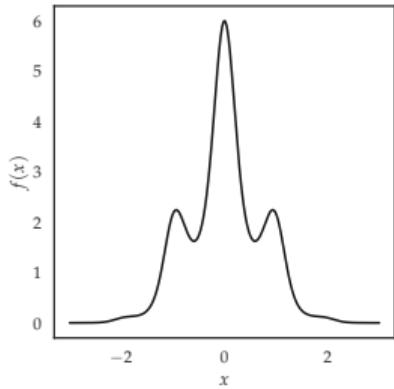
- Error  $\epsilon$  is normal (Gaussian) and its variance shrinks  $\propto 1/S$ , independent of the dimensionality

# Monte Carlo estimation

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

- ▶ 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) ▶▶ Iain Murray’s NeurIPS-2015 tutorial

# Example



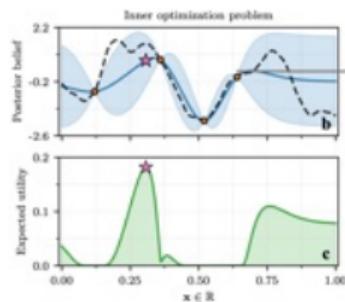
$$Z = \mathbb{E}_x[f(x)] = \int f(x)p(x)dx = \int_{-3}^3 6 \exp\left(-x^2 - \sin(3x)^2\right) \mathcal{U}[-3, 3] dx$$

► Monte-Carlo estimator

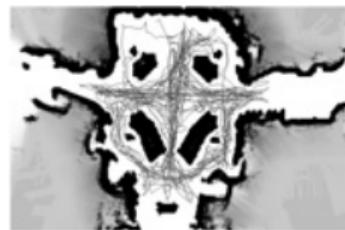
$$\mathbb{E}_{x \sim \mathcal{U}}[f(x)] \approx \frac{1}{S} \sum_{s=1}^S f(x^{(s)}), \quad x^{(s)} \sim \mathcal{U}[-3, 3]$$

# Some application areas

- ▶ Empirical risk minimization (Vapnik, 1991)
- ▶ Reinforcement learning (e.g., Sutton & Barto, 1998)
- ▶ Bayesian optimization  
(e.g., Snoek et al., 2012; Wilson et al., 2018)
- ▶ Variational deep learning  
(e.g., Rezende et al., 2014; Kingma & Welling, 2014)
- ▶ Probabilistic programming
  - ▶▶ Frank Wood's NeurIPS-2015 tutorial
- ▶ High-energy physics (e.g., Buckley et al., 2011)
- ▶ Robotics (e.g., Dellaert et al., 1999)



From Wilson et al. (2018)



From Dellaert et al. (1999)

# Considerations

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

- ▶ 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 (O'Hagan, 1987)

# Summary: Monte Carlo estimation

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



# Change of Variables and Normalizing Flows

Cheng Soon Ong  
Marc Peter Deisenroth

December 2020



# Normalizing flows for density estimation

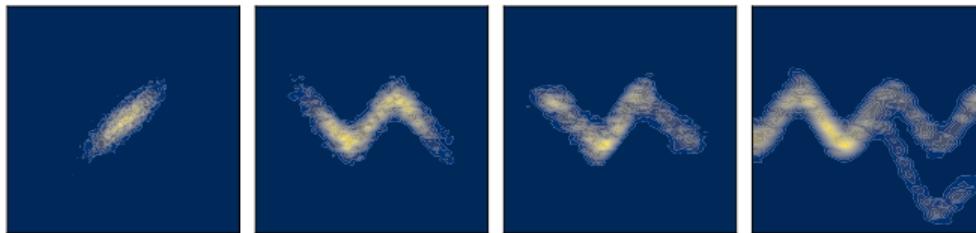


Figure: Generated with PyMC3 (Salvatier et al., 2016)

Key idea

(Tabak & Turner, 2013; Rippel & Adams, 2013; Rezende & Mohamed, 2015)

Build complex distributions from simple distributions via a flow of successive (invertible) transformations

# Normalizing flows for density estimation

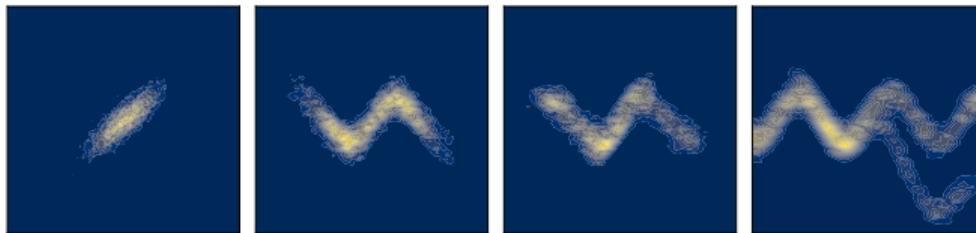


Figure: Generated with PyMC3 (Salvatier et al., 2016)

Key idea

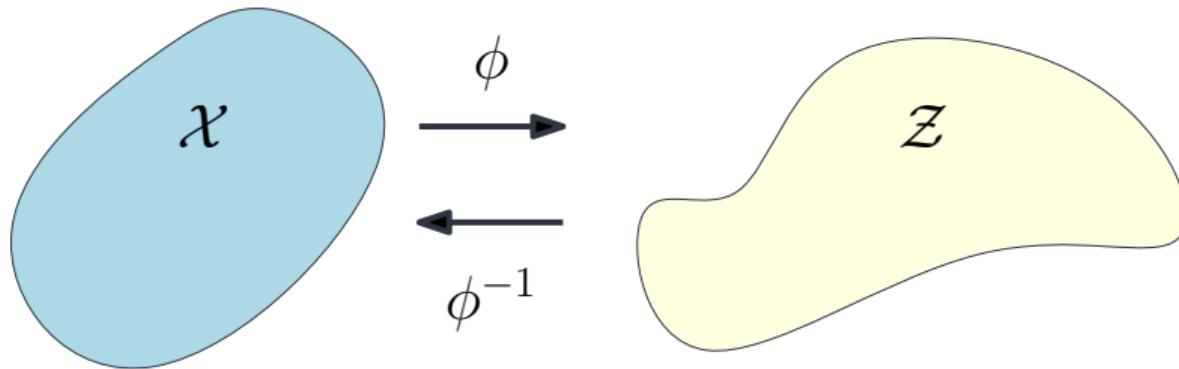
(Tabak & Turner, 2013; Rippel & Adams, 2013; Rezende & Mohamed, 2015)

Build complex distributions from simple distributions via a flow of successive (invertible) transformations

Key ingredient: **Change-of-variables trick**

# Change of Variables

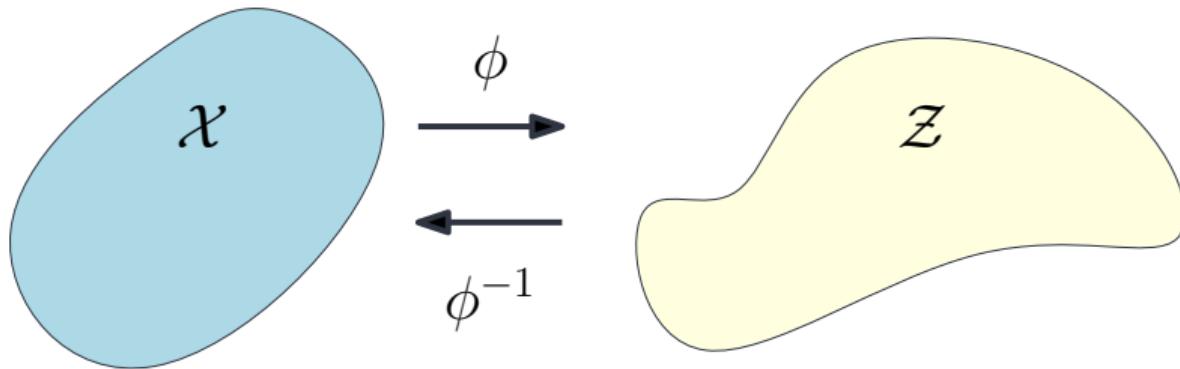
## Change of variables: Key idea



### Key idea

Transform random variable  $X$  into random variable  $Z$  using an invertible transformation  $\phi$ , while keeping track of the change in distribution

## Change of variables: Key idea

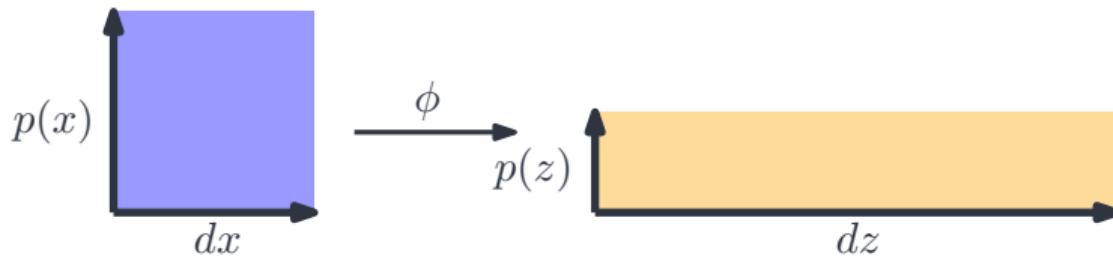


### Key idea

Transform random variable  $X$  into random variable  $Z$  using an invertible transformation  $\phi$ , while keeping track of the change in distribution

- ▶ Distribution  $p_X$  induces distribution  $p_Z$  via transformation  $\phi$
- ▶ Distribution  $p_Z$  induces distribution  $p_X$  via transformation  $\phi^{-1}$

# Jacobian determinant



- ▶ Determinant of Jacobian

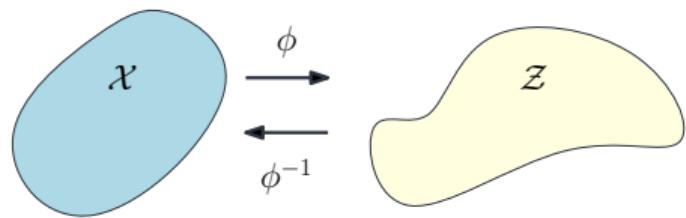
$$\left| \det \left( \frac{dz}{dx} \right) \right| = \left| \det \left( \frac{d\phi(x)}{dx} \right) \right|$$

tells us how much the domain  $dx$  is stretched to  $dz$

# How it works

- ▶ Constraint: volume preservation

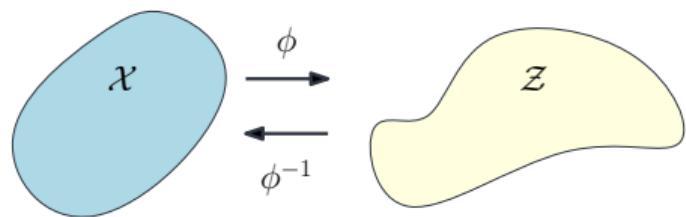
$$\int_{\mathcal{X}} p_X(\mathbf{x}) d\mathbf{x} = 1 = \int_{\mathcal{Z}} p_Z(\mathbf{z}) d\mathbf{z}$$



# How it works

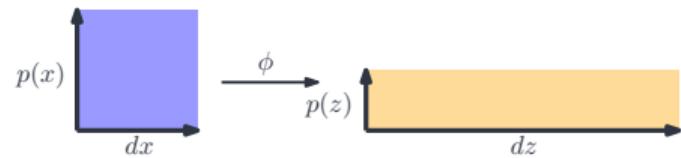
- ▶ Constraint: volume preservation

$$\int_{\mathcal{X}} p_X(\mathbf{x}) d\mathbf{x} = 1 = \int_{\mathcal{Z}} p_Z(\mathbf{z}) d\mathbf{z}$$

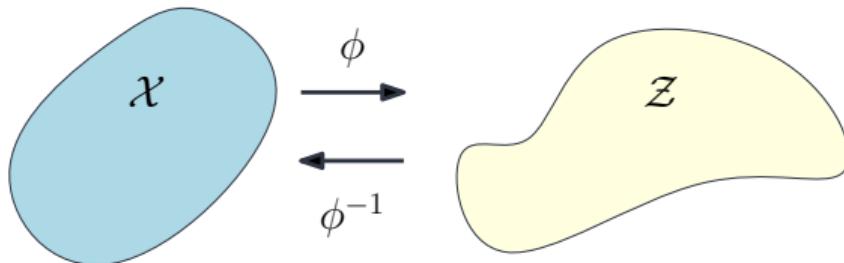


- ▶ Volume preservation: Rescale  $p_Z$  by the inverse of Jacobian determinant

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



# Considerations



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

- ▶ Express target distribution  $p_Z$  in terms of known distribution  $p_X$  and the Jacobian determinant of an invertible mapping  $\phi$
- ▶ No need to invert  $\phi$  explicitly
- ▶ Generate expressive distributions  $p_Z$  by simple  $p_X$  and flexible transformation  $\phi$

# Applications

- ▶ Numerical integration (turn indefinite integrals into definite ones)
- ▶ Neural ODEs (E 2017, Chen et al., 2018)
- ▶ Learning in implicit generative models (e.g., GANs) and likelihood-free inference (e.g., ABC)  
(e.g., Mohamed & Lakshminarayanan, 2016; Sisson et al., 2007)
- ▶ **Normalizing flows** (Rezende & Mohamed, 2015)

# Normalizing Flows

# Normalizing flows for density estimation

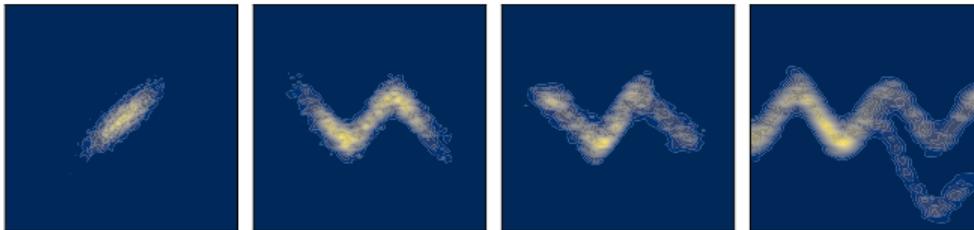


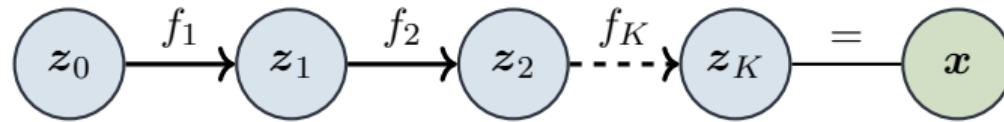
Figure: Generated with PyMC3 (Salvatier et al., 2016)

Key idea

(Tabak & Turner, 2013; Rippel & Adams, 2013; Rezende & Mohamed, 2015)

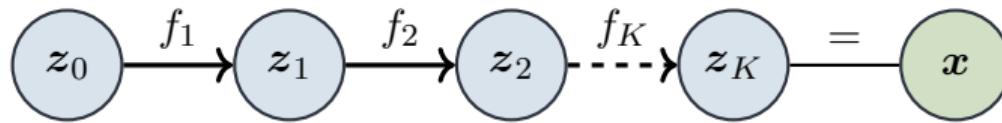
Build complex distributions from simple distributions via a flow of successive (invertible) transformations

## How it works



- ▶ Random variable  $z_0 \sim p_0$
- ▶ Simple base distribution  $p_0$ , e.g.  $p_0 = \mathcal{N}(\mathbf{0}, \mathbf{I})$

## How it works



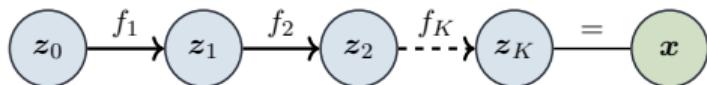
- ▶ Random variable  $z_0 \sim p_0$
- ▶ Simple base distribution  $p_0$ , e.g.  $p_0 = \mathcal{N}(\mathbf{0}, \mathbf{I})$
- ▶ Successive transformation of  $z_k$  via invertible transformations  $f_k$ :

$$z_k = f_k(z_{k-1})$$

- ▶ Observed data  $x = z_K$  at the end of the chain

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

# Marginal distribution

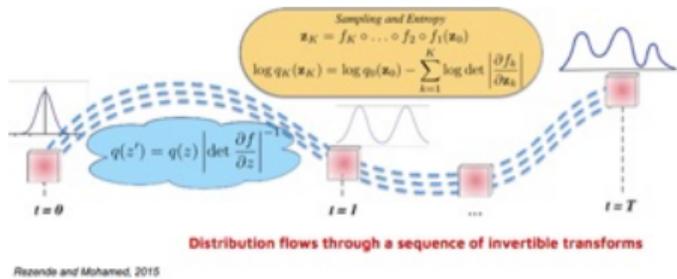


- ▶ Repeated application of **change-of-variables** trick

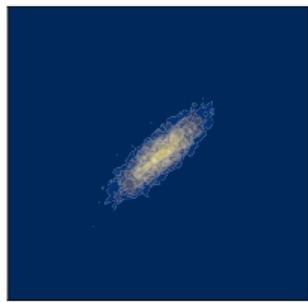
$$p(\mathbf{x}) = p(\mathbf{z}_K) = p(\mathbf{z}_0) \prod_{k=1}^K \left| \det \frac{df_k(\mathbf{z}_{k-1})}{d\mathbf{z}_{k-1}} \right|^{-1}$$

- ▶ Entropy

$$\log p(\mathbf{x}) = \log p(\mathbf{z}_K) = \log p(\mathbf{z}_0) - \sum_{k=1}^K \log \left| \det \left( \frac{df_k(\mathbf{z}_{k-1})}{d\mathbf{z}_{k-1}} \right) \right|$$



## Illustration with PyMC3 (Salvatier et al., 2016)

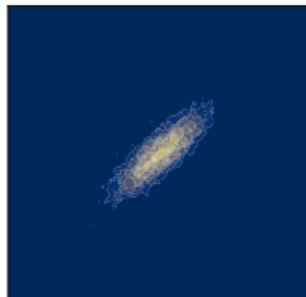


1 planar flow

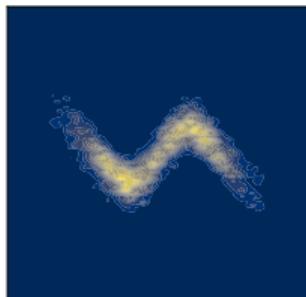
Figure: Generated using a PyMC3 tutorial (Salvatier et al., 2016)

- ▶ Repeated application of a planar flow  $\mathbf{z}_k = f_k(\mathbf{z}_{k-1}) = \mathbf{z}_{k-1} + \mathbf{u}\sigma(\mathbf{w}^\top \mathbf{z}_{k-1} + b)$

## Illustration with PyMC3 (Salvatier et al., 2016)



1 planar flow

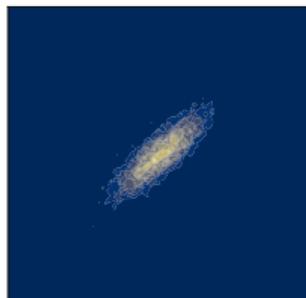


2 planar flows

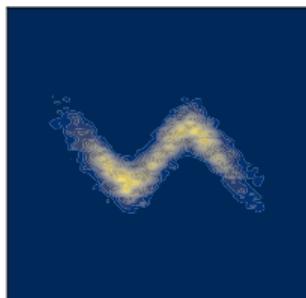
Figure: Generated using a PyMC3 tutorial (Salvatier et al., 2016)

- ▶ Repeated application of a planar flow  $\mathbf{z}_k = f_k(\mathbf{z}_{k-1}) = \mathbf{z}_{k-1} + \mathbf{u}\sigma(\mathbf{w}^\top \mathbf{z}_{k-1} + b)$

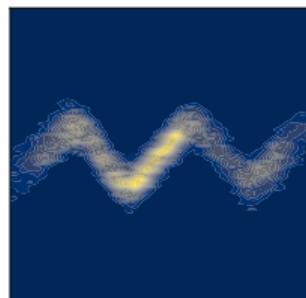
## Illustration with PyMC3 (Salvatier et al., 2016)



1 planar flow



2 planar flows



3 planar flows

Figure: Generated using a PyMC3 tutorial (Salvatier et al., 2016)

- ▶ Repeated application of a planar flow  $\mathbf{z}_k = f_k(\mathbf{z}_{k-1}) = \mathbf{z}_{k-1} + \mathbf{u}\sigma(\mathbf{w}^\top \mathbf{z}_{k-1} + b)$

## Illustration with PyMC3 (Salvatier et al., 2016)

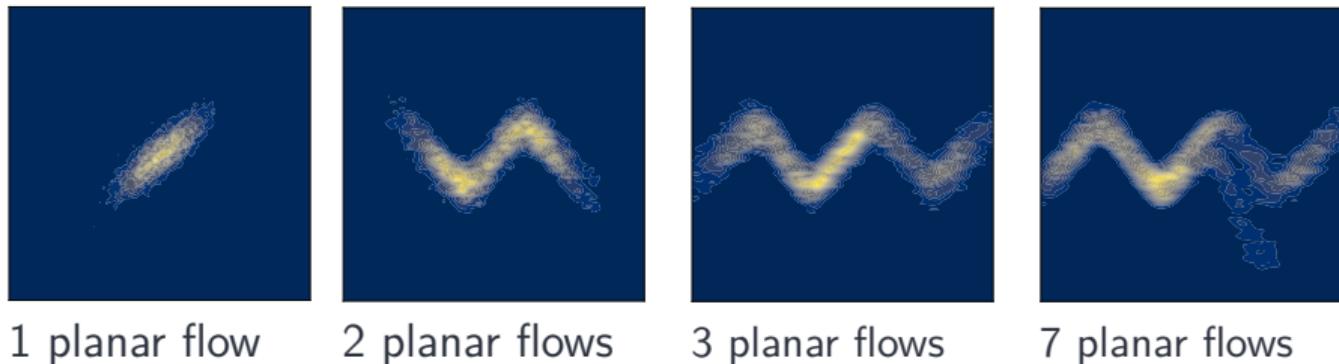


Figure: Generated using a PyMC3 tutorial (Salvatier et al., 2016)

- ▶ Repeated application of a planar flow  $\mathbf{z}_k = f_k(\mathbf{z}_{k-1}) = \mathbf{z}_{k-1} + \mathbf{u}\sigma(\mathbf{w}^\top \mathbf{z}_{k-1} + b)$

## Illustration with PyMC3 (Salvatier et al., 2016)

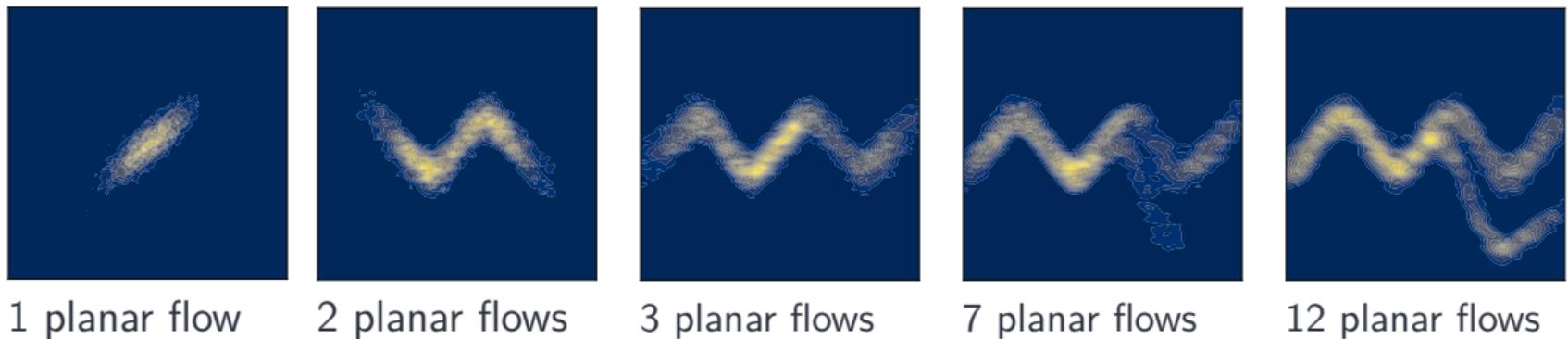


Figure: Generated using a PyMC3 tutorial (Salvatier et al., 2016)

- ▶ Repeated application of a planar flow  $\mathbf{z}_k = f_k(\mathbf{z}_{k-1}) = \mathbf{z}_{k-1} + \mathbf{u}\sigma(\mathbf{w}^\top \mathbf{z}_{k-1} + b)$

## Computing expectations

$$\mathbb{E}_{p_X}[l(\boldsymbol{x})] = \mathbb{E}_{p_K}[l(\boldsymbol{z}_K)] = \mathbb{E}_{p_0}[l(f_K \circ \dots \circ f_1(\boldsymbol{z}_0))]$$

# Computing expectations

$$\mathbb{E}_{p_X}[l(\mathbf{x})] = \mathbb{E}_{p_K}[l(\mathbf{z}_K)] = \mathbb{E}_{p_0}[l(f_K \circ \dots \circ f_1(\mathbf{z}_0))]$$

- ▶ Expectations w.r.t.  $p_K$  can be computed without explicitly knowing  $p_K$  or  $p_X$ 
  - ▶ Sample  $\mathbf{z}_0^{(s)} \sim p_0$
  - ▶ Push sample forward through sequence of deterministic transformations
    - ▶▶ Valid sample  $\mathbf{x}^{(s)} \sim p_X(x)$
- ▶ Monte-Carlo estimation to get expected value

## Computational considerations

---

- ▶ Compute log-determinant of Jacobian
- ▶ Cheap (linear) if Jacobian is (block-)diagonal or triangular

# Computational considerations

- ▶ Compute log-determinant of Jacobian
- ▶ Cheap (linear) if Jacobian is (block-)diagonal or triangular
- ▶ Require partial derivatives

$$\frac{\partial z_k^{(d)}}{\partial z_{k-1}^{(>d)}} = 0 \quad \Rightarrow \quad \frac{d\mathbf{z}_k}{d\mathbf{z}_{k-1}} = \begin{bmatrix} \frac{\partial z_k^{(1)}}{\partial z_{k-1}^{(1)}} & \mathbf{0} & \dots & \mathbf{0} \\ \frac{\partial z_k^{(2)}}{\partial z_{k-1}^{(1)}} & \frac{\partial z_k^{(2)}}{\partial z_{k-1}^{(2)}} & \dots & \vdots \\ \vdots & \ddots & & \mathbf{0} \\ \frac{\partial z_k^{(D)}}{\partial z_{k-1}^{(1)}} & \dots & \dots & \frac{\partial z_k^{(D)}}{\partial z_{k-1}^{(D)}} \end{bmatrix} \in \mathbb{R}^{D \times D}.$$

# Autoregressive flows

- ▶ High-level idea:

$$z_k^{(d)} = \phi(z_{k-1}^{(\leq d)})$$

# Autoregressive flows

- ▶ High-level idea:

$$z_k^{(d)} = \phi(z_{k-1}^{(\leq d)})$$

- ▶ NICE (Dinh et al., 2014)
- ▶ Inverse autoregressive flow (Kingma et al., 2016)
- ▶ Real NVP (Dinh et al., 2017)
- ▶ Masked autoregressive flow (Papamakarios et al., 2017)
- ▶ Glow (Kingma & Dhariwal, 2018)
- ▶ (Block) neural autoregressive flows, spline flows, ... (e.g., Huang et al., 2018; de Cao et al., 2019; Durkan et al., 2019 )

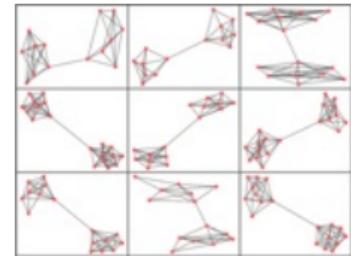
## Application areas

---

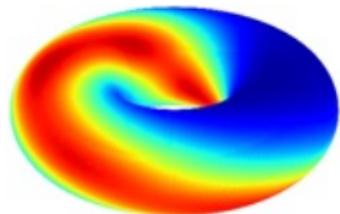
- ▶ Variational inference in deep generative models  
(e.g., Rezende & Mohamed, 2015)
- ▶ Graph neural networks (Liu et al., 2019)
- ▶ Parallel WaveNet (van den Oord et al., 2018)

# Application areas

- ▶ Variational inference in deep generative models  
(e.g., Rezende & Mohamed, 2015)
- ▶ Graph neural networks (Liu et al., 2019)
- ▶ Parallel WaveNet (van den Oord et al., 2018)
- ▶ Continuous flows
  - ▶ Neural ODEs (e.g, E, 2017; Chen et al., 2018)
  - ▶ Flows on manifolds (e.g., Gemici et al., 2016; Rezende et al., 2020; Mathieu & Nickel, 2020)

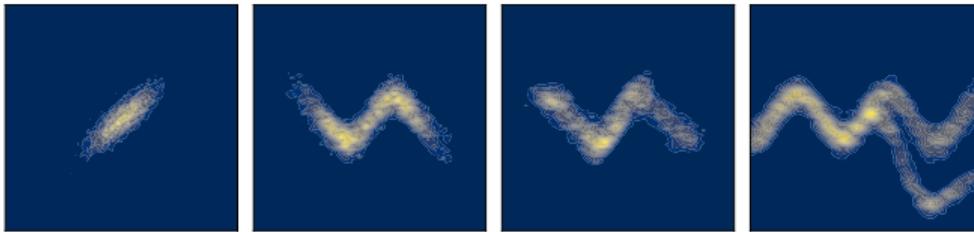


From Liu et al. (2019)



From Rezende et al. (2020)

# Summary



- ▶ Normalizing flows provide a constructive way to generate rich distributions
- ▶ Key idea: Transform a simple distribution using a flow of successive (invertible) transformations
- ▶ Key ingredient: Change-of-variables trick
- ▶ Jacobians can be computed efficiently, if the transformations are defined appropriately
- ▶ Can be used as a generator and inference mechanism

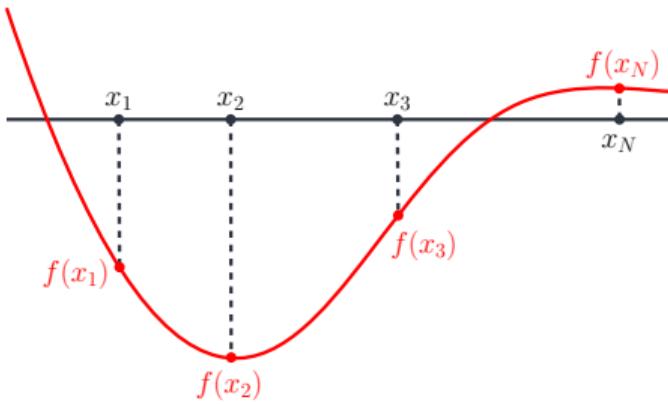
# Numerical Integration

Cheng Soon Ong  
Marc Peter Deisenroth

December 2020



# Setting

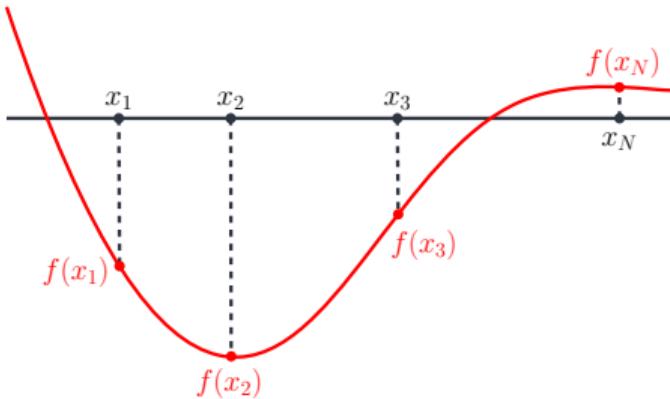


► Approximate

$$\int_a^b f(x)dx \approx \sum_{n=1}^N w_n f(x_n), \quad x \in \mathbb{R}$$

► Nodes  $x_n$  and corresponding function values  $f(x_n)$

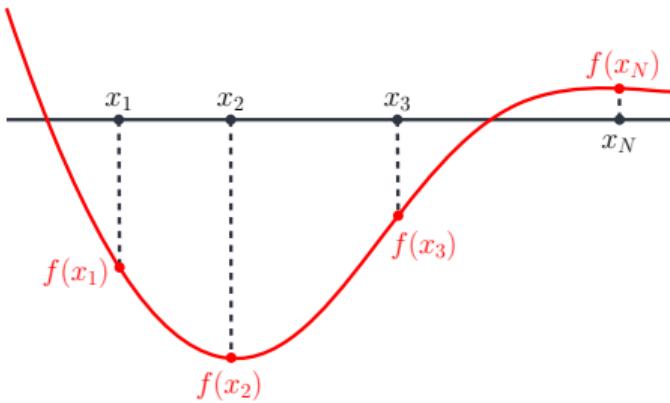
# Numerical integration (quadrature)



## Key idea

Approximate  $f$  using an interpolating function that is easy to integrate  
(e.g., polynomial)

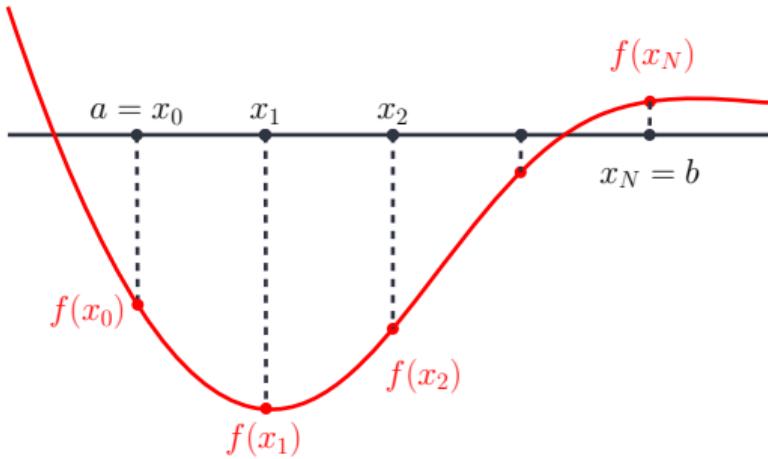
# Quadrature approaches



Quadrature	Interpolant	Nodes
<b>Newton–Cotes</b>	low-degree polynomials	equidistant
<b>Gaussian</b>	orthogonal polynomials	roots of polynomial
<b>Bayesian</b>	Gaussian process	user defined

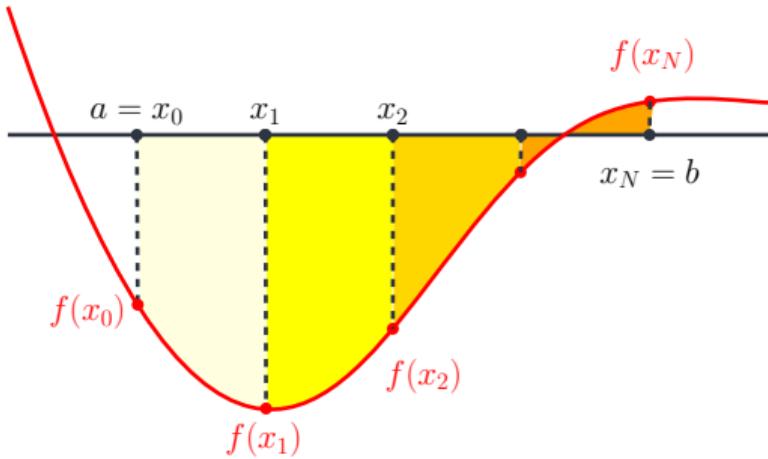
# Newton-Cotes Quadrature

# Overview



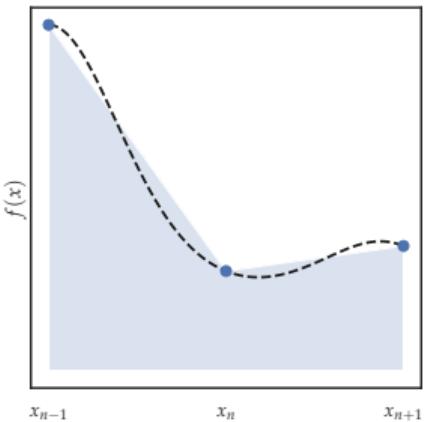
- ▶ Equidistant nodes  $a = x_0, \dots, x_N = b$  ►► Partition interval  $[a, b]$
- ▶ Approximate  $f$  in each partition with a low-degree polynomial

# Overview



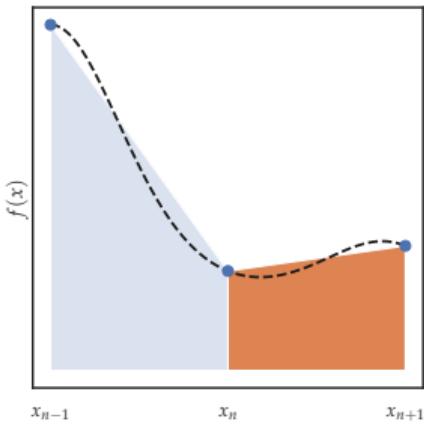
- ▶ Equidistant nodes  $a = x_0, \dots, x_N = b$  ►► Partition interval  $[a, b]$
- ▶ Approximate  $f$  in each partition with a low-degree polynomial
- ▶ Compute integral for each partition analytically and sum them up

# Trapezoidal rule



- ▶ Partition  $[a, b]$  into  $N$  segments with equidistant nodes  $x_n$
- ▶ **Locally linear approximation** of  $f$  between nodes

## Trapezoidal rule (2)



- ▶ Area of a trapezoid with corners  $(x_n, x_{n+1}, f(x_{n+1}), f(x_n))$

$$\int_{x_n}^{x_{n+1}} f(x)dx \approx \frac{h}{2}(f(x_n) + f(x_{n+1}))$$

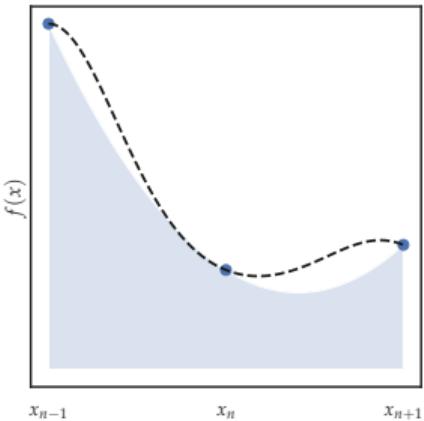
$h := |x_{n+1} - x_n|$  ➡ Distance between nodes

- ▶ Error  $\mathcal{O}(h^2)$

- ▶ Full integral:

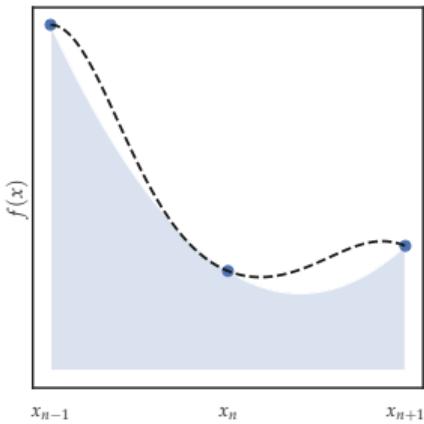
$$\int_a^b f(x)dx \approx \frac{h}{2}(f_0 + 2f_1 + \cdots + 2f_{N-1} + f_N), \quad f_n := f(x_n)$$

# Simpson's rule



- ▶ Partition  $[a, b]$  into  $N$  segments with equidistant nodes  $x_n$
- ▶ **Locally quadratic approximation** of  $f$  connecting triplets  $(f(x_{n-1}), f(x_n), f(x_{n+1}))$

## Simpson's rule (2)



► Area of segment:

$$\int_{x_{n-1}}^{x_{n+1}} f(x) dx \approx \frac{h}{3} (f_{n-1} + 4f_n + f_{n+1})$$

$h := |x_{n+1} - x_n|$  ►► Distance between nodes

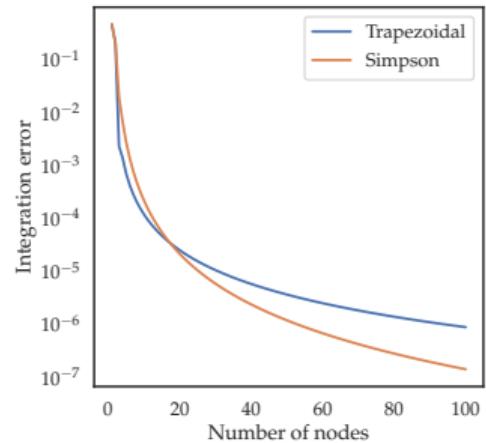
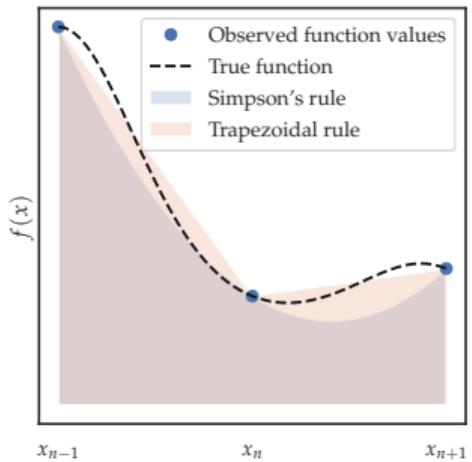
► Error:  $\mathcal{O}(h^4)$

► Full integral:

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

# Example

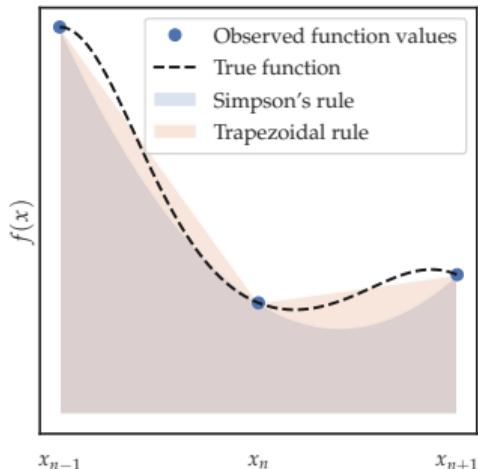
$$\int_0^1 \exp(-x^2 - \sin(3x)^2) dx$$



- ▶ Simpson's rule yields better approximations
- ▶ Very good approximations obtained fairly quickly

# 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 and smaller integration error



# Gaussian Quadrature

# Gaussian quadrature

- ▶ Named after Carl Friedrich Gauß
- ▶ Quadrature scheme that no longer relies on equidistant nodes ➡ Higher accuracy
- ▶ Central approximation

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

- ▶ **Weight function**  $w(x) \geq 0$  (and some other integration-related properties, which are satisfied if  $w(x)$  is a pdf)
- ▶ Goal: Find nodes  $x_n$  and weights  $w_n$ , so that the approximation error is minimized

## Central idea

- ▶ Quadrature nodes  $x_n$  are the roots of a family of **orthogonal polynomials**
  - ▶▶ Nodes no longer equidistant
- ▶ Exact if  $f$  is a polynomial of degree  $\leq 2N - 1$ , i.e.,

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

- ▶▶ Integral can be computed exactly by evaluating  $f$   $N$  times at the optimal locations  $x_n$  (roots of an orthogonal polynomial) with corresponding optimal weights  $w_n$
- ▶▶ More accurate than Newton–Cotes for the same number of evaluations (with some memory overhead)

## Example: Gauß–Hermite quadrature

► Solve

$$\begin{aligned}\int f(x) \underbrace{\exp(-x^2)}_{w(x)} dx &= \int f(x) \frac{\sqrt{2\pi}}{\exp(-x^2/2)} \mathcal{N}(x|0, 1) dx \\ &= \sqrt{2\pi} \mathbb{E}_{x \sim \mathcal{N}(0, 1)} \left[ \frac{f(x)}{\exp(-x^2/2)} \right]\end{aligned}$$

► With change-of-variables trick ➡ Expectation w.r.t. a Gaussian measure

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

## Example: Gauß–Hermite quadrature (2)

- ▶ Follow general approximation scheme

$$\int f(x) \underbrace{\exp(-x^2)}_{w(x)} dx \approx \sum_{n=1}^N w_n f(x_n)$$

- ▶ **Nodes**  $x_1, \dots, x_N$  are the roots of Hermite polynomial

$$H_N(x) := (-1)^n \exp\left(\frac{x^2}{2}\right) \frac{d^n}{dx^n} \exp(-x^2)$$

- ▶ **Weights**  $w_n$  are

$$w_n := \frac{2^{N-1} N! \sqrt{\pi}}{N^2 H_{N-1}^2(x_n)}$$

## Overview (Stoer & Bulirsch, 2002)

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

$[a, b]$	$w(x)$	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

## Application areas

- ▶ Probabilities for rectangular bivariate/trivariate Gaussian and  $t$  distributions (Genz, 2004)
- ▶ Integrating out (marginalizing) a few hyper-parameters in a latent-variable model (INLA; Rue et al., 2009)
- ▶ Predictions with a Gaussian process classifier (GPFlow; Matthews et al., 2017)

## Summary: Gaussian quadrature

---

- ▶ Orthogonal polynomials to approximate  $f$
- ▶ Nodes are the roots of the polynomial
- ▶ Higher accuracy than Newton–Cotes
- ▶ **Method of choice** for low-dimensional problems (1–3 dimensions)
- ▶ Can't naturally deal with noisy observations
- ▶ Only works in low dimensions
- ▶ Approaches that scale better with dimensionality
  - ▶▶ **Bayesian quadrature** (up to  $\approx 10$  dimensions)
  - ▶▶ **Monte Carlo estimation** (high dimensions)

# Bayesian Quadrature

# Bayesian quadrature: Setting and key idea

$$Z := \int f(\boldsymbol{x}) p(\boldsymbol{x}) d\boldsymbol{x} = \mathbb{E}_{\boldsymbol{x} \sim p}[f(\boldsymbol{x})]$$

- ▶ Function  $f$  is expensive to evaluate
- ▶ Integration in moderate ( $\leq 10$ ) dimensions
- ▶ Deal with noisy function observations

## Key idea

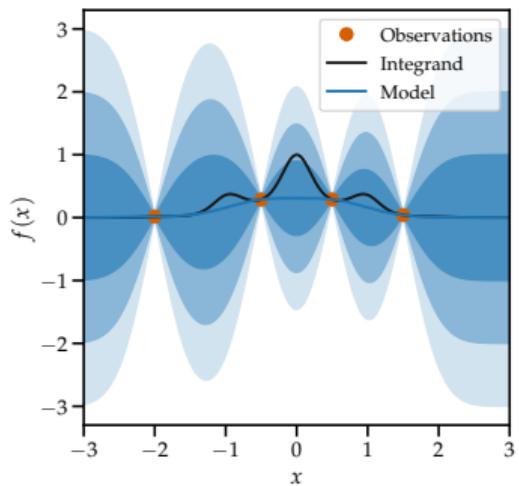
- ▶ Phrase quadrature as a statistical inference problem
  - ▶▶ Probabilistic numerics (e.g., Hennig et al., 2015; Briol et al., 2015)
- ▶ Estimate distribution on  $Z$  using a dataset  $\mathcal{D} := \{(\boldsymbol{x}_1, f(\boldsymbol{x}_1)), \dots, (\boldsymbol{x}_N, f(\boldsymbol{x}_N))\}$

# Bayesian quadrature: How it works

$$Z := \int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \mathbb{E}_{\mathbf{x} \sim p}[f(\mathbf{x})]$$

- ▶ Estimate distribution on  $Z$  using a dataset  
 $\mathcal{D} := \{(\mathbf{x}_1, f(\mathbf{x}_1)), \dots, (\mathbf{x}_N, f(\mathbf{x}_N))\}$
- ▶ Place (Gaussian process) prior distribution on  $f$  and determine the posterior via Bayes' theorem (Diaconis 1988; O'Hagan 1991; Rasmussen & Ghahramani 2003)
  - ▶▶ Distribution on  $f$  induces a distribution on  $Z$
- ▶ Generalizes to noisy function observations

$$y = f(\mathbf{x}) + \epsilon$$



## Bayesian quadrature: Details

$$Z := \int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}, \quad f \sim GP(0, k)$$

- Exploit **linearity of the integral** (integral of a GP is another GP)

$$p(Z) = p\left(\int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}\right) = \mathcal{N}(Z | \mu_Z, \sigma_Z^2)$$

$$\mu_Z = \int \mu_{\text{post}}(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \mathbb{E}_{\mathbf{x}}[\mu_{\text{post}}(\mathbf{x})]$$

$$\sigma_Z^2 = \iint k_{\text{post}}(\mathbf{x}, \mathbf{x}') p(\mathbf{x}) p(\mathbf{x}') d\mathbf{x} d\mathbf{x}' = \mathbb{E}_{\mathbf{x}, \mathbf{x}'}[k_{\text{post}}(\mathbf{x}, \mathbf{x}')]$$

# Bayesian quadrature: Mean

$$\mathbb{E}_f[Z] = \mu_Z = \overbrace{\mathbb{E}_{\mathbf{x} \sim p}[\mu_{\text{post}}(\mathbf{x})]}^{\text{expected predictive mean}}$$

$$\mu_{\text{post}}(\mathbf{x}) = k(\mathbf{x}, \mathbf{X}) \underbrace{\mathbf{K}^{-1} \mathbf{y}}_{=: \boldsymbol{\alpha}}, \quad \mathbf{K} := k(\mathbf{X}, \mathbf{X})$$

$$\mathbb{E}_f[Z] = \overbrace{\int k(\mathbf{x}, \mathbf{X}) p(\mathbf{x}) d\mathbf{x}}^{=: \mathbf{z}^\top} \boldsymbol{\alpha} = \mathbf{z}^\top \boldsymbol{\alpha}$$

$$z_n = \int k(\mathbf{x}, \mathbf{x}_n) p(\mathbf{x}) d\mathbf{x} = \mathbb{E}_{\mathbf{x} \sim p}[k(\mathbf{x}, \mathbf{x}_n)]$$

$$\begin{aligned} Z &= \int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} \\ f &\sim GP(0, k) \\ p(Z) &= \mathcal{N}(Z | \mu_Z, \sigma_Z^2) \\ \text{Training data: } &\mathbf{X}, \mathbf{y} \end{aligned}$$

# Bayesian quadrature: Variance

$$\begin{aligned}\mathbb{V}_f[Z] = \sigma_Z^2 &= \overbrace{\mathbb{E}_{\mathbf{x}, \mathbf{x}' \sim p}[k_{\text{post}}(\mathbf{x}, \mathbf{x}')] }^{\text{expected posterior covariance}} \\ &= \iint \underbrace{k(\mathbf{x}, \mathbf{x}')}_{\text{prior covariance}} - \underbrace{k(\mathbf{x}, \mathbf{X}) \mathbf{K}^{-1} k(\mathbf{X}, \mathbf{x}') p(\mathbf{x}) p(\mathbf{x}') d\mathbf{x} d\mathbf{x}'}_{\text{information from training data}} \\ &= \iint k(\mathbf{x}, \mathbf{x}') p(\mathbf{x}) p(\mathbf{x}') d\mathbf{x} d\mathbf{x}' - \underbrace{\int k(\mathbf{x}, \mathbf{X}) p(\mathbf{x}) d\mathbf{x} \mathbf{K}^{-1} \int k(\mathbf{X}, \mathbf{x}') p(\mathbf{x}') d\mathbf{x}'}_{=\mathbf{z}^\top \quad \quad \quad =\mathbf{z}'} \\ &= \mathbb{E}_{\mathbf{x}, \mathbf{x}'}[k(\mathbf{x}, \mathbf{x}')] - \mathbf{z}^\top \mathbf{K}^{-1} \mathbf{z}' \\ &= \mathbb{E}_{\mathbf{x}, \mathbf{x}'}[k(\mathbf{x}, \mathbf{x}')] - \mathbb{E}_{\mathbf{x}}[k(\mathbf{x}, \mathbf{X})] \mathbf{K}^{-1} \mathbb{E}_{\mathbf{x}'}[k(\mathbf{X}, \mathbf{x}')]\end{aligned}$$

# Computing kernel expectations

$$\mathbb{E}_{\mathbf{x} \sim p}[k(\mathbf{x}, \mathbf{X})], \quad \mathbb{E}_{\mathbf{x}, \mathbf{x}' \sim p}[k(\mathbf{x}, \mathbf{x}')]$$

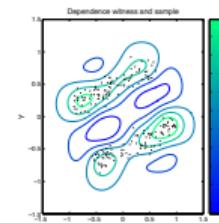
- ▶ Solve a different (easier) integration problem

Kernel $k$	Input distribution $p$	
	Gaussian	non-Gaussian
RBF/ polynomial/ trigonometric	analytical	analytical via importance-sampling trick
otherwise	Monte Carlo (numerical integration)	Monte Carlo (numerical integration)

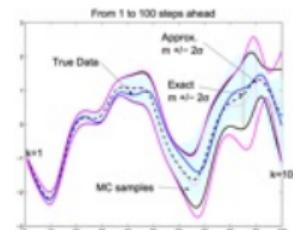
# Kernel expectations in other areas

$$\mathbb{E}_{\mathbf{x} \sim p}[k(\mathbf{x}, \mathbf{X})], \quad \mathbb{E}_{\mathbf{x}, \mathbf{x}' \sim p}[k(\mathbf{x}, \mathbf{x}')]$$

- ▶ Kernel MMD  
(e.g., Gretton et al., 2012)
- ▶ Time-series analysis with Gaussian processes  
(e.g., Girard et al., 2003)
- ▶ Deep Gaussian processes  
(e.g., Damianou & Lawrence, 2013)
- ▶ Model-based RL with Gaussian processes  
(e.g., Deisenroth & Rasmussen, 2011)



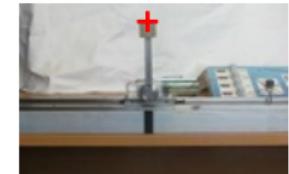
from Gretton et al. (2012)



from Girard et al. (2003)



from Salimbeni et al. (2019)



from Deisenroth & Rasmussen (2011)

## Iterative procedure: Where to measure $f$ next?

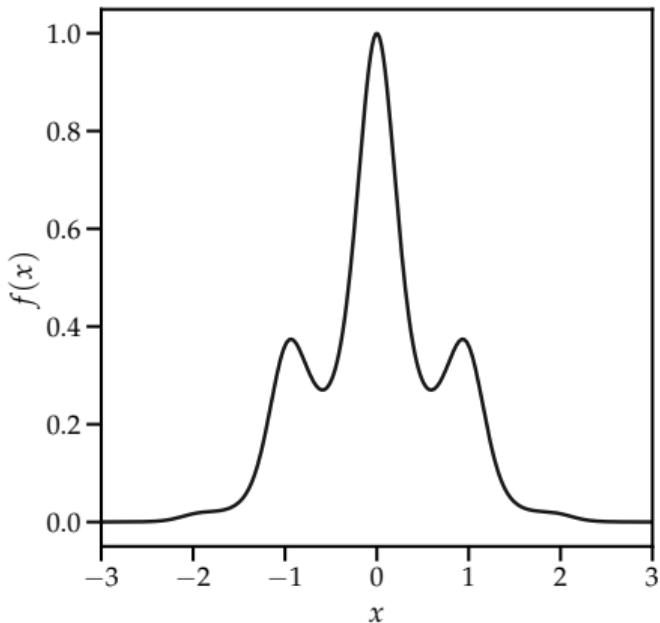
- ▶ Define an **acquisition function** (similar to Bayesian optimization)
- ▶ Example: Choose next node  $x_{n+1}$  so that the **variance of the estimator** is reduced **maximally** (e.g., O'Hagan, 1991; Gunter et al., 2014)

$$x_{n+1} = \operatorname{argmax}_{x_*} \underbrace{\mathbb{V}[Z|\mathcal{D}]}_{\text{current variance}} - \mathbb{E}_{y_*} \left[ \mathbb{V}[Z|\mathcal{D} \cup \{(x_*, y_*)\}] \right]$$

## Example with EmuKit (Paleyès et al., 2019)

Compute

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

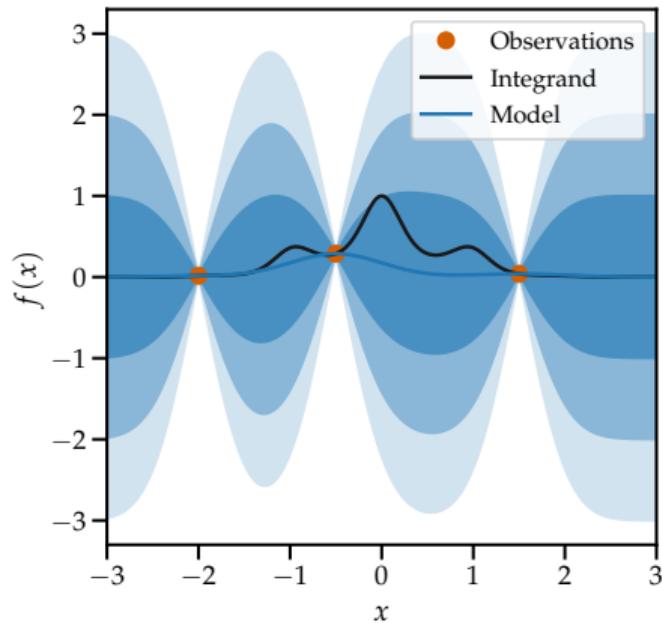


## Example with EmuKit (Paley et al., 2019)

Compute

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

- ▶ Fit Gaussian process to observations  $f(x_1), \dots, f(x_n)$  at nodes  $x_1, \dots, x_n$

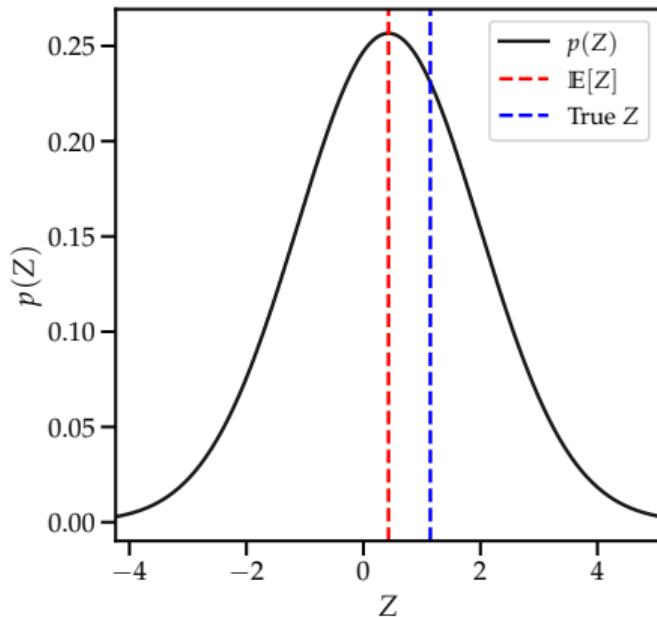


## Example with EmuKit (Paley et al., 2019)

Compute

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

- ▶ Fit Gaussian process to observations  $f(x_1), \dots, f(x_n)$  at nodes  $x_1, \dots, x_n$
- ▶ Determine  $p(Z)$

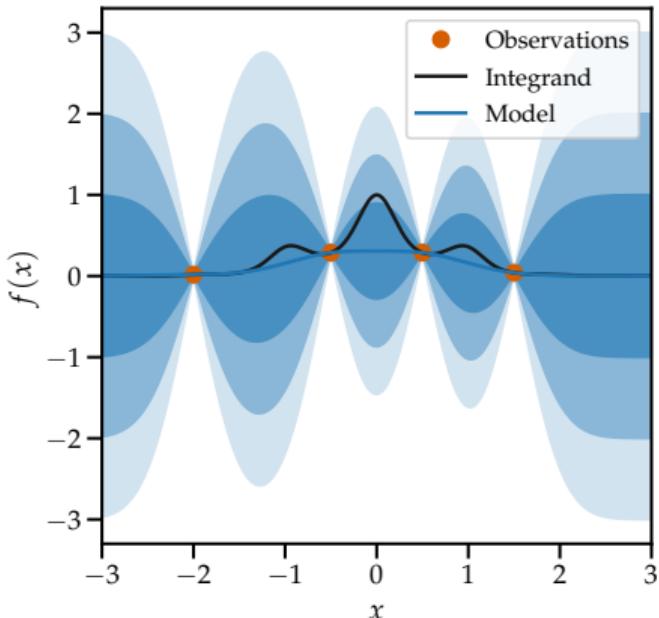


## Example with EmuKit (Paley et al., 2019)

Compute

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

- ▶ Fit Gaussian process to observations  $f(x_1), \dots, f(x_n)$  at nodes  $x_1, \dots, x_n$
- ▶ 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  $(x_{n+1}, f(x_{n+1}))$

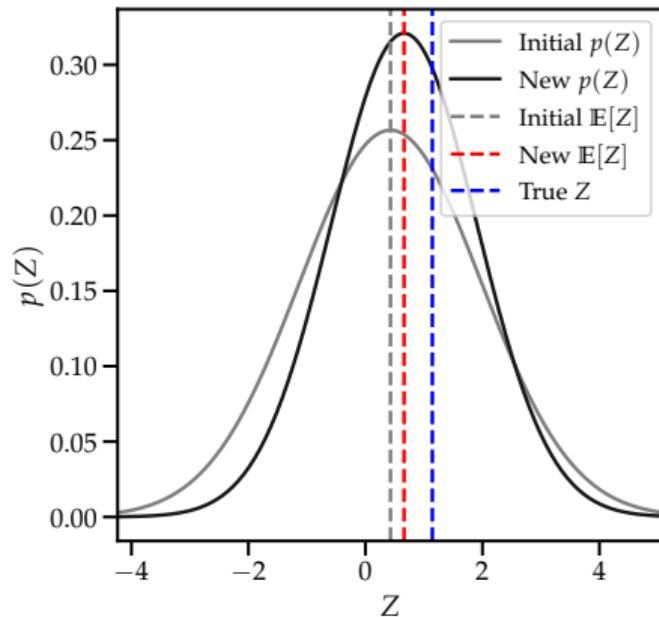


## Example with EmuKit (Paley et al., 2019)

Compute

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

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

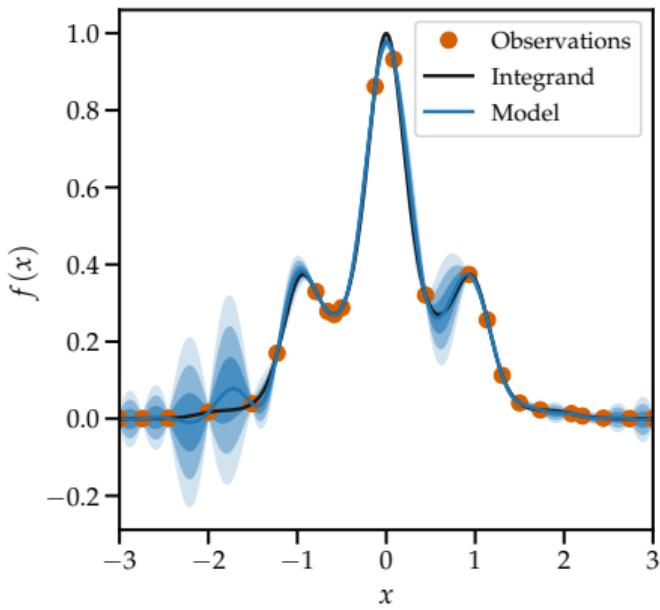


# Example with EmuKit (Paley et al., 2019)

Compute

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

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

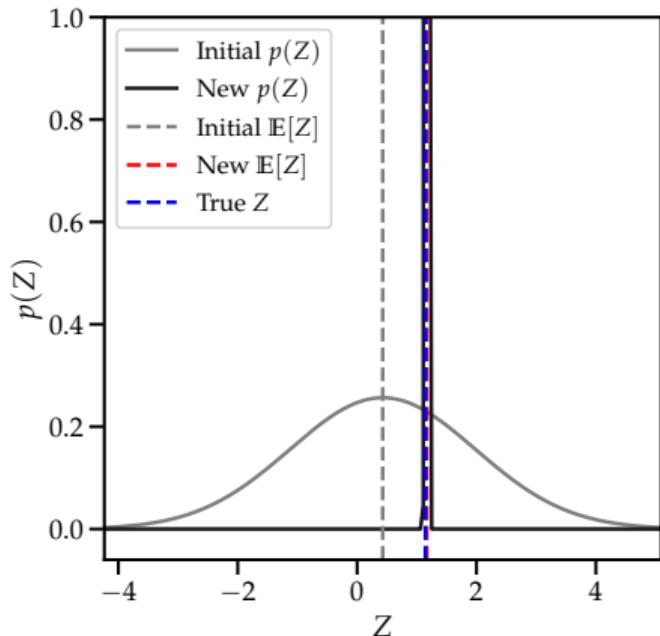


## Example with EmuKit (Paley et al., 2019)

Compute

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

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



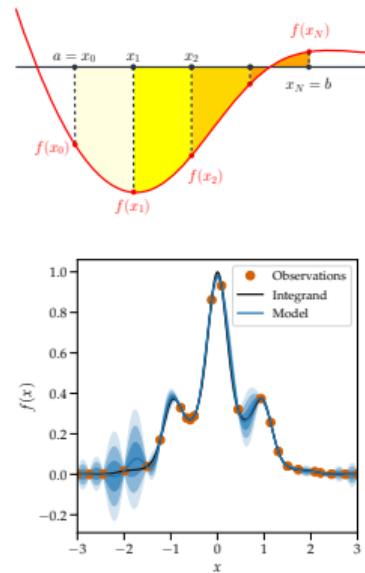
# Summary

- ▶ Central approximation

$$\int f(\mathbf{x}) d\mathbf{x} \approx \sum_{n=1}^N w_n f(\mathbf{x}_n)$$

- ▶ **Newton–Cotes:** Equidistant nodes  $\mathbf{x}_n$ , low-degree polynomial approximation of  $f$
- ▶ **Gaussian quadrature:** Nodes  $\mathbf{x}_n$  as the roots of interpolating orthogonal polynomials of  $f$
- ▶ **Bayesian quadrature:** Integration as a statistical inference problem; Global approximation of  $f$  using a Gaussian process; scales to moderate dimensions

►► Numerical integration is a really good idea in low dimensions



# Inference in Time Series

Cheng Soon Ong  
Marc Peter Deisenroth

December 2020



# Setting

- ▶ Time-series model

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t) + \boldsymbol{\epsilon}, \quad \boldsymbol{x}_0 \sim p(\boldsymbol{x}_0), \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q})$$

- ▶ Compute an expected utility of a state trajectory  $\tau := (\boldsymbol{x}_0, \dots, \boldsymbol{x}_T)$

$$\mathbb{E}_\tau[U(\tau)]$$

- ▶ Reinforcement learning and optimal control
- ▶ Demand forecasting (logistics)
- ▶ Weather/climate forecasts
- ▶ Challenge: Long-term predictions and uncertainty propagation

# Approaches

## ► Deterministic inference via iterative computation

- ▶ Iteratively determine marginal distributions  $p(\mathbf{x}_1), \dots, p(\mathbf{x}_T)$
- ▶ Compute expectations  $\mathbb{E}_{\mathbf{x}_t}[u(\mathbf{x}_t)]$  and compute utilities of the form

$$\mathbb{E}_{\boldsymbol{\tau}}[U(\boldsymbol{\tau})] = \sum_{t=0}^T \mathbb{E}_{\mathbf{x}_t}[u(\mathbf{x}_t)] = \sum_{t=0}^T \int u(\mathbf{x}_t) p(\mathbf{x}_t) d\mathbf{x}_t$$

## ► Stochastic inference via trajectory sampling

- ▶ Generate sample trajectories  $\boldsymbol{\tau}^{(s)} = (\mathbf{x}_0^{(s)}, \dots, \mathbf{x}_T^{(s)})$
- ▶ Monte-Carlo integration

$$\mathbb{E}_{\boldsymbol{\tau}}[U(\boldsymbol{\tau})] \approx \frac{1}{S} \sum_{s=1}^S U(\boldsymbol{\tau}^{(s)})$$

# Deterministic Approximate Inference

# Deterministic approximate inference

- ▶ Iteratively compute marginals

$$\begin{aligned} p(\mathbf{x}_{t+1}) &= \int p(\mathbf{x}_{t+1}|\mathbf{x}_t)p(\mathbf{x}_t)d\mathbf{x}_t \\ &= \int \mathcal{N}(f(\mathbf{x}_t), \mathbf{Q})p(\mathbf{x}_t)d\mathbf{x}_t \end{aligned}$$

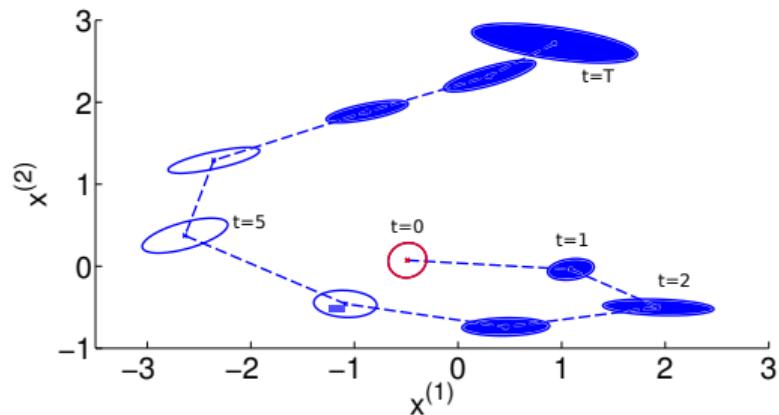
$$\mathbf{x}_{t+1} = f(\mathbf{x}_t) + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$$

►► No closed-form solution for nonlinear  $f$

# Iterative Gaussian approximation

- ▶ Common approach: Iterative Gaussian approximation of marginals:

$$p(\mathbf{x}_t) \approx \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$$



- ▶ Linearization
- ▶ Unscented transformation
- ▶ Moment matching
- ▶ Extended Kalman filter
- ▶ Unscented Kalman filter
- ▶ Assumed density filter

## Two approaches

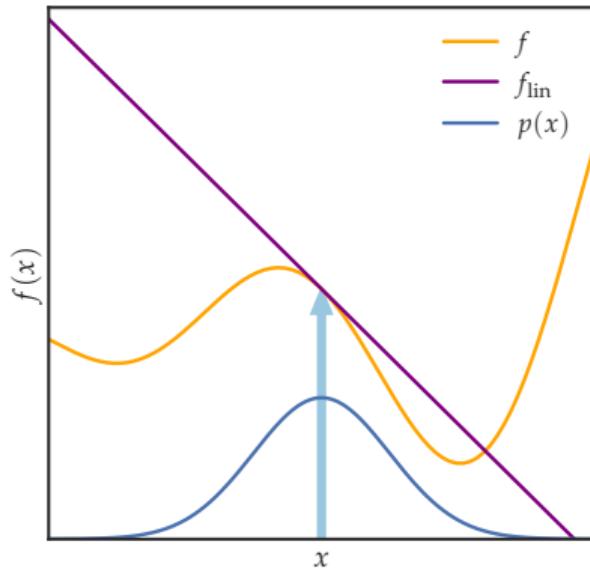
$$p(\mathbf{x}_{t+1}) = \int \mathcal{N}(\mathbf{x}_{t+1} | f(\mathbf{x}_t), \mathbf{Q}) p(\mathbf{x}_t) d\mathbf{x}_t \approx \mathcal{N}(\mathbf{x}_{t+1} | \boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1})$$

- ▶ Approximate  $f$       ►► Linearization (e.g., Smith et al., 1962)
- ▶ Approximate  $p(\mathbf{x}_t)$       ►► Unscented transformation (Julier & Uhlmann, 1995)

# Approach 1: Linearization

Key idea (e.g., Smith et al., 1962; Ohab & Stubberud, 1965)

1. Locally linearize  $f$  around mean  $\mu_t$
2. Compute predictive distribution (Gaussian) for linearized function in closed form



- ▶ Linearization: First-order Taylor-series expansion around  $\mu_t$ 
  - ▶▶ Gradient (Jacobian)  $df/dx_t$  of  $f$  evaluated at  $\mu_t$
- ▶ Key insight: **Gaussians can be pushed through linear functions in closed form**

# How it works

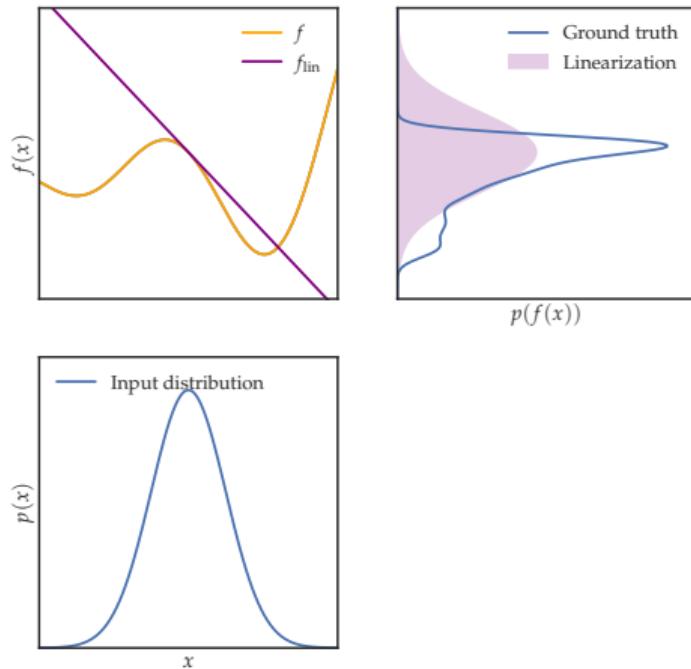
- ▶ Compute gradient  $\mathbf{J}_t := df/d\mathbf{x}_t|_{\mathbf{x}_t=\mu_t}$
- ▶ Linearized model:

$$f(\mathbf{x}) \approx f(\boldsymbol{\mu}_t) + \mathbf{J}_t(\mathbf{x} - \boldsymbol{\mu}_t)$$

- ▶ Approximate predictive distribution is Gaussian:

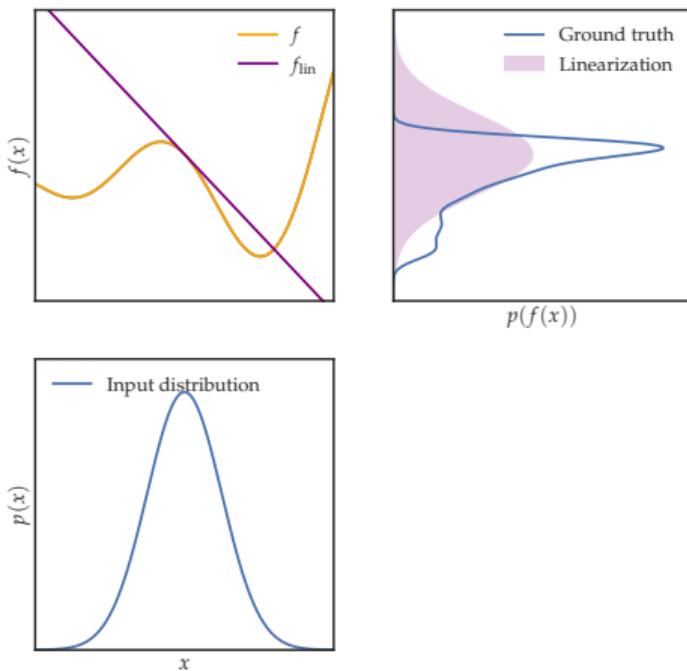
$$p(f(\mathbf{x}_t)) \approx \mathcal{N}(f(\boldsymbol{\mu}_t), \mathbf{J}_t \boldsymbol{\Sigma}_t \mathbf{J}_t^\top)$$

$$p(\mathbf{x}_{t+1}) \approx \mathcal{N}(f(\boldsymbol{\mu}_t), \mathbf{J}_t \boldsymbol{\Sigma}_t \mathbf{J}_t^\top + \mathbf{Q})$$



# Linearization: Properties

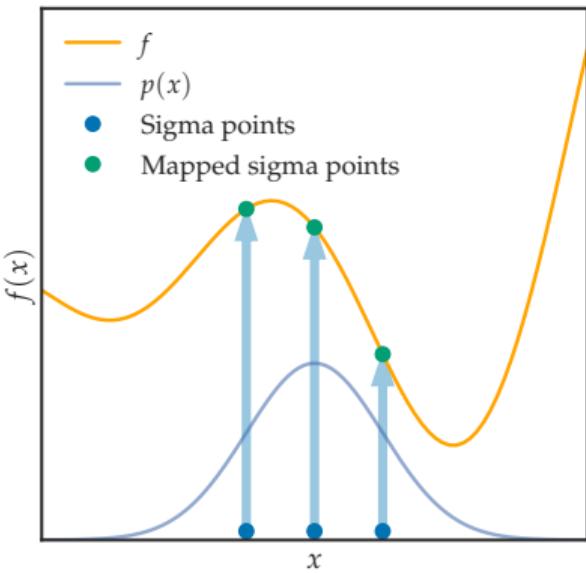
- ▶ Conceptually straightforward
- ▶ Requires differentiable  $f$
- ▶ Tends to underestimate true covariance matrix ➡ Overconfidence
- ▶ Scales cubically in the dimension of  $x$
- ▶ Widely used in engineering (e.g., navigation systems, GPS, Apollo missions)



## Approach 2: Unscented transformation

Key idea (Julier & Uhlmann, 1995)

1. Approximate  $p(\mathbf{x}_t)$  using a small set of deterministically chosen sigma points
2. Map sigma points through  $f$
3. Compute a weighted average of the mean and covariance of the predictive distribution.

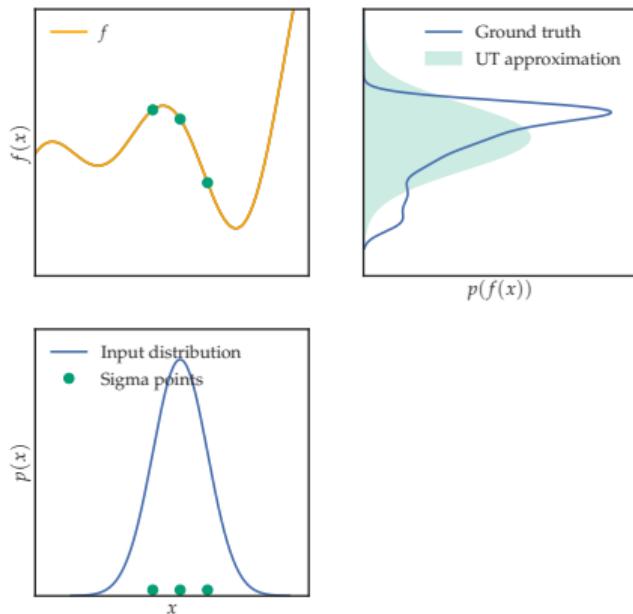


# How it works

- ▶ Determine  $2D + 1$  sigma points  
 $\mathcal{X}_t = \{\boldsymbol{\mu}_t \pm \alpha(\sqrt{\Sigma_t})_i, i = 1, \dots, D\}$
- ▶ Map sigma points through  $f$  to get  $f(\mathcal{X}_t)$
- ▶ Compute mean/covariance of predictive distribution  $p(f(\mathbf{x}_t))$  as a weighted average

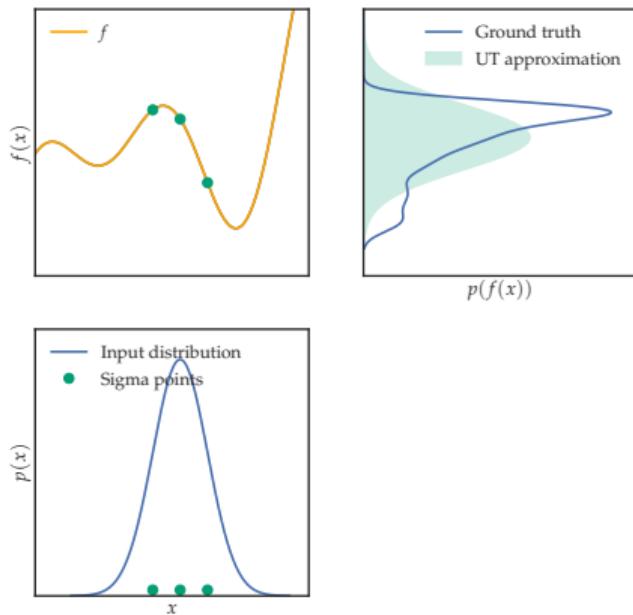
$$\boldsymbol{\mu}_{t+1} \approx \sum_{d=1}^{2D+1} w_d^\mu f(\mathcal{X}_t^{(d)})$$

$$\boldsymbol{\Sigma}_{t+1} \approx \sum_{d=1}^{2D+1} w_d^\Sigma (f(\mathcal{X}_t^{(d)}) - \boldsymbol{\mu}_{t+1})(f(\mathcal{X}_t^{(d)}) - \boldsymbol{\mu}_{t+1})^\top$$



# Unscented transformation: Properties

- ▶ Not a Monte-Carlo method: Sigma points are deterministic, not random
- ▶ No explicit calculation of Jacobians
  - ▶▶  $f$  can be non-differentiable
- ▶ Input distribution does not need to be Gaussian
- ▶ Higher accuracy (covariance) than linearization  
(Julier & Uhlmann, 2004)



# Stochastic Approximate Inference

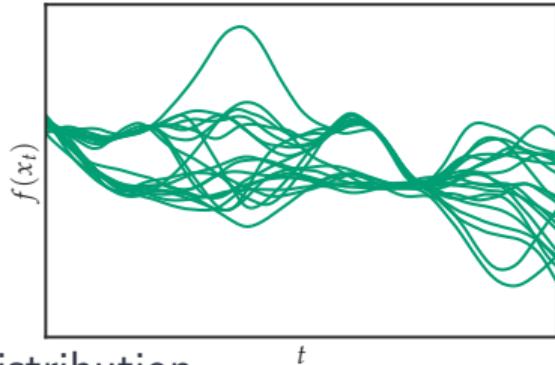
# Stochastic approximate inference

- ▶ Sample trajectories  $\tau^{(i)} = (\mathbf{x}_0^{(i)}, \dots, \mathbf{x}_T^{(i)})$ :

1. Sample initial state:  $\mathbf{x}_0^{(i)} \sim p(\mathbf{x}_0)$
2. For  $t = 1, \dots, T$ :

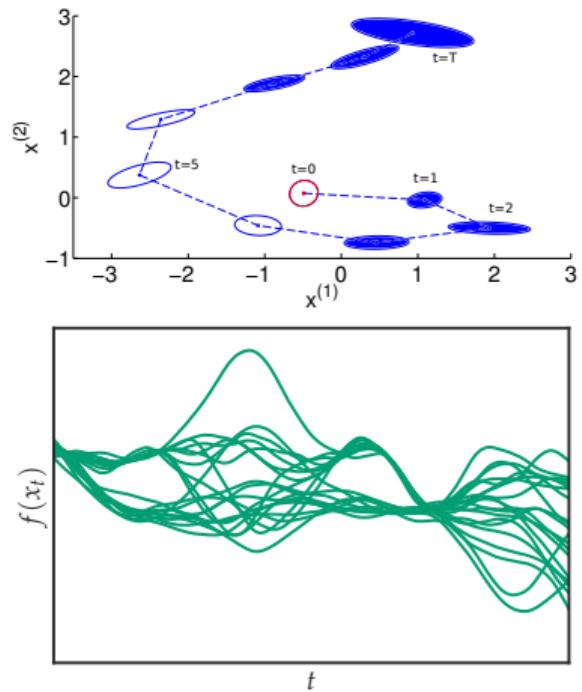
$$\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t | \mathbf{x}_{t-1}^{(i)}) = \mathcal{N}(\mathbf{x}_t | f(\mathbf{x}_{t-1}^{(i)}), \mathbf{Q})$$

- ▶ No parametric restriction to a specific kind of distribution
- ▶ Have to store all samples (particles) ➡ Potential memory issue
- ▶ **Sequential Monte Carlo** (particle filter)  
(e.g., Doucet et al., 2000; Thrun et al., 2005;)



# Discussion: Long-term predictions

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



# Inference in Gaussian Process Time Series Models

## Setting

- ▶ Time-series model

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t) + \boldsymbol{\epsilon}, \quad \boldsymbol{x}_0 \sim p(\boldsymbol{x}_0), \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}), \quad f \sim GP(\mu, k)$$

- ▶ Two approaches for long-term predictions:

- ▶ Deterministic approximate inference of the marginals  $p(\boldsymbol{x}_1), \dots, p(\boldsymbol{x}_T)$
- ▶ Stochastic approximate inference by sampling trajectories

$$\boldsymbol{\tau}^{(i)} = (\boldsymbol{x}_0^{(i)}, \dots, \boldsymbol{x}_T^{(i)})$$

# Deterministic approximate inference

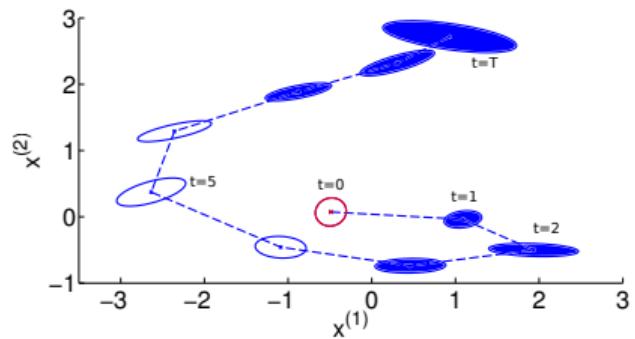
$$\begin{aligned} p(\mathbf{x}_{t+1}) &= \int p(\mathbf{x}_{t+1}|\mathbf{x}_t)p(\mathbf{x}_t)d\mathbf{x}_t \\ &= \int \left[ \int p(f(\mathbf{x}_t)|f, \mathbf{x}_t)p(f)df \right] p(\mathbf{x}_t)d\mathbf{x}_t \end{aligned}$$

Approaches:

- ▶ Linearization (e.g., Ko & Fox, 2009)
- ▶ Unscented transformation (e.g., Ko & Fox, 2009)
- ▶ Moment matching (e.g., Deisenroth et al., 2009)

# Long-term predictions

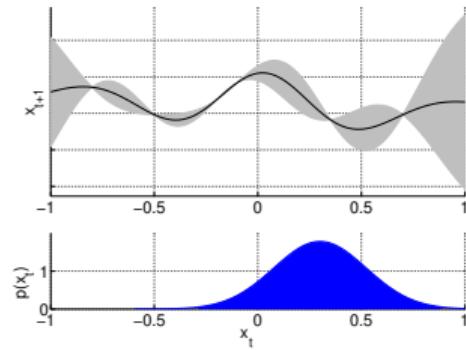
- ▶ Iteratively compute  $p(\mathbf{x}_1), \dots, p(\mathbf{x}_T)$



# Long-term predictions

- ▶ Iteratively compute  $p(\mathbf{x}_1), \dots, p(\mathbf{x}_T)$

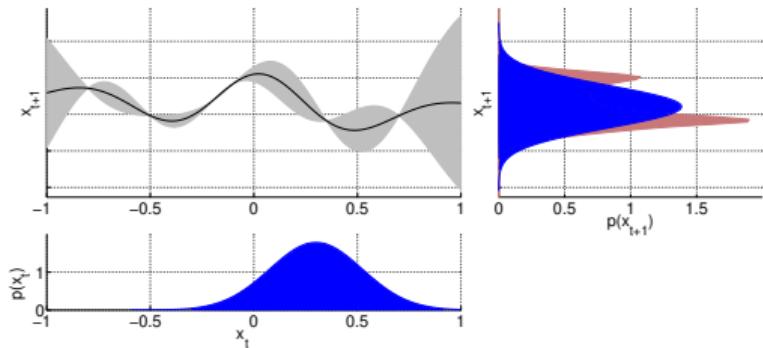
$$\begin{array}{c} p(\mathbf{x}_{t+1} | \mathbf{x}_t) \\ \text{GP prediction} \end{array} \quad \begin{array}{c} p(\mathbf{x}_t) \\ \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \end{array}$$



# Long-term predictions

- ▶ Iteratively compute  $p(\mathbf{x}_1), \dots, p(\mathbf{x}_T)$

$$p(\mathbf{x}_{t+1}) = \iint \underbrace{p(\mathbf{x}_{t+1}|\mathbf{x}_t)}_{\text{GP prediction}} \underbrace{p(\mathbf{x}_t)}_{\mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)} df d\mathbf{x}_t$$



- ▶ GP moment matching (Girard et al., 2003; Quiñonero-Candela et al., 2003)
- ▶ Key ingredient: Computing kernel expectations

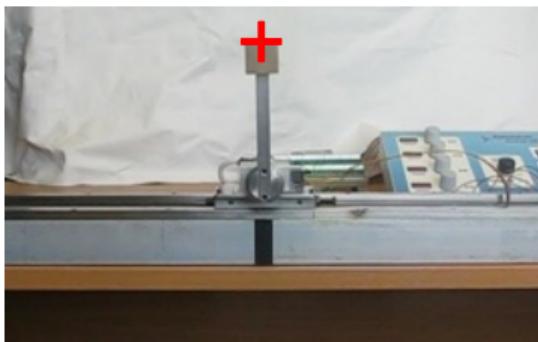
# Example: Model-based reinforcement learning

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

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

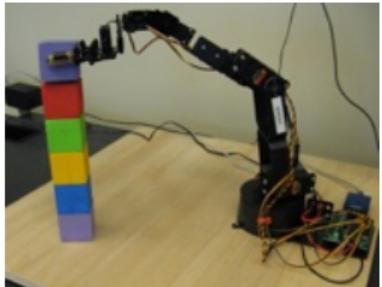
- ▶ GP moment matching for long-term predictions
- ▶ Gradient descent to find  $\theta^*$

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

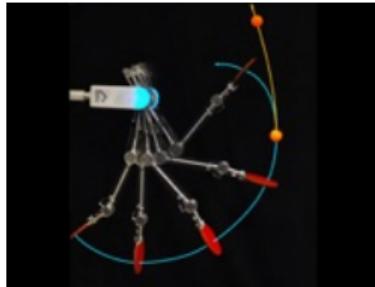


From Deisenroth & Rasmussen (2011)  
<https://www.youtube.com/PilcoLearner>

# Wide Applicability



Deisenroth et al. (2011)



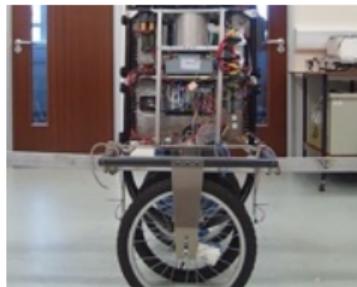
Englert et al. (2013)



Kupcsik et al. (2017)



Bischoff et al. (2013b)



McHutchon (2014)



Bischoff et al. (2013a)

► Application to a wide range of robotic systems

# Stochastic approximate inference

- ▶ Generating a function draw: Sampling from a  $T$ -dimensional multivariate Gaussian  
 $T$ : Number of query points

Figure: Generated with GPflow (Matthews et al., 2017)

# Stochastic approximate inference

- ▶ Generating a function draw: Sampling from a  $T$ -dimensional multivariate Gaussian  
 $T$ : Number of query points
  - ▶ Drawing a sample from a GP **scales cubically in  $T$**
  - ▶ There are some ways around this in low dimensions (e.g., Särkkä et al., 2013; Solin et al., 2018) or by making structural assumptions (e.g., Pleiss et al., 2018)
- ▶▶ Let's try something else

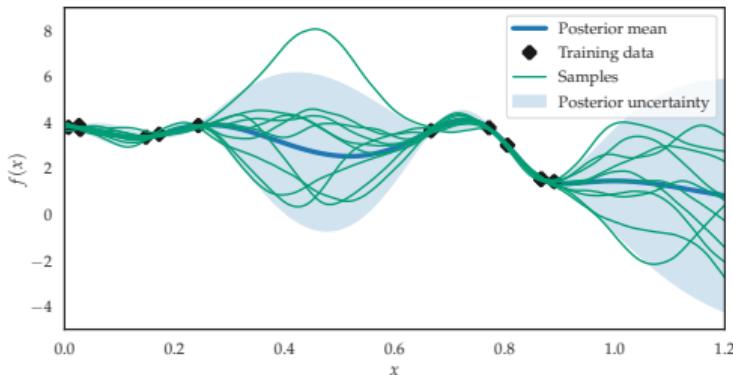


Figure: Generated with GPflow (Matthews et al., 2017)

# Decoupled sampling (Wilson et al., 2020a)

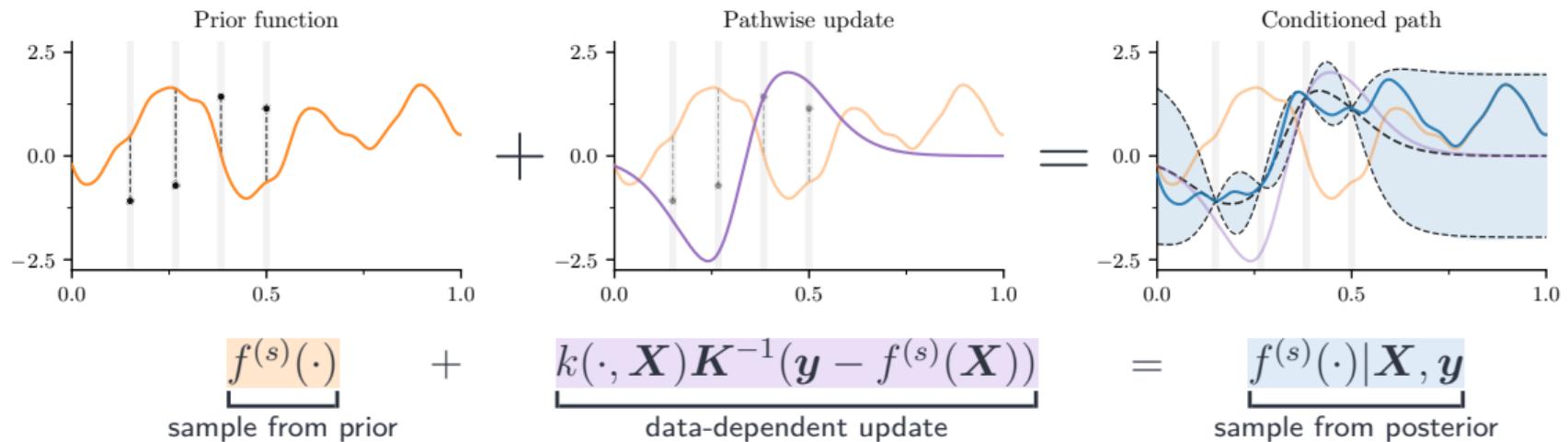
## Key idea

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

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

- ▶ 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

# Illustration: Decoupled sampling (Wilson et al., 2020a)



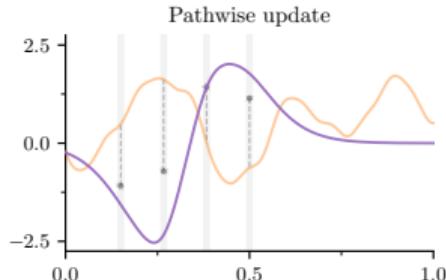
1. Sample from the prior
2. Add data-dependent update term

► Sample from the posterior

# Properties

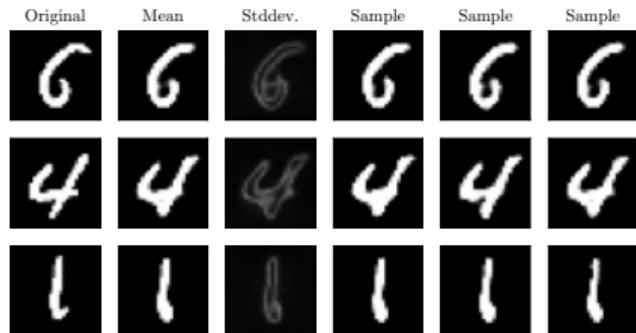
$$f^{(s)}(\cdot) + k(\cdot, \mathbf{X}) \mathbf{K}^{-1} (\mathbf{y} - f^{(s)}(\mathbf{X})) = f^{(s)}(\cdot | \mathbf{X}, \mathbf{y})$$

sample from prior      data-dependent update      sample from posterior



- ▶ Update term depends on error/residual between the prior sample and data  $\mathbf{y}$
- ▶ 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)

# Applications



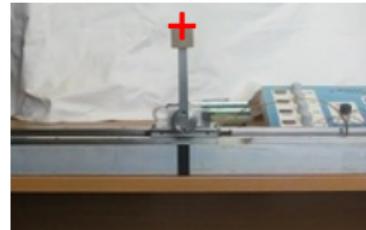
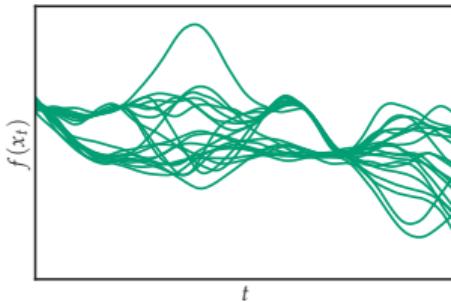
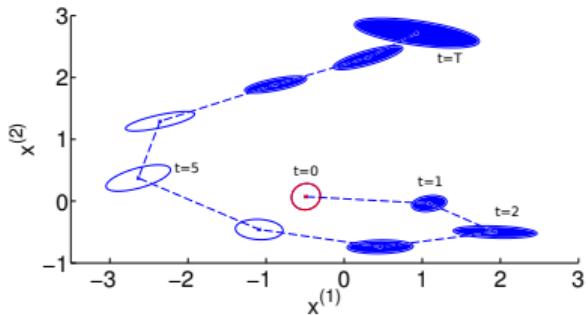
From Wilson et al. (2020b)



From Borovitskiy et al. (2020)

- ▶ Deep convolutional GP auto-encoders (Wilson et al., 2020b)
- ▶ Bayesian optimization with Thompson sampling (Wilson et al., 2020a)
- ▶ Sampling from GPs on manifolds (Borovitskiy et al., 2020)
- ▶ Model-based reinforcement learning

# 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

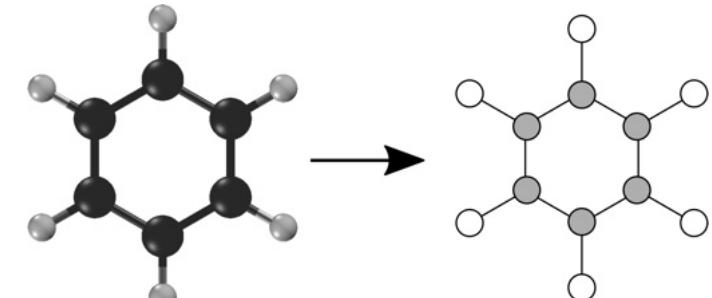
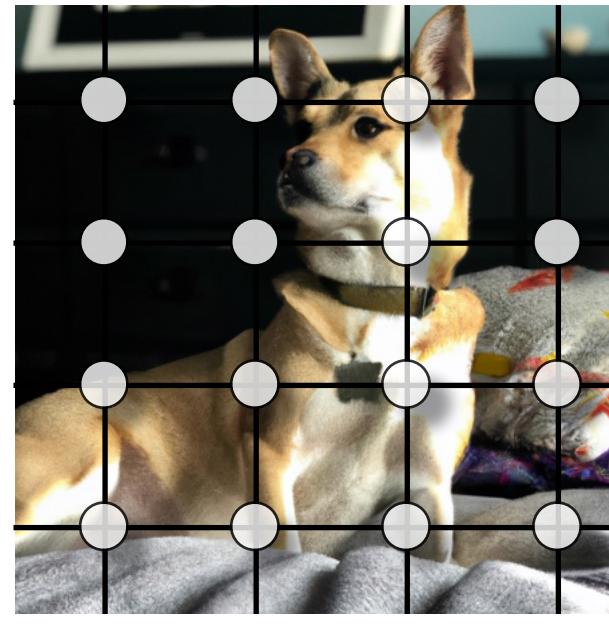
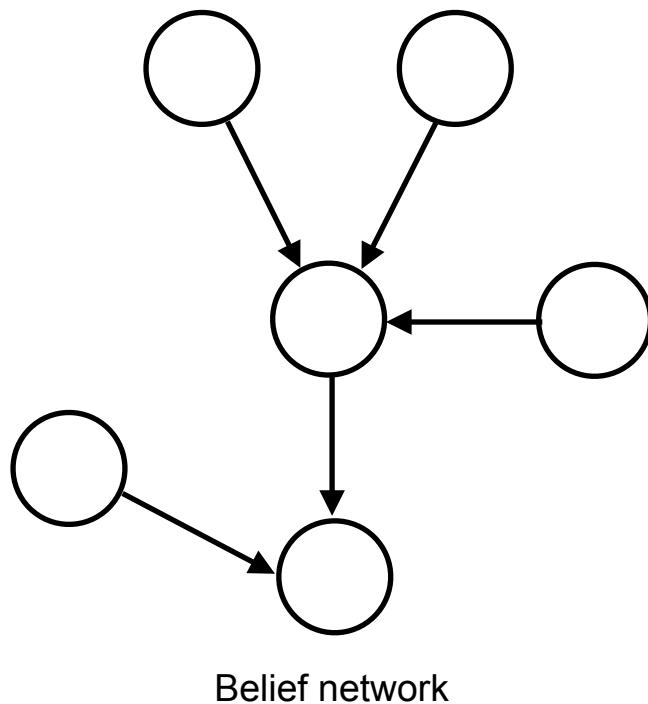
# Message Passing Algorithms in Machine Learning

So Takao

[so.takao@ucl.ac.uk](mailto:so.takao@ucl.ac.uk)  
[www.sotakao.com](http://www.sotakao.com)

# What we will cover in this lecture

We will study machine learning algorithms on **graphs**



Molecules



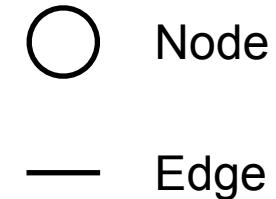
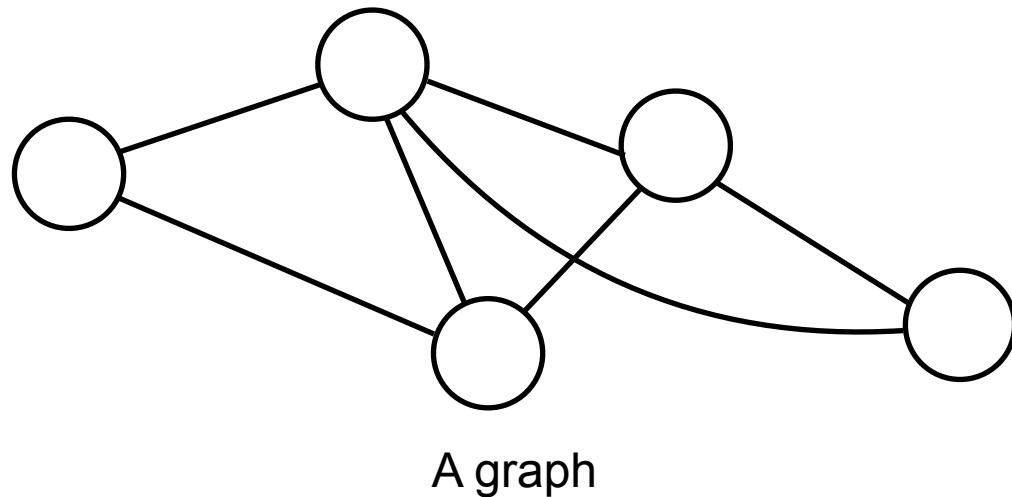
Social networks

# What are graphs?

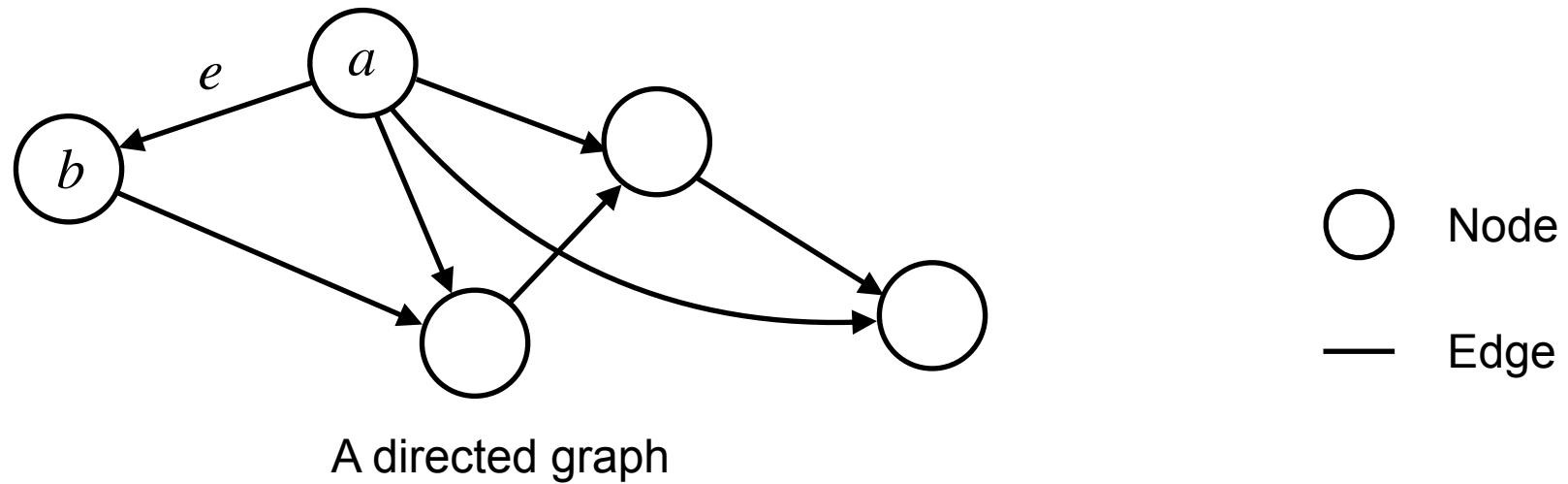
A **Graph** is a collection  $(V, E)$  of

- $V$ : nodes
- $E$ : edges

such that an edge  $e \in E$  can be associated with a pair of nodes  $u, v \in V$ .



- A graph is *directed* if the ordering of nodes associated to an edge “matters”  
i.e.,  $\exists \phi : E \rightarrow V \times V$  mapping an edge to an *ordered tuple* of nodes.

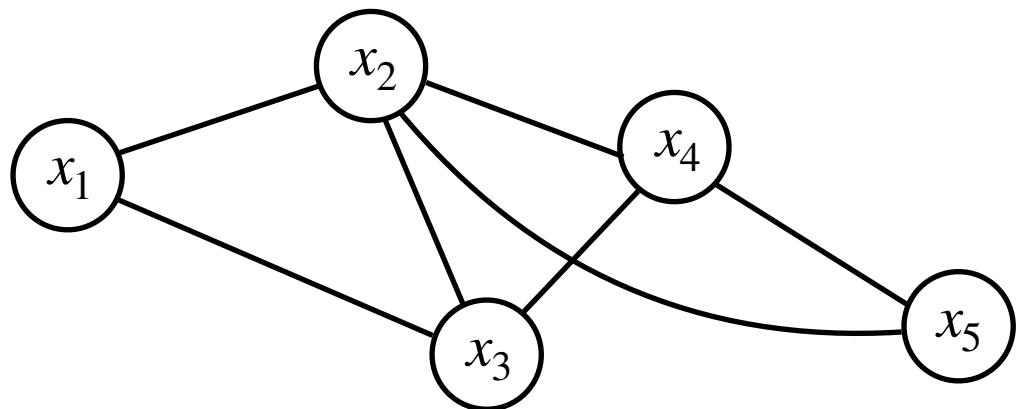


- Edges  $\phi(e) = (a, b)$  in a directed graph represented graphically as arrows
- A graph is *undirected* if ordering of nodes in an edge doesn't matter

- The edges  $E$  of a graph define an **adjacency relation**  $\sim$  on  $V$ :

For  $x, y \in V$ ,

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



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

- $x_1 \sim x_2$
- $x_4 \sim x_5$
- $x_1 \not\sim x_4$
- $x_3 \not\sim x_5$

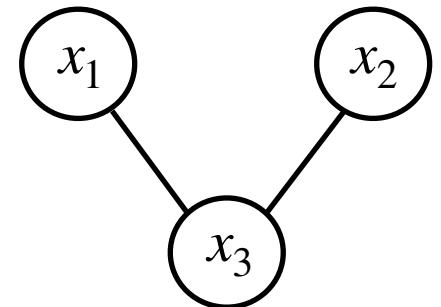
- If  $x \sim y$ , we say that  $y$  is a *neighbour* of  $x$  and vice versa

- Adjacency matrix  $\mathbf{A}$  encodes the adjacency structure of  $G$ :

$$\mathbf{A}_{ij} = \begin{cases} 1, & \text{if } x_i \sim x_j, \\ 0, & \text{if } x_i \not\sim x_j. \end{cases}$$

- Degree matrix  $\mathbf{D}$  encodes the degree of connectivity of each node:

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

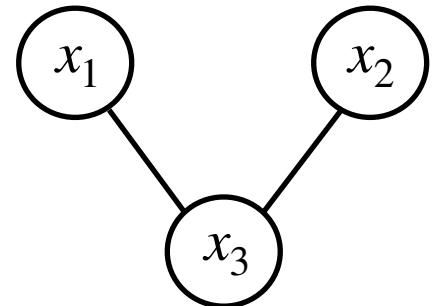


- Adjacency matrix  $\mathbf{A}$  encodes the adjacency structure of  $G$ :

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

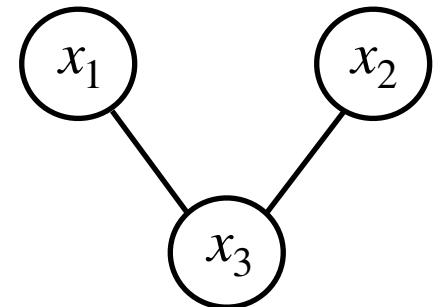
- Degree matrix  $\mathbf{D}$  encodes the degree of connectivity of each node:

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



- Adjacency matrix  $\mathbf{A}$  encodes the adjacency structure of  $G$ :

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

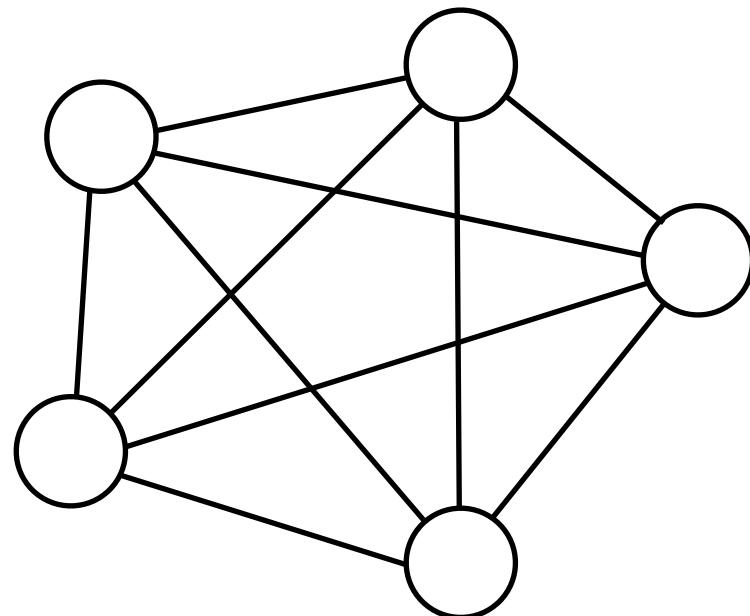


- Degree matrix  $\mathbf{D}$  encodes the degree of connectivity of each node:

$$\mathbf{D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

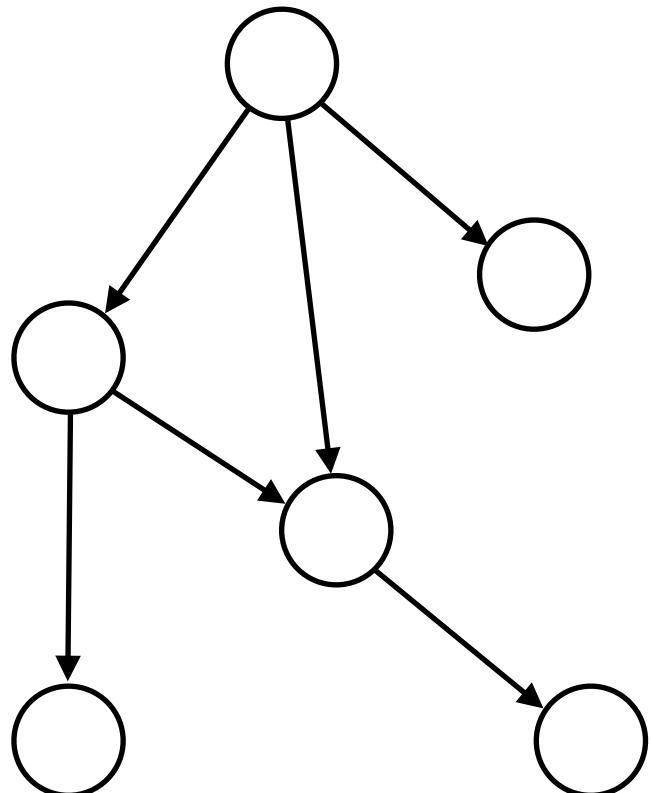
# Types of Graphs

## 1. Fully-connected graphs



- Undirected
- Each node is connected to every other nodes

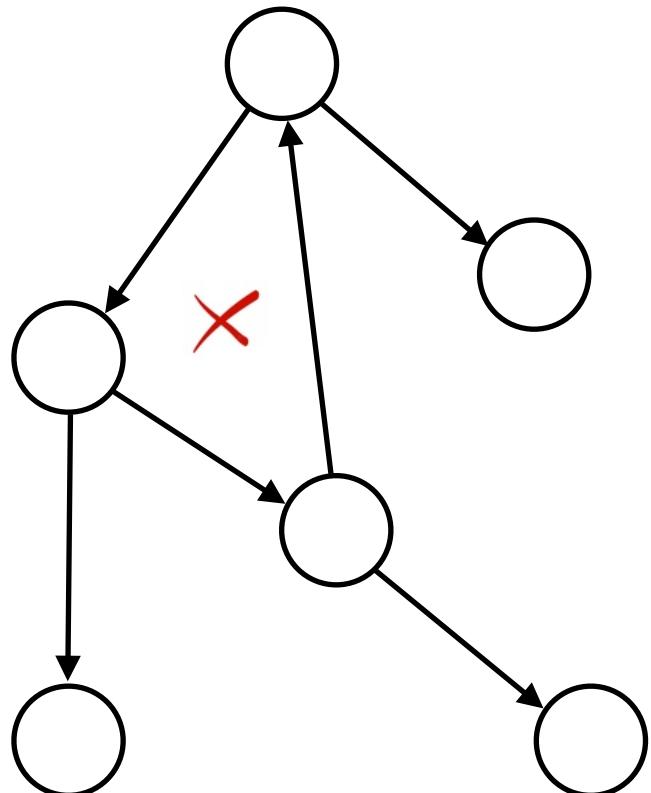
# Types of Graphs



## 2. Directed Acyclic Graph (DAG)

- Directed
- Does not contain any *directed* cycles

# Types of Graphs

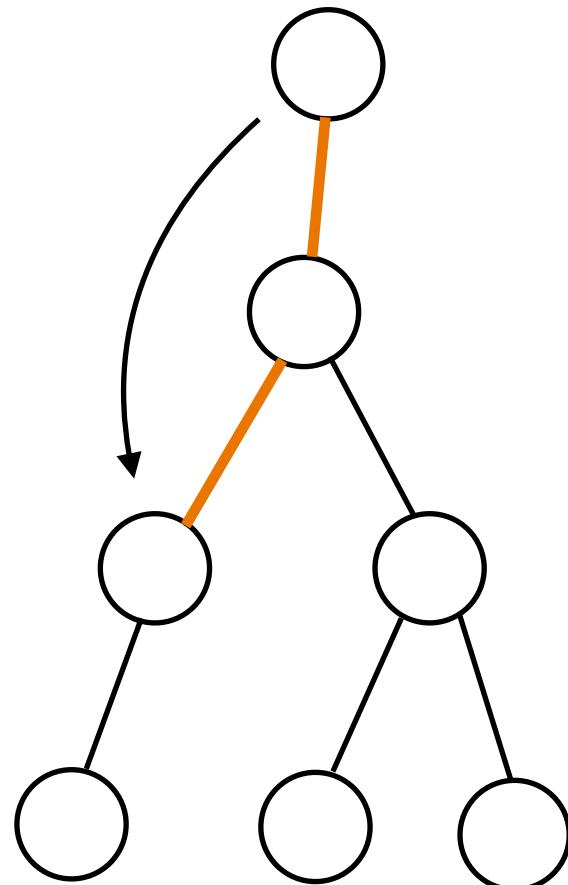


## 2. Directed Acyclic Graph (DAG)

- Directed
- Does not contain any *directed* cycles

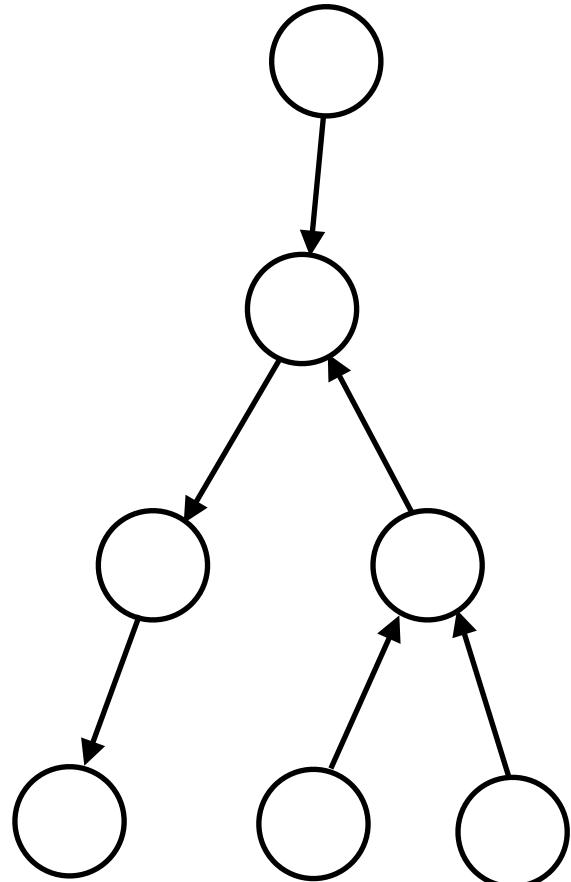
# Types of Graphs

## 3. Trees and polytrees



- A tree is an *undirected* graph such that two nodes are connected by a unique path

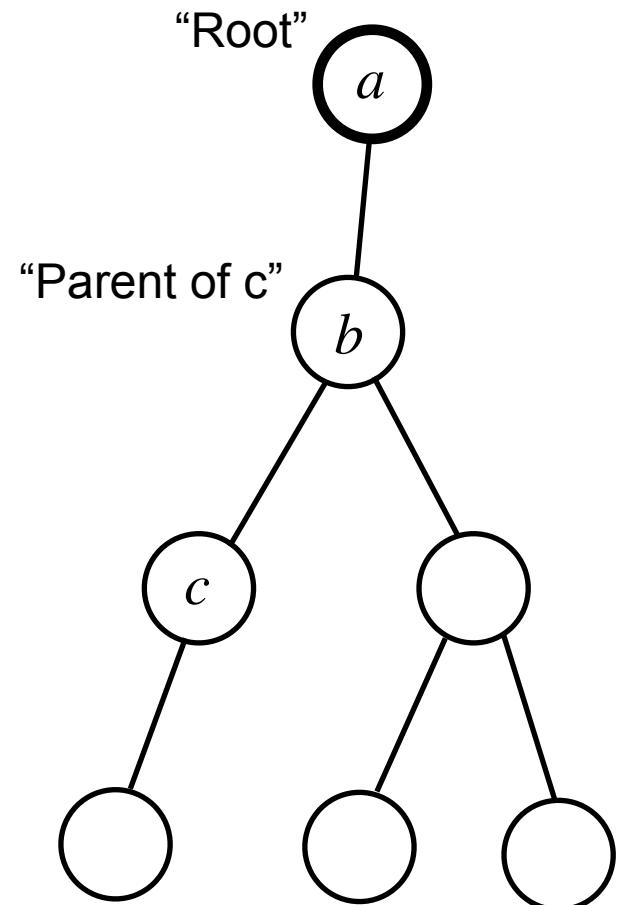
# Types of Graphs



## 3. Trees and polytrees

- A tree is an *undirected* graph such that two nodes are connected by a unique path
- A polytree is a *DAG* such that its underlying structure is a tree

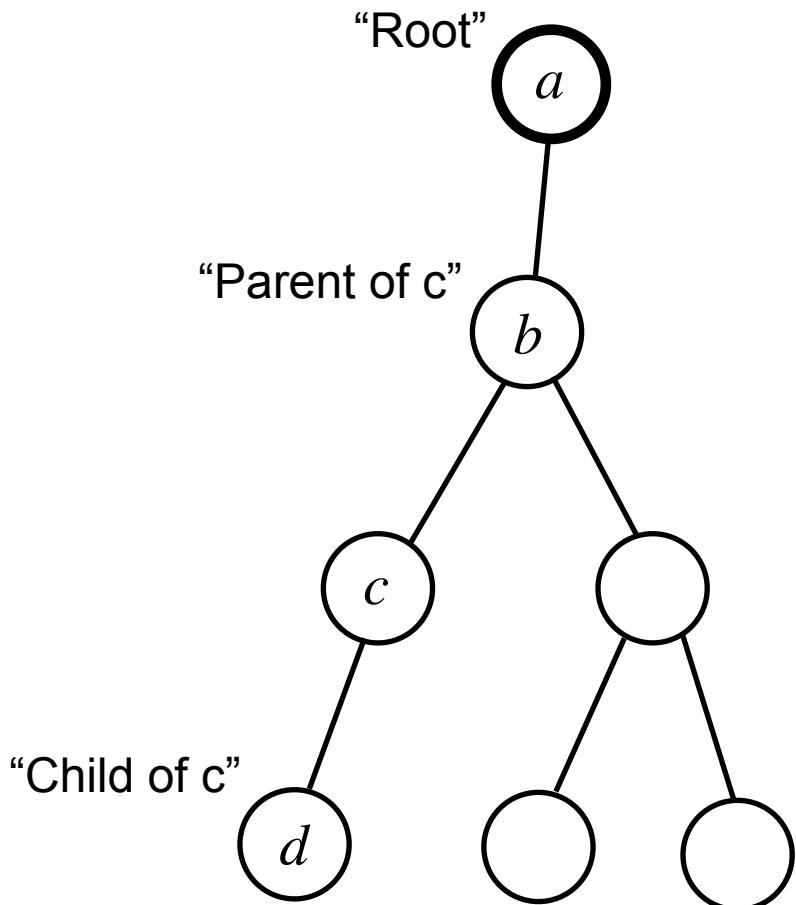
# Types of Graphs



## 3. Trees and polytrees

- A tree is an *undirected* graph such that two nodes are connected by a unique path
- A polytree is a *DAG* such that its underlying structure is a tree
- Designating node  $a$  as a “root”, we say that node  $b$  is a *parent* of node  $c$  if it is a neighbouring node *on the path to  $a$*

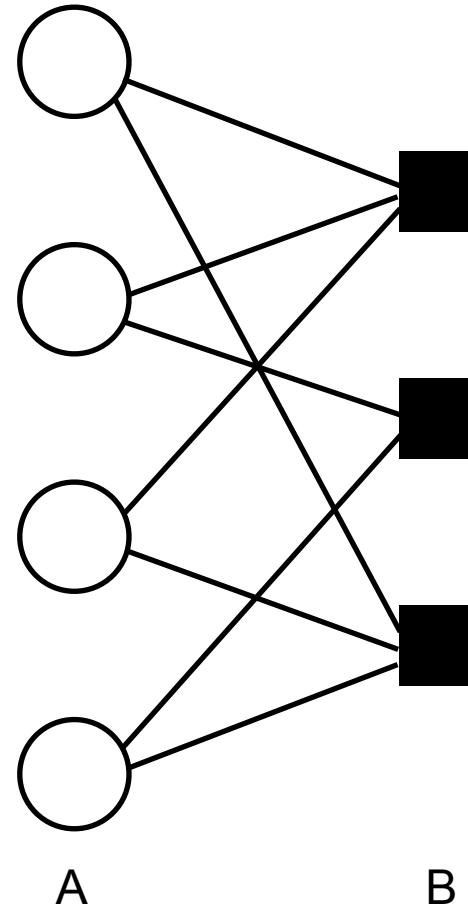
# Types of Graphs



## 3. Trees and polytrees

- A tree is an *undirected* graph such that two nodes are connected by a unique path
- A polytree is a *DAG* such that its underlying structure is a tree
- Designating node  $a$  as a “root”, we say that node  $b$  is a *parent* of node  $c$  if it is a neighbouring node *on the path to  $a$*
- Likewise  $d$  is a *child* of  $c$  if  $c$  is its parent

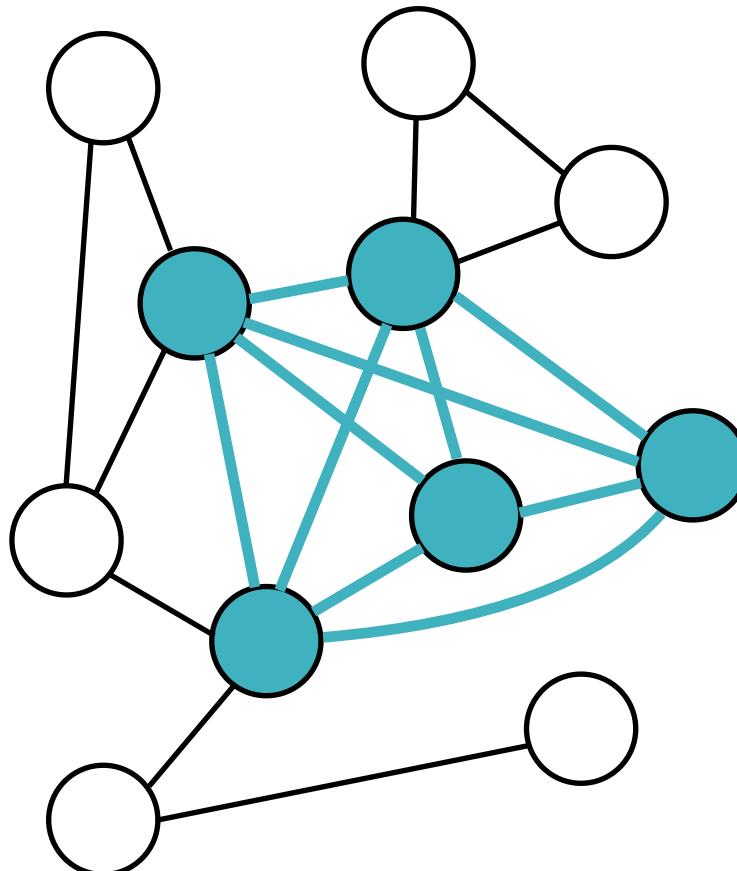
# Types of Graphs



## 4. Bipartite graphs

- Nodes can be divided into two “classes” (say A and B)
- Each edge connects a node in A with a node in B
- Can be either directed or undirected

# Types of Graphs



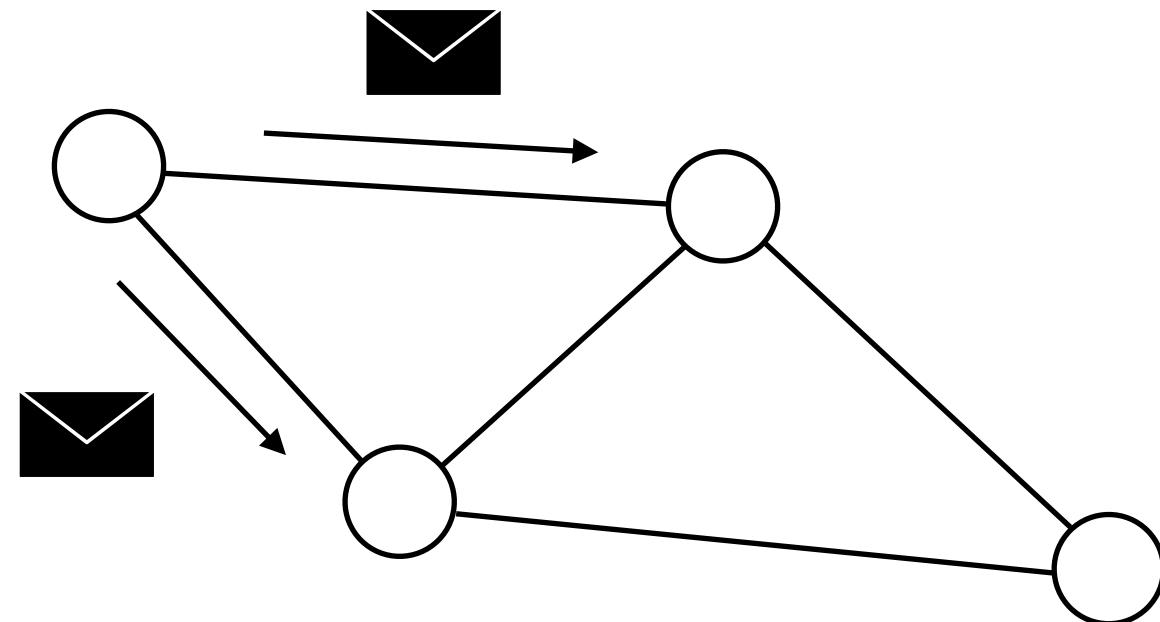
## 5. Subgraphs

Let  $G = (V, E)$  be a graph.

- A subgraph  $G_1 = (V_1, E_1)$  of  $G$  is a graph such that  $V_1 \subset V$  and  $E_1 \subset E$
- If a subgraph is *fully-connected*, then we call it a **clique**

# Message passing

Algorithms defined on graphs where information is passed between neighbours



# Topics covered in this lecture

1. Probabilistic graphical models (PGMs)
2. Belief propagation on PGMs
3. Some extensions of belief propagation
4. Message passing neural networks



# Supplementary materials

- Github link: <https://github.com/sotakao/ml-seminar-ucl>
- References provided at the end of each section
- See Bishop's book [1] for necessary background in graphs and probability theory

[1] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

# 1. Probabilistic Graphical Models (PGMs)

# Example

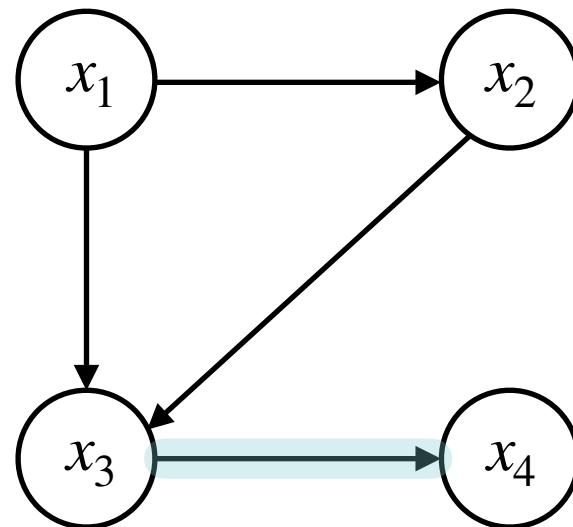
$$\begin{aligned} p(x_1, x_2, x_3, x_4, x_5, x_6, x_7) &= p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3) \\ &\quad p(x_5|x_1, x_3)p(x_6|x_4)p(x_7|x_4, x_5) \end{aligned}$$

## Questions:

- If  $x_4$  is observed, are the variables  $x_2$  and  $x_6$  independent?  
i.e.,  $p(x_2, x_6 | x_4) \stackrel{?}{=} p(x_2 | x_4)p(x_6 | x_4)$
- Which variable should we observe for  $x_6$  and  $x_7$  to be independent?  
i.e.,  $p(x_6, x_7 | ?) = p(x_6 | ?)p(x_7 | ?)$

PGMs provide elegant answers to such questions!

# Bayesian Networks



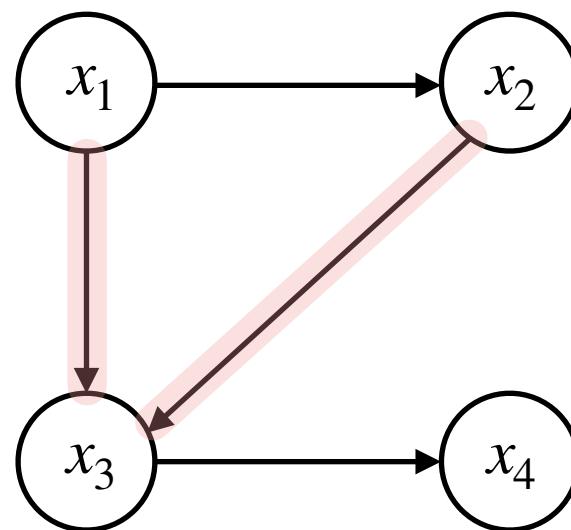
**Bayesian networks (BN)** visualise how a **joint probability distribution** factorises into **conditional probability distributions**

**Example:**

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

- Represented by a **directed acyclic graph (DAG)**
- Nodes represent variables in the model
- Edges represent causal relations between variables

# Bayesian Networks



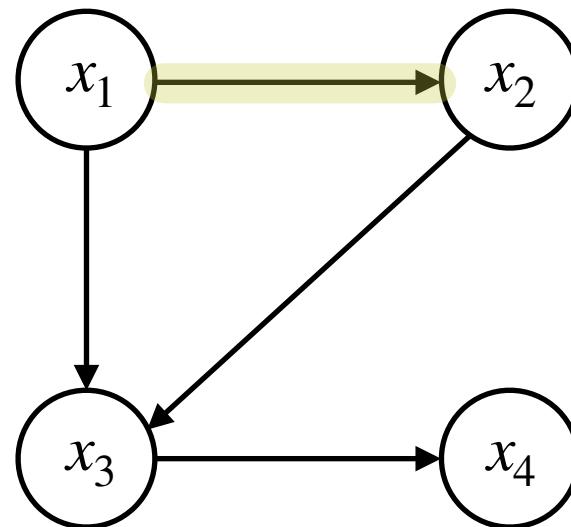
**Bayesian networks (BN)** visualise how a **joint probability distribution** factorises into **conditional probability distributions**

**Example:**

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

- Represented by a **directed acyclic graph (DAG)**
- Nodes represent variables in the model
- Edges represent causal relations between variables

# Bayesian Networks



**Bayesian networks (BN)** visualise how a **joint probability distribution** factorises into **conditional probability distributions**

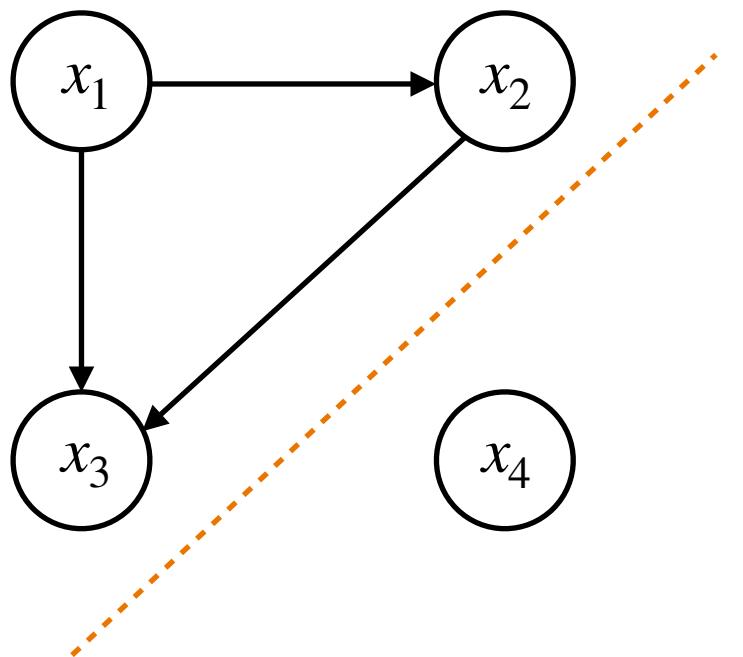
**Example:**

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

- Represented by a **directed acyclic graph (DAG)**
- Nodes represent variables in the model
- Edges represent causal relations between variables

# Independence

Two nodes are *independent* if there are no paths connecting them



$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_4) p(x_3 | x_1, x_2) p(x_2 | x_1) p(x_1) \\ &= p(x_4) p(x_1, x_2, x_3) \end{aligned}$$

→  $x_4$  is independent of all other nodes

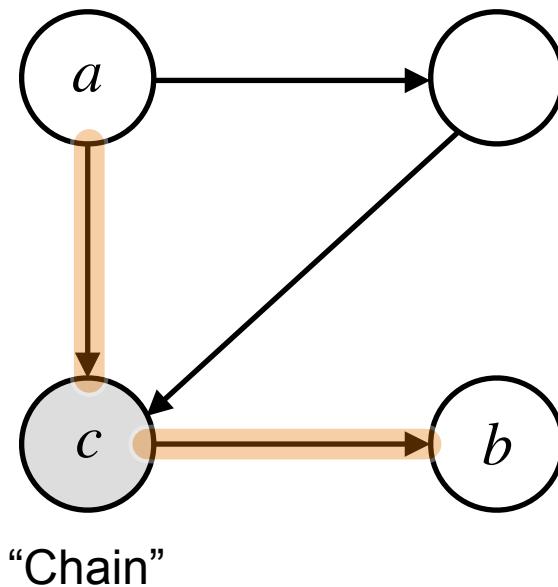
# d-Separation and conditional independence

Two nodes  $a$  and  $b$  in a DAG are **d-separated** by a set of nodes  $Z$  if and only if *any* loop-free path from  $a$  to  $b$  satisfies one of the following:

1.  Path contains a **chain** and  $c$  belongs to  $Z$ .
2.  Path contains a **fork** and  $c$  belongs to  $Z$ .
3.  Path contains a **collider** and  $c$  *does not* belong to  $Z$ .  
In addition, no descendant of  $c$  belongs to  $Z$ .

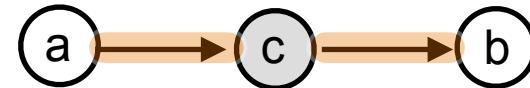
**Property:** variables  $a, b$  are independent given  $Z \Leftrightarrow$  they are d-separated by  $Z$

# Example of d-separation



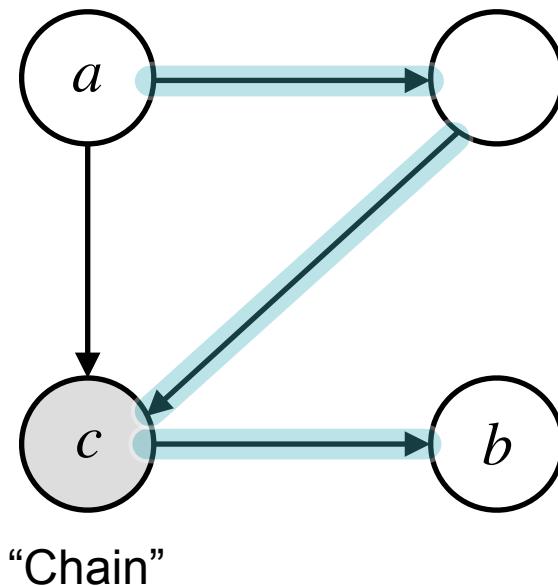
$a$  and  $b$  are d-separated by  $c$  because

1.  $c$  is sandwiched by a **chain** in the path



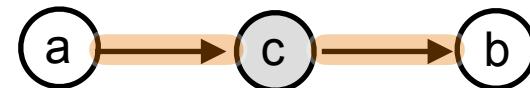
"Chain"

# Example of d-separation



$a$  and  $b$  are d-separated by  $c$  because

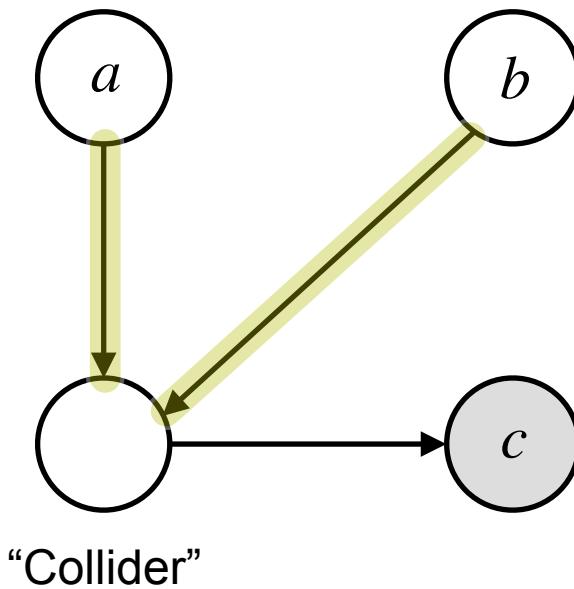
1.  $c$  is sandwiched by a **chain** in the path



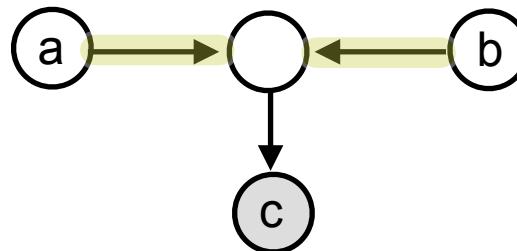
2.  $c$  is sandwiched by a **chain** in the path



# Non-example of d-separation



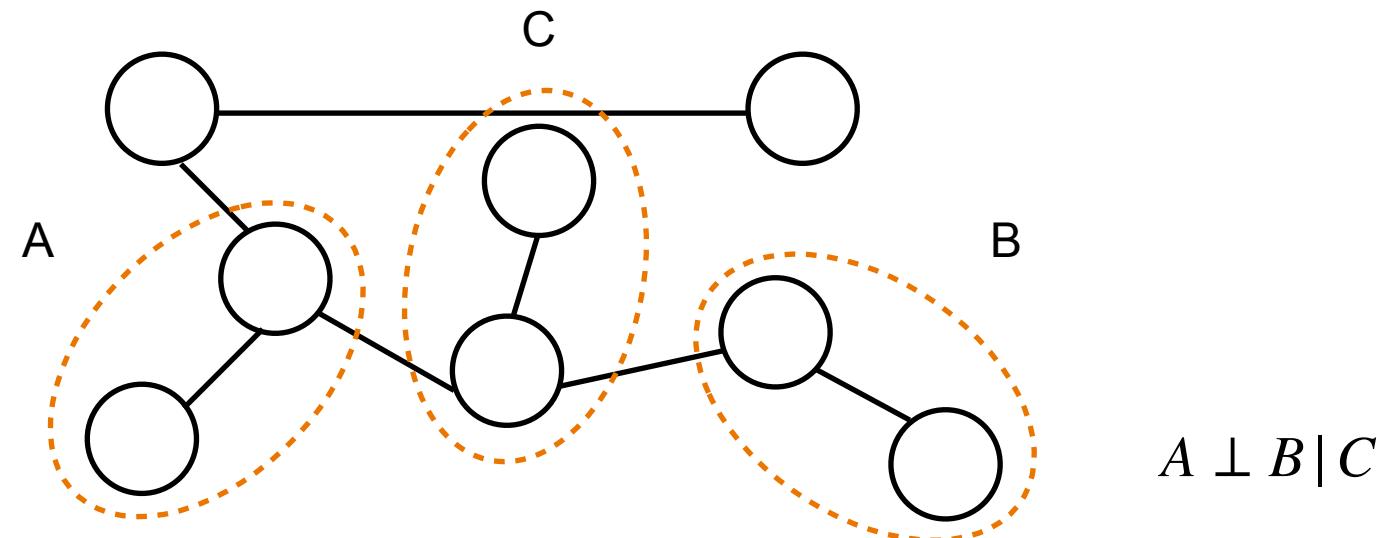
Nodes  $a$  and  $b$  are *not* d-separated by  $c$   
(i.e.,  $a$  and  $b$  are d-connected)  
because



contains a **collider** and  $c$  is a descendant of  
the collider node

# Markov Random Fields

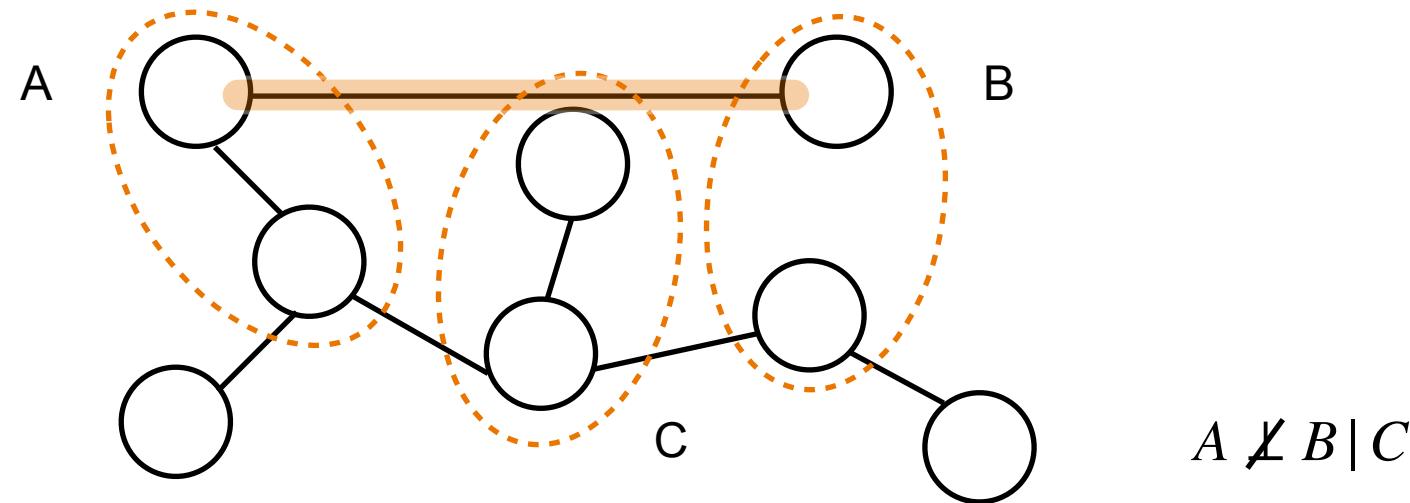
- **Markov random fields (MRF)** are represented by **undirected graphs**



- A and B are conditionally independent given C if and only if paths between points in A and B are **blocked** by C

# Markov Random Fields

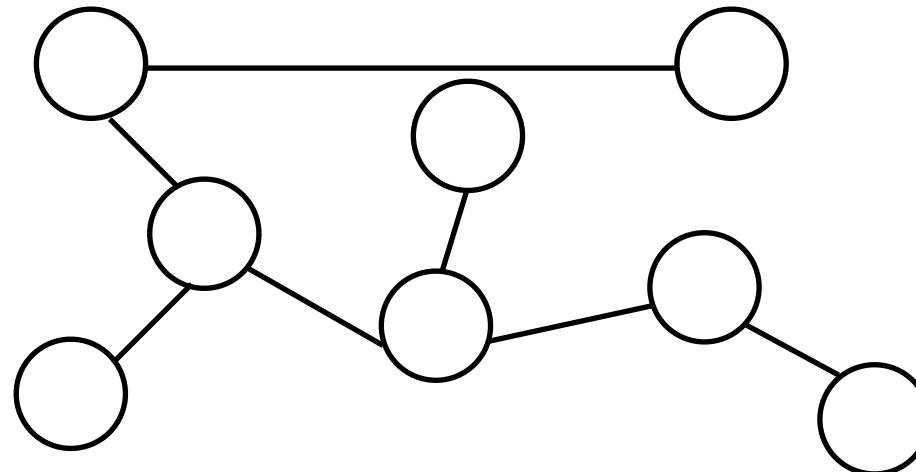
- **Markov random fields (MRF)** are represented by **undirected graphs**



- A and B are conditionally independent given C if and only if paths between points in A and B are **blocked** by C

# Markov Random Fields

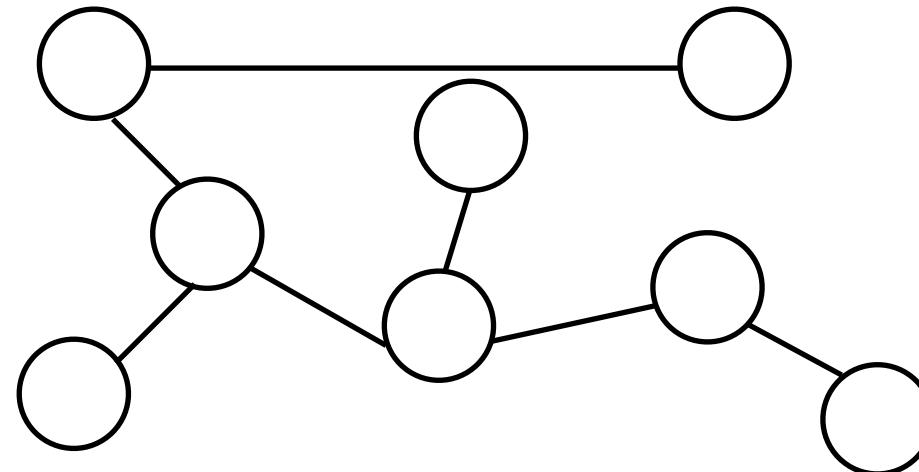
- **Markov random fields (MRF)** are represented by **undirected graphs**



- A and B are conditionally independent given C if and only if paths between points in A and B are **blocked** by C

# Markov Random Fields

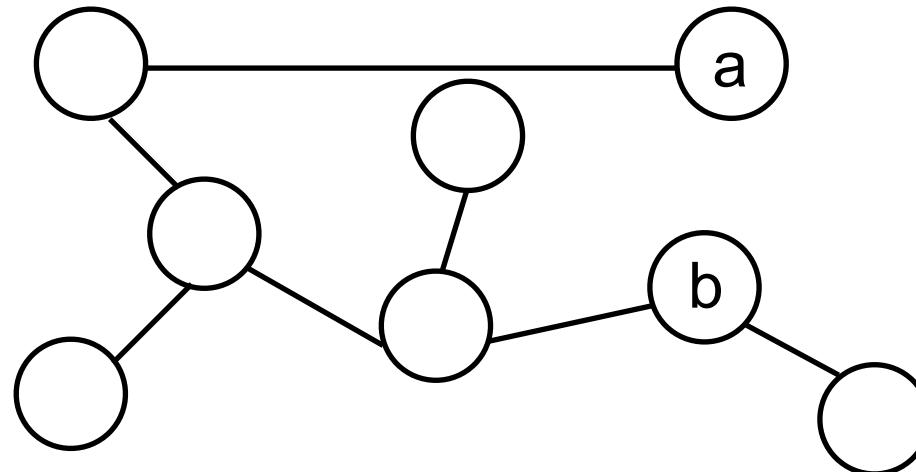
- **Markov random fields (MRF)** are represented by **undirected graphs**



- A and B are conditionally independent given C if and only if paths between points in A and B are **blocked** by C

# Markov Random Fields

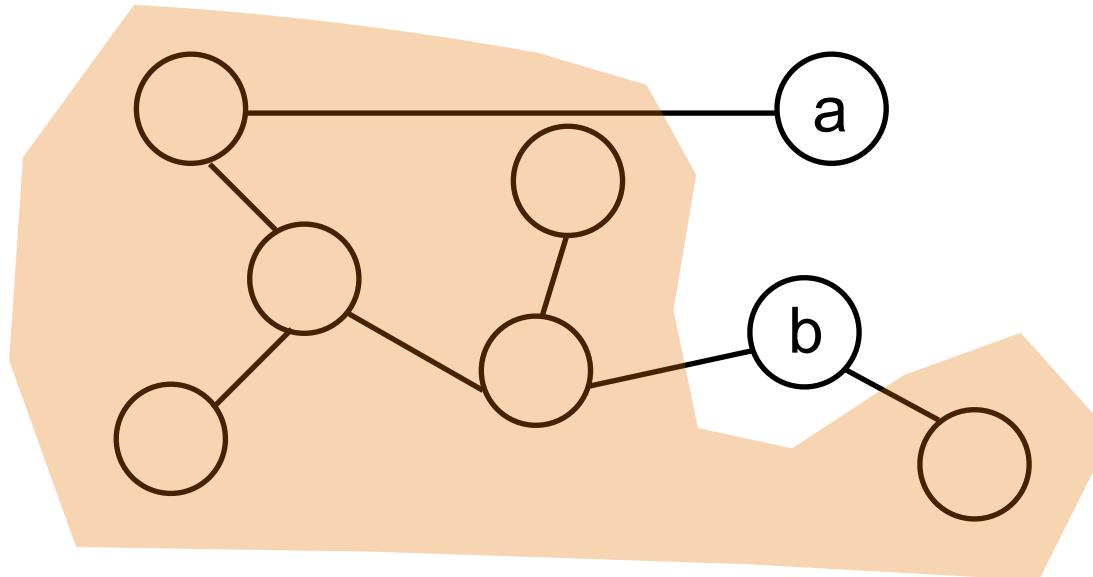
- **Markov random fields (MRF)** are represented by **undirected graphs**



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

# Markov Random Fields

- **Markov random fields (MRF)** are represented by **undirected graphs**



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

# Hammersley-Clifford Theorem

In MRFs, we can consider factorisations into **potential functions**  $\psi_C(x_C) \geq 0$ :

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

where  $C$  is a clique of the graph\*.

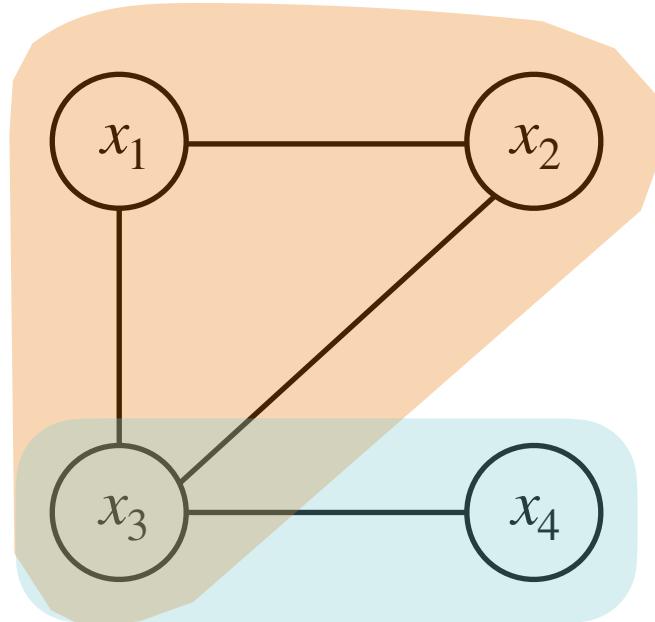
Akin to factorising **joint distributions** into *conditional distributions* in BNs.

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

\*Recall that a *clique* is a fully-connected subgraph of a graph

# Example illustrating the Hammersley-Clifford theorem

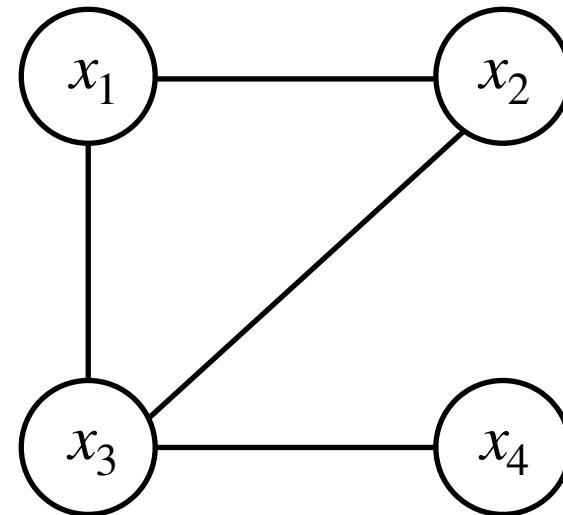
## 1. Factorisation into maximal cliques



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

# Example illustrating the Hammersley-Clifford theorem

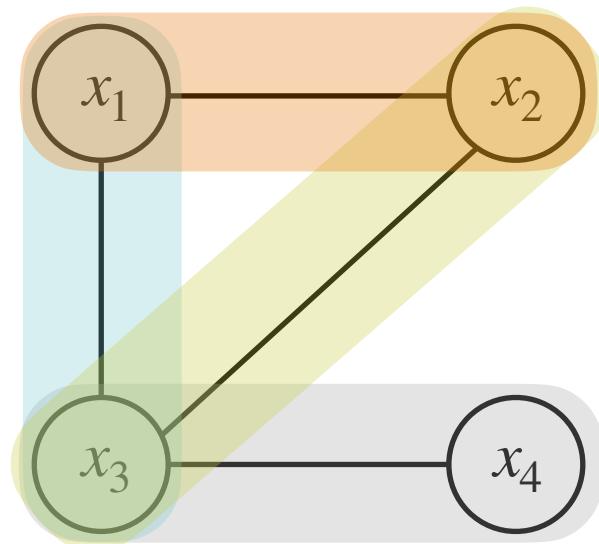
## 2. Factorisation into pairwise cliques



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

# Example illustrating the Hammersley-Clifford theorem

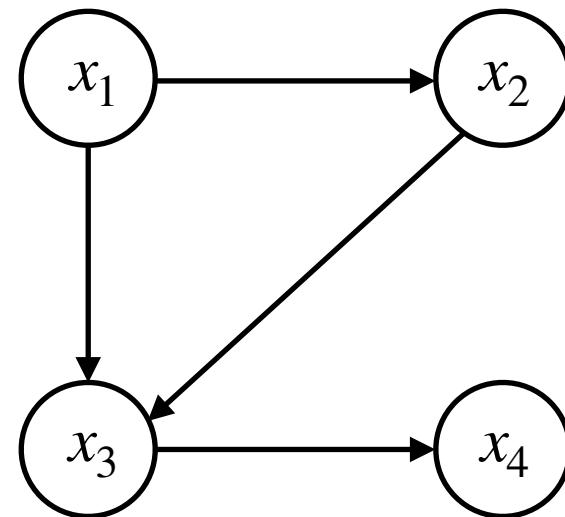
## 2. Factorisation into pairwise cliques



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

# Example illustrating the Hammersley-Clifford theorem

## 3. Factorisation of Bayesian networks

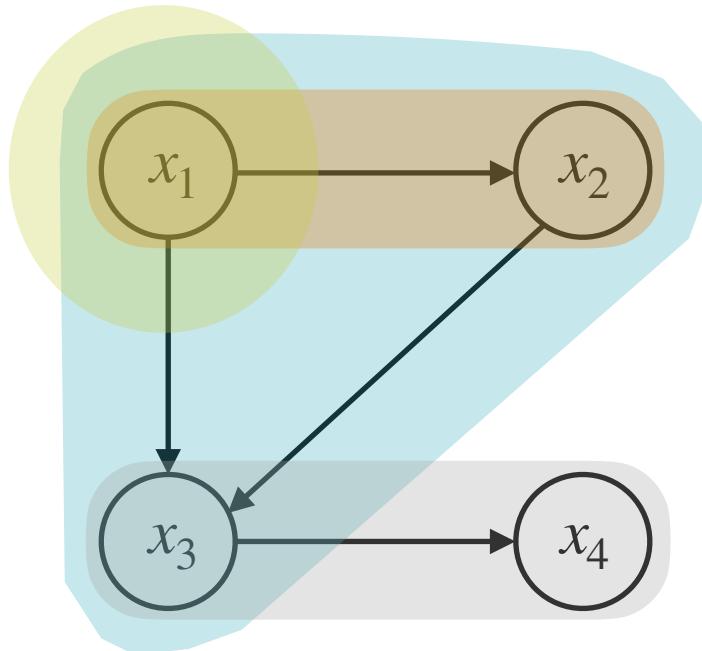


$$p(x_1, x_2, x_3, x_4)$$

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

# Example illustrating the Hammersley-Clifford theorem

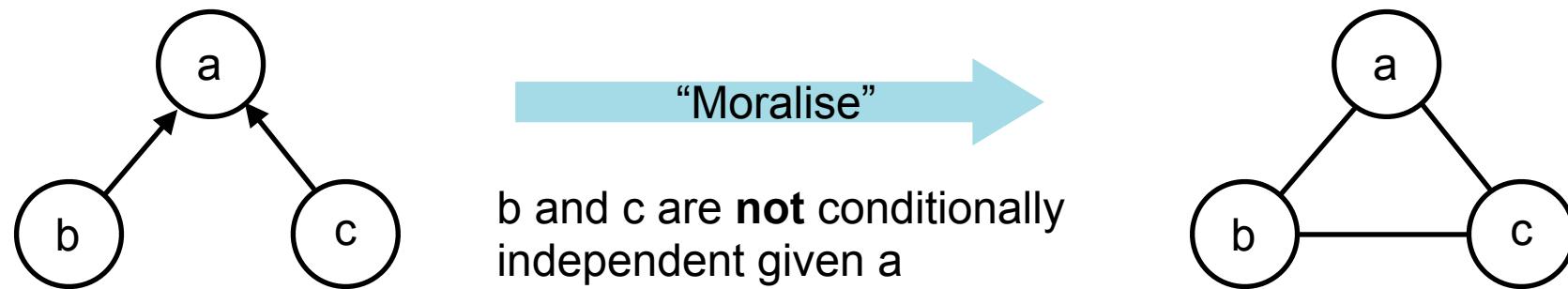
## 3. Factorisation of Bayesian networks



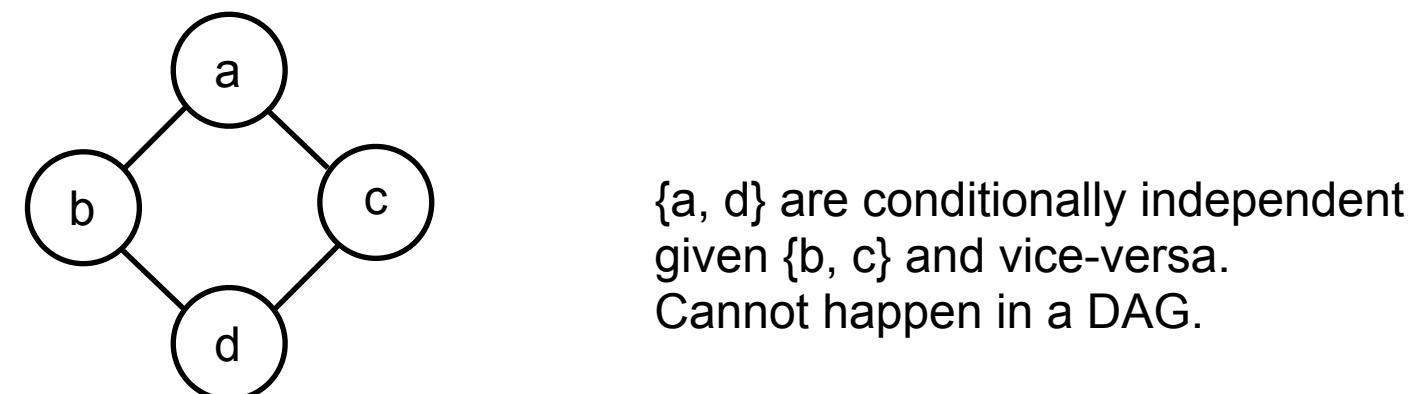
$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_4 | x_3) p(x_3 | x_1, x_2) p(x_2 | x_1) p(x_1) \\ &= \psi_{34}(x_3, x_4) \psi_{123}(x_1, x_2, x_3) \psi_{12}(x_1, x_2) \psi_1(x_1) \end{aligned}$$

# Markov Random Fields $\neq$ Bayesian Networks

- MRFs *do not* represent BNs without altering the graph structure, e.g.

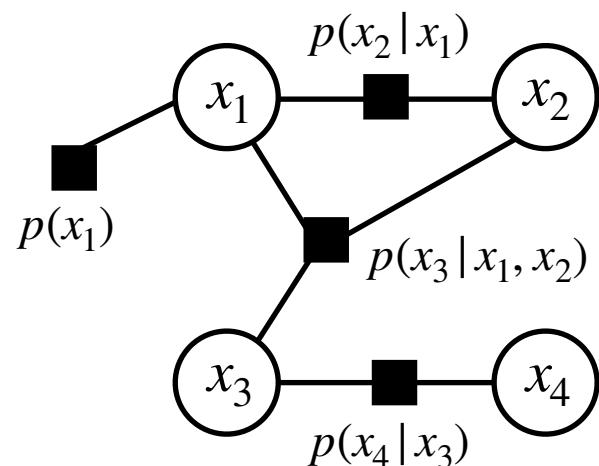
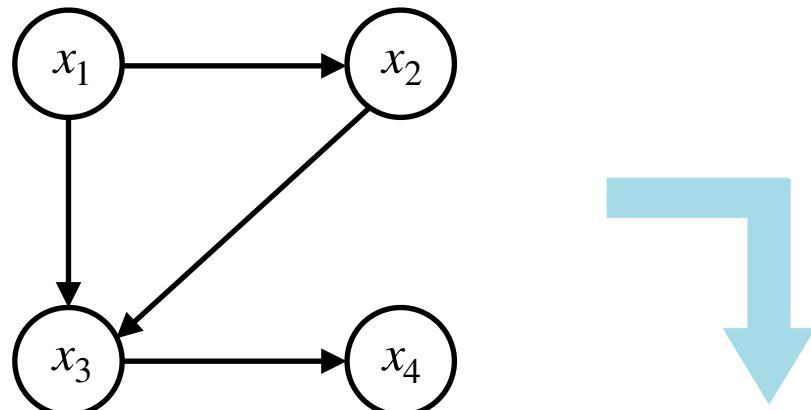


- Not all MRFs can be represented as a BN, e.g.



# Factor Graphs

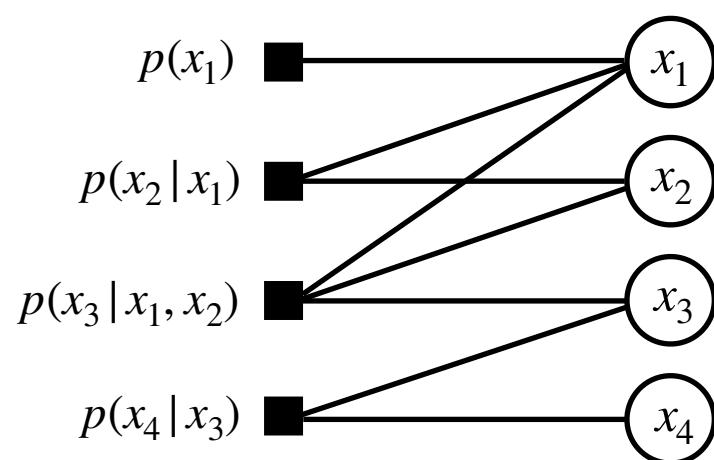
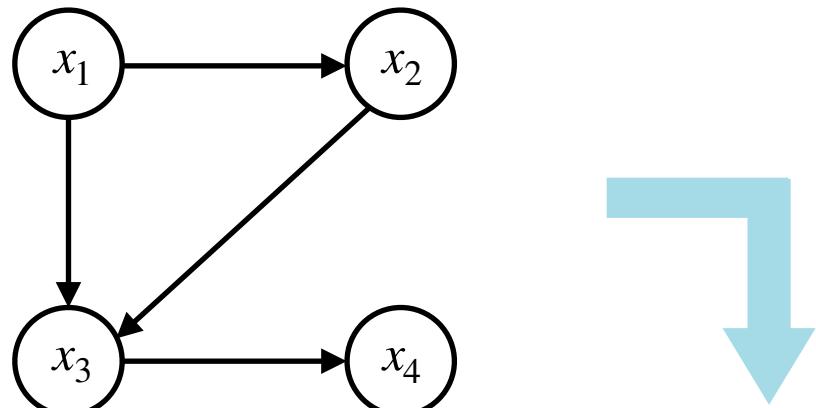
$$p(x_1, x_2, x_3, x_4) = p(x_4 | x_3) p(x_3 | x_1, x_2) p(x_2 | x_1) p(x_1)$$



- **Factor graphs** are alternative representations of BNs and MRFs
- Makes the factorisation explicit
- Circle nodes ( $\circ$ ) represent variables
- Square nodes ( $\blacksquare$ ) represent *factors*
- Graph is **undirected** and **bipartite**

# Factor Graphs

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



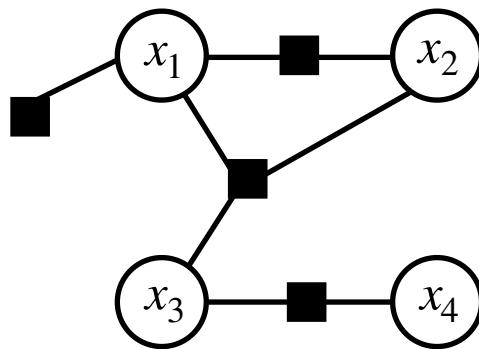
- **Factor graphs** are alternative representations of BNs and MRFs
- Makes the factorisation explicit
- Circle nodes ( $\circ$ ) represent variables
- Square nodes ( $\blacksquare$ ) represent *factors*
- Graph is **undirected** and **bipartite**

# Factor Graphs

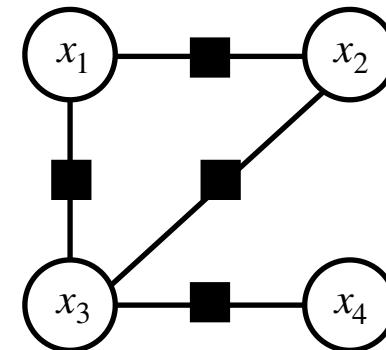
Factor graphs make the factorisation explicit

⇒ Useful for MRFs where factorisation is non-unique

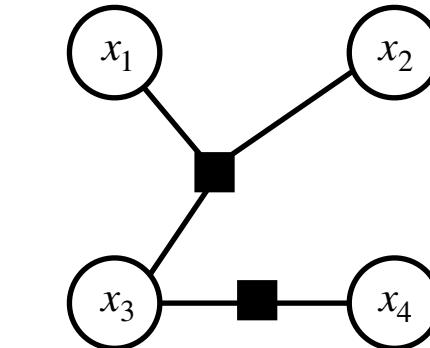
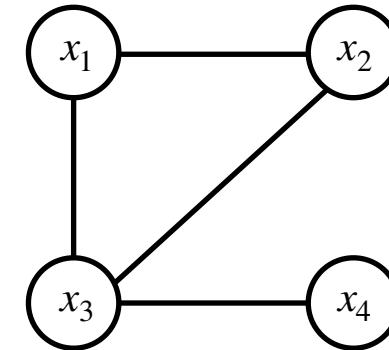
There are many ways of factorising into potentials:



$$\psi_1(x_1)\psi_{12}(x_1, x_2)\psi_{13}(x_1, x_2, x_3)\psi_{34}(x_3, x_4)$$



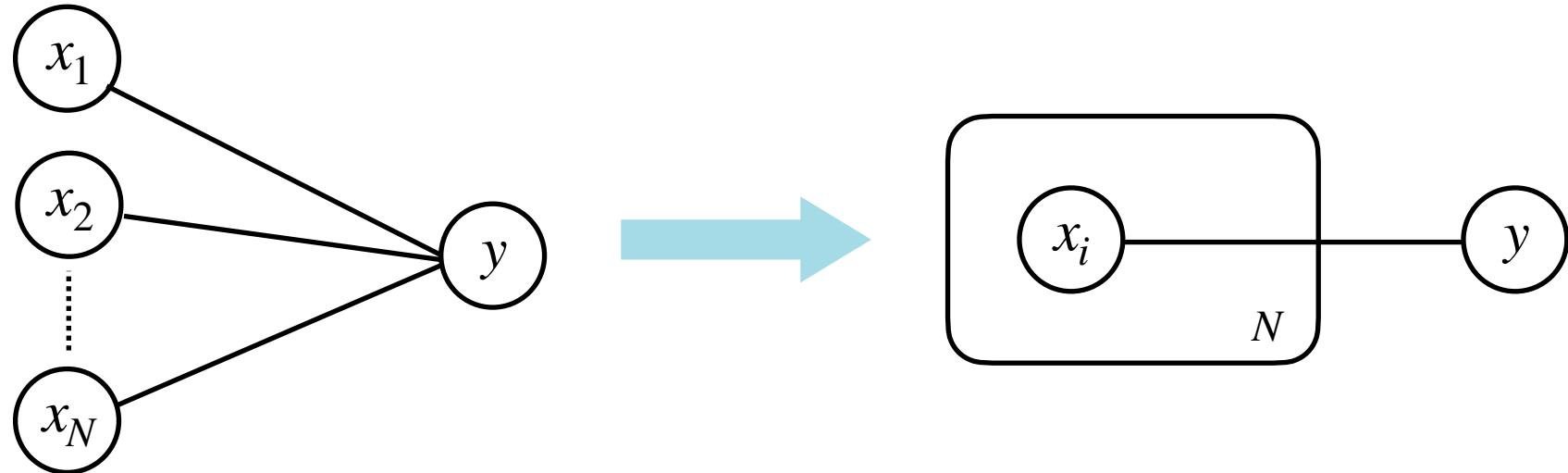
$$\psi_{12}(x_1, x_2)\psi_{23}(x_2, x_3)\psi_{13}(x_1, x_2, x_3)\psi_{34}(x_3, x_4)$$



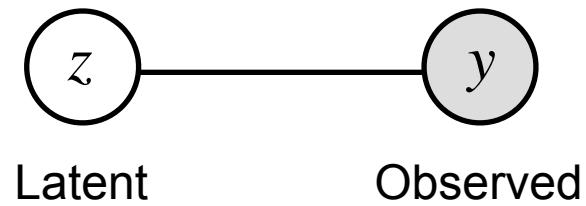
$$\psi_{123}(x_1, x_2, x_3)\psi_{34}(x_3, x_4)$$

# Useful notations

- Plate notation

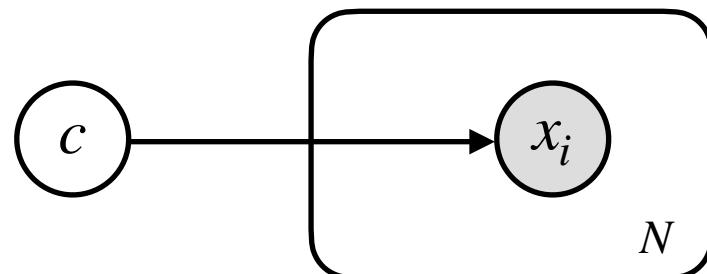


- Shaded vs. unshaded nodes

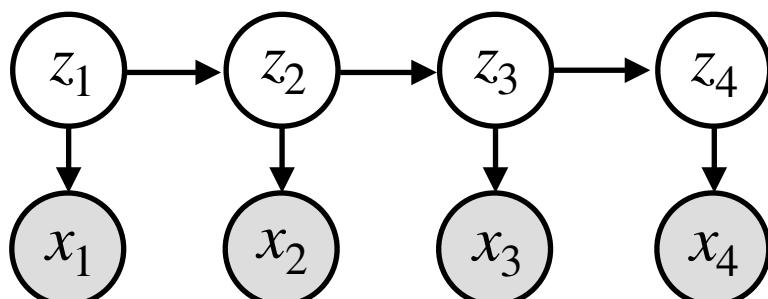


# Examples of Bayesian networks

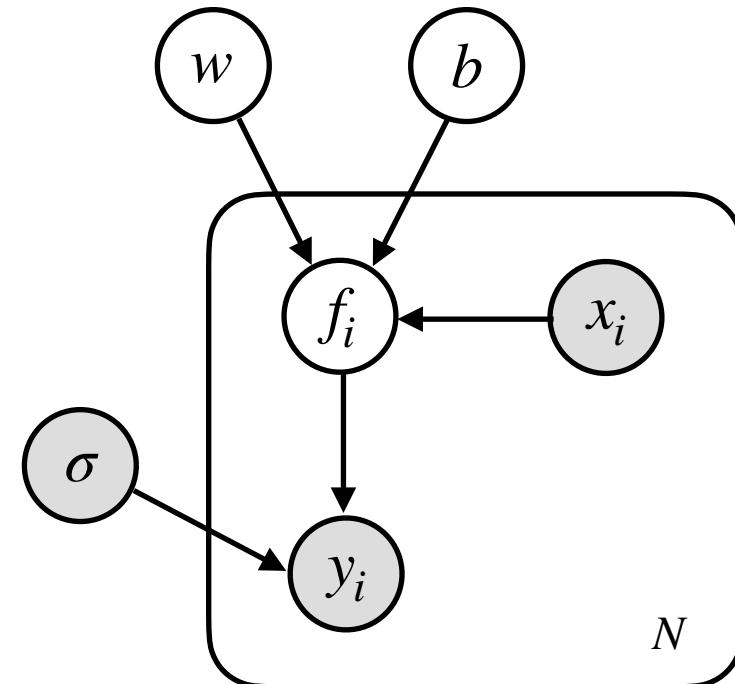
- Naive Bayes classifier



- Hidden Markov model



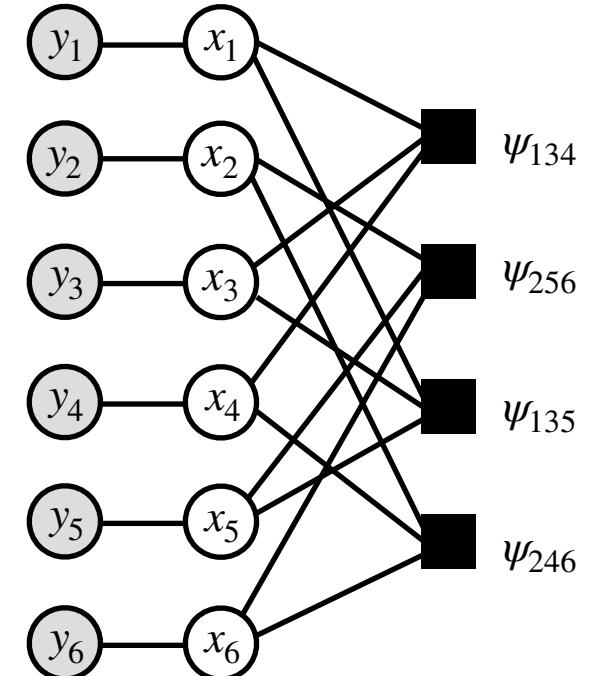
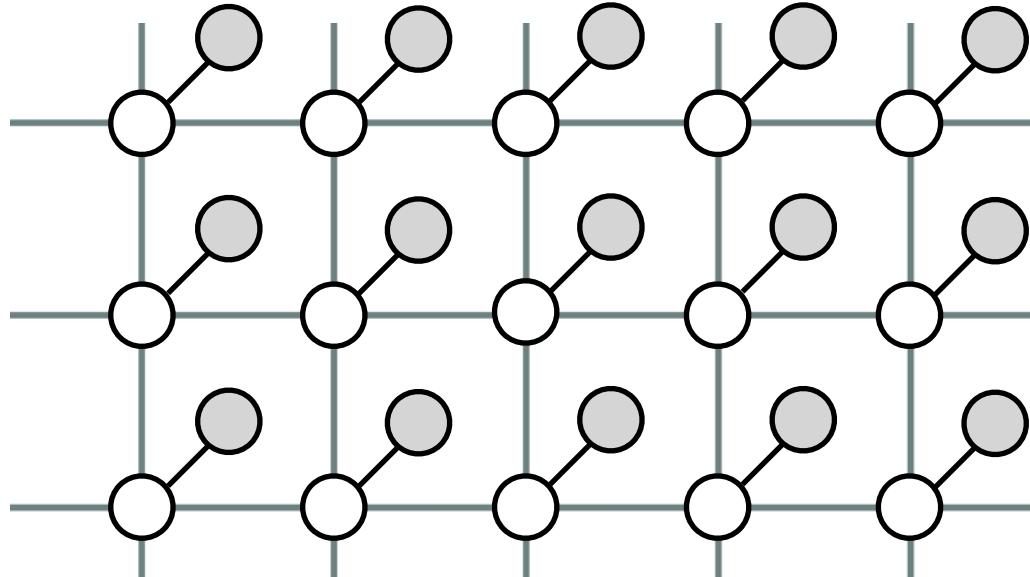
- Bayesian linear regression



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

# Examples of Markov random fields

- Spatial analysis / image processing [3,4]
- Error-correcting codes [5]



## 2. Belief Propagation on PGMs

# 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  $x^* = \operatorname{argmax}_x p(x)$

Here, we will focus on computing *marginal distributions* using PGMs.

# Example

Let  $X_1, \dots, X_N$  be random variables, each with  $K$  discrete states and  $p$  is the joint probability mass function.

**Question:** What is the *marginal distribution*  $p(X_i = x_i)$  for all  $i = 1, \dots, N$ ?

**A naive solution:**

$$p(X_i = x_i) = \sum_{j \neq i}^N \left( \sum_{x_j \in \{1, \dots, K\}} p(x_1, \dots, x_N) \right).$$

This has computational cost  $\mathcal{O}(K^N)$

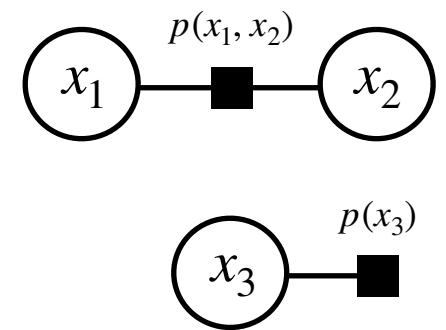


# Assuming independence

Let  $N = 3$  and assume we can write  $p(x_1, x_2, x_3) = p(x_1, x_2)p(x_3)$ .

Then, we have

$$\begin{aligned}
 p(x_1) &= \sum_{x_2 \in \{1, \dots, K\}} \sum_{x_3 \in \{1, \dots, K\}} p(x_1, x_2, x_3) \\
 &= \sum_{x_2 \in \{1, \dots, K\}} \sum_{x_3 \in \{1, \dots, K\}} p(x_1, x_2)p(x_3) \\
 &= \sum_{x_2 \in \{1, \dots, K\}} p(x_1, x_2) \underbrace{\sum_{x_3 \in \{1, \dots, K\}} p(x_3)}_{=1} \\
 &= \sum_{x_2 \in \{1, \dots, K\}} p(x_1, x_2)
 \end{aligned}$$



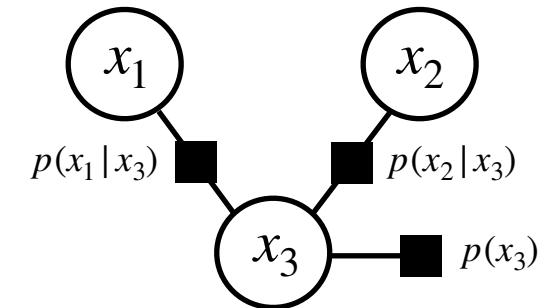
A single sum is cheaper to compute than a double sum!

# Assuming conditional independence

Now assume we have  $p(x_1, x_2, x_3) = p(x_1 | x_3)p(x_2 | x_3)p(x_3)$ .

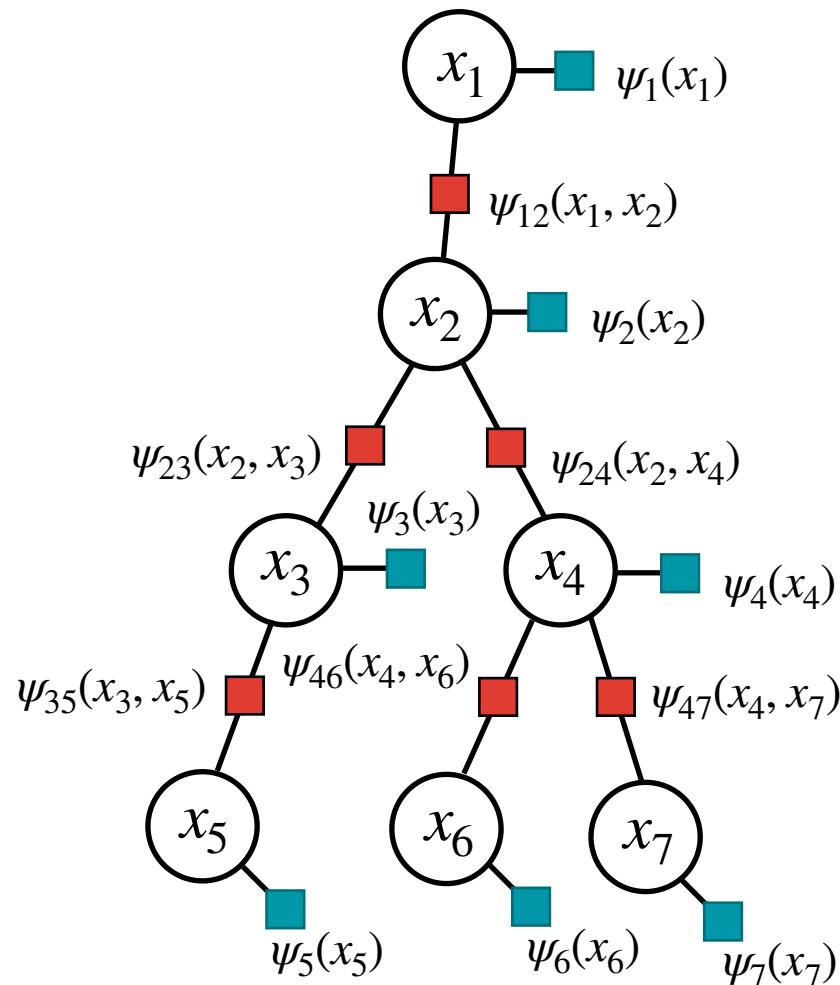
Then,

$$\begin{aligned}
 p(x_1) &= \sum_{x_2 \in \{1, \dots, K\}} \sum_{x_3 \in \{1, \dots, K\}} p(x_1, x_2, x_3) \\
 &= \sum_{x_2 \in \{1, \dots, K\}} \sum_{x_3 \in \{1, \dots, K\}} p(x_1 | x_3)p(x_2 | x_3)p(x_3) \\
 &= \sum_{x_3 \in \{1, \dots, K\}} p(x_1 | x_3)p(x_3) \underbrace{\sum_{x_2 \in \{1, \dots, K\}} p(x_2 | x_3)}_{=1} \\
 &= \sum_{x_3 \in \{1, \dots, K\}} p(x_1 | x_3)p(x_3)
 \end{aligned}$$



**Observation:** Independence / conditional independence helps to reduce complexity!  
 $\equiv$  sparsity of graph

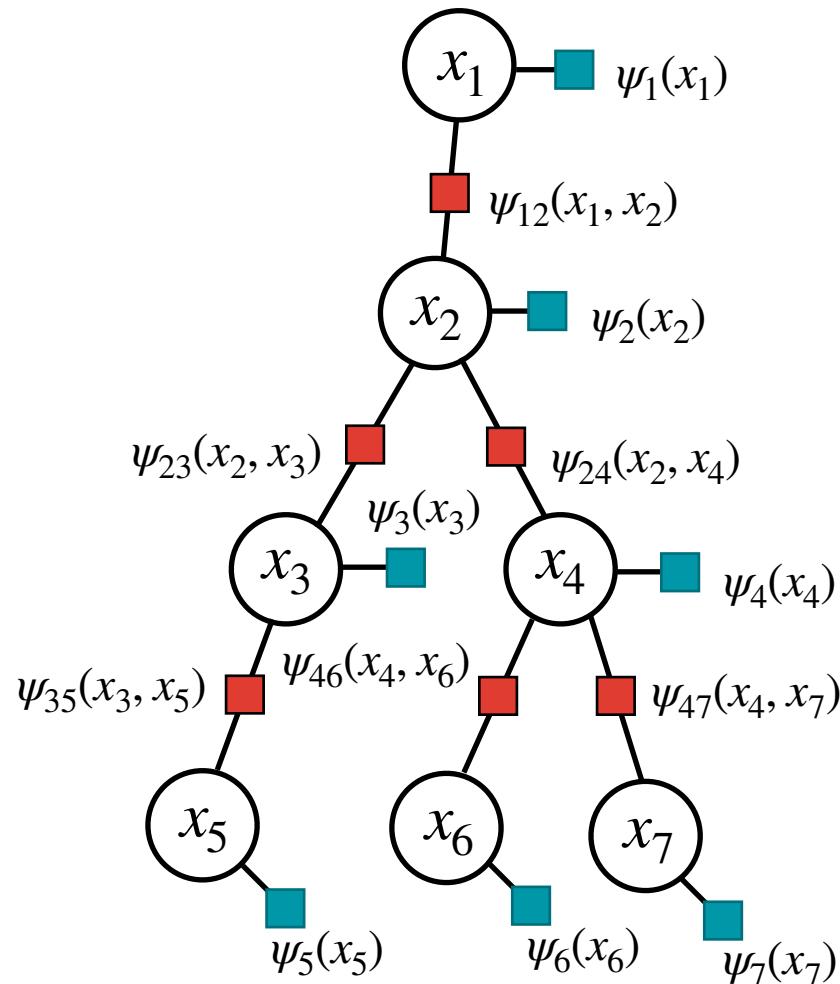
# Belief propagation algorithm



- **Belief propagation** efficiently computes *marginal probabilities*  $p(x_i)$  on trees
- Assume that the graph is **tree-structured**
- Operate on factor graphs

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

# Belief propagation algorithm



BP proceeds by iteratively updating:

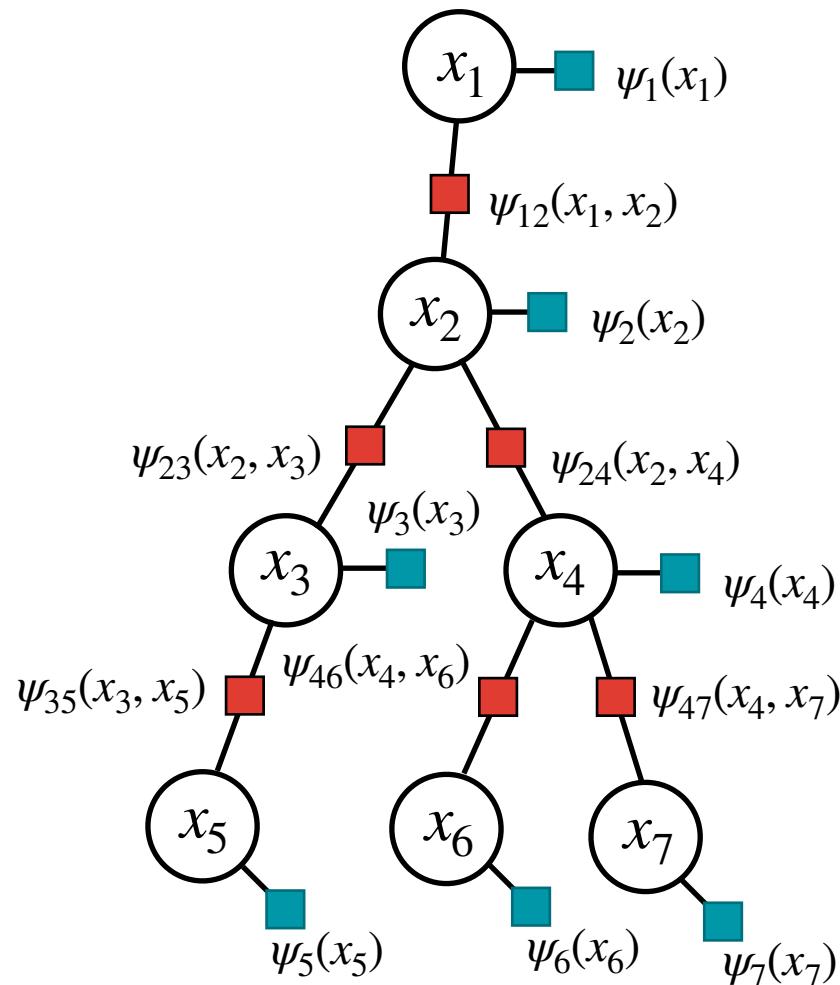
1. The “**messages**” between two nodes

$$M_{j \rightarrow i}(x_i)$$

2. The “**state**” of each node

$$p(x_i)$$

# Belief propagation algorithm



BP proceeds by iteratively updating:

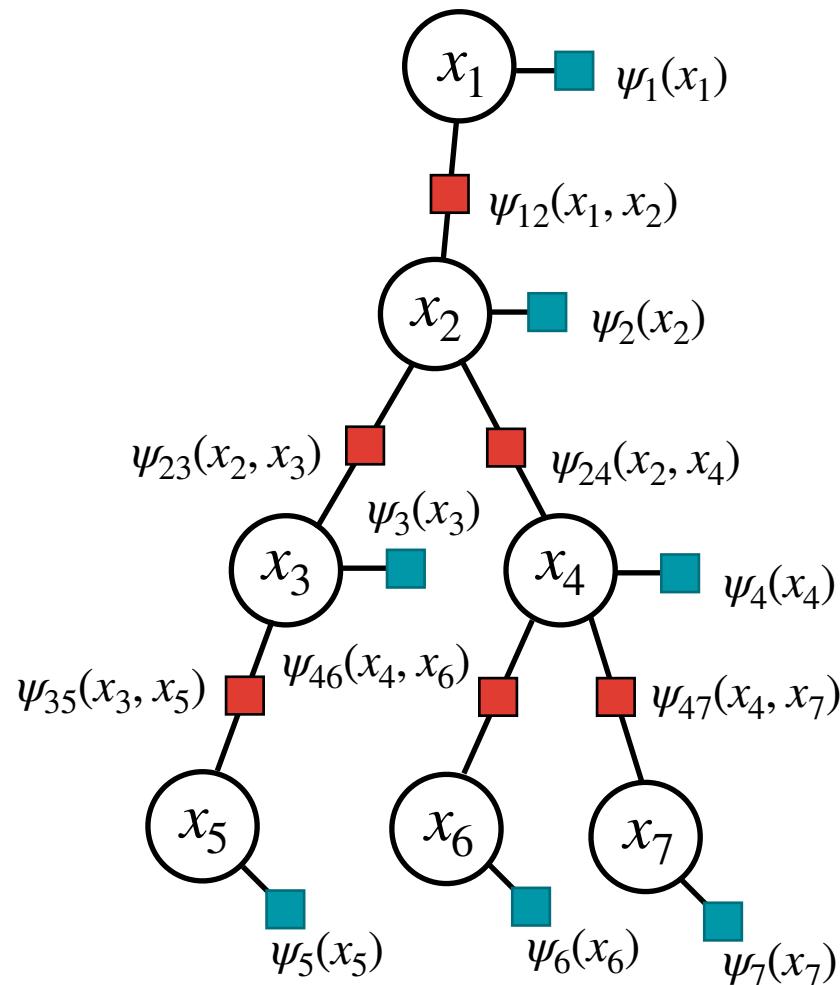
1. The “**messages**” between two nodes

$$M_{j \rightarrow i}(x_i) \rightarrow \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

2. The “**state**” of each node

$$p(x_i) \rightarrow \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i)$$

# Belief propagation algorithm



BP proceeds by iteratively updating:

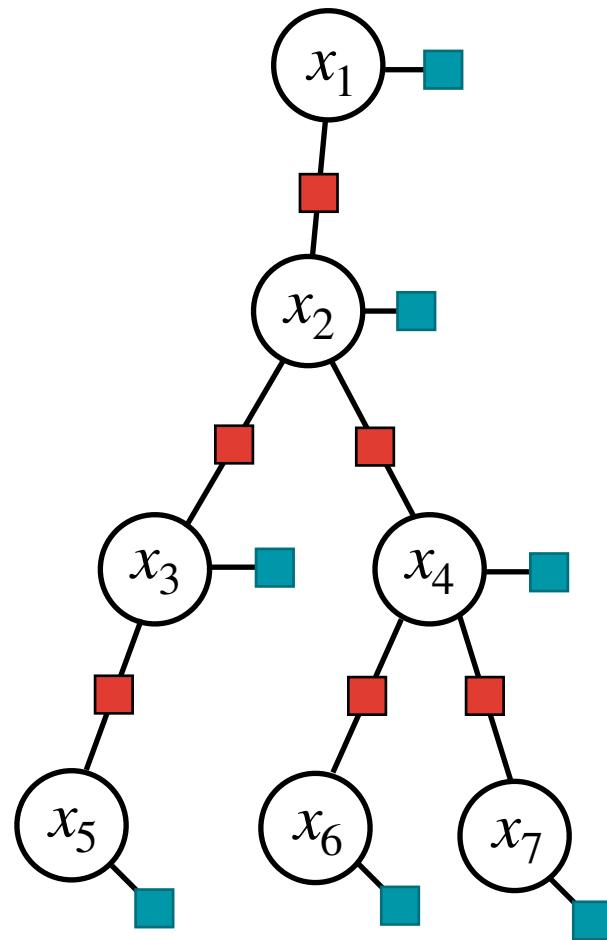
- 1. The “**messages**” between two nodes

$$M_{j \rightarrow i}(x_i) \rightarrow \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

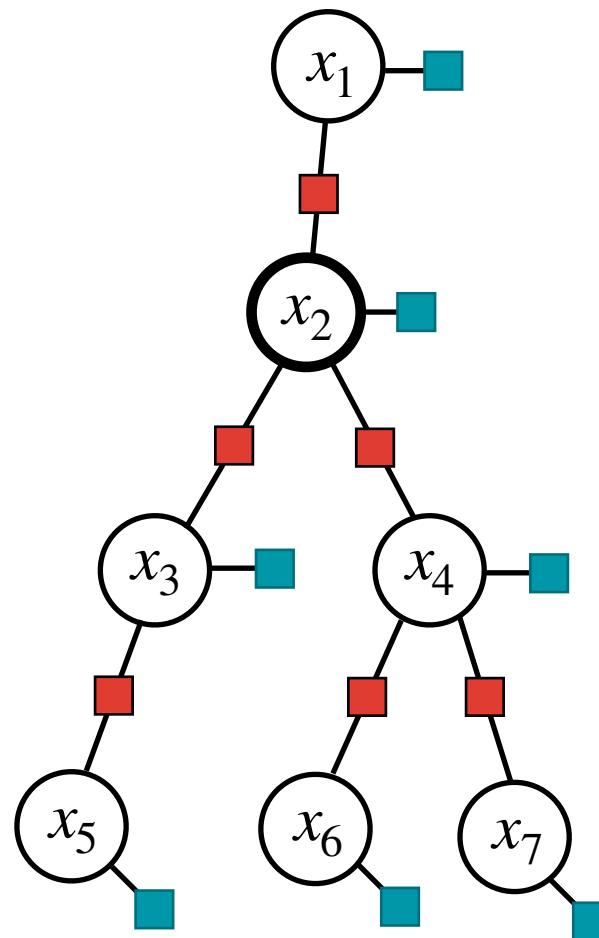
- 2. The “**state**” of each node

$$p(x_i) \rightarrow \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i)$$

## Step 1. Message update

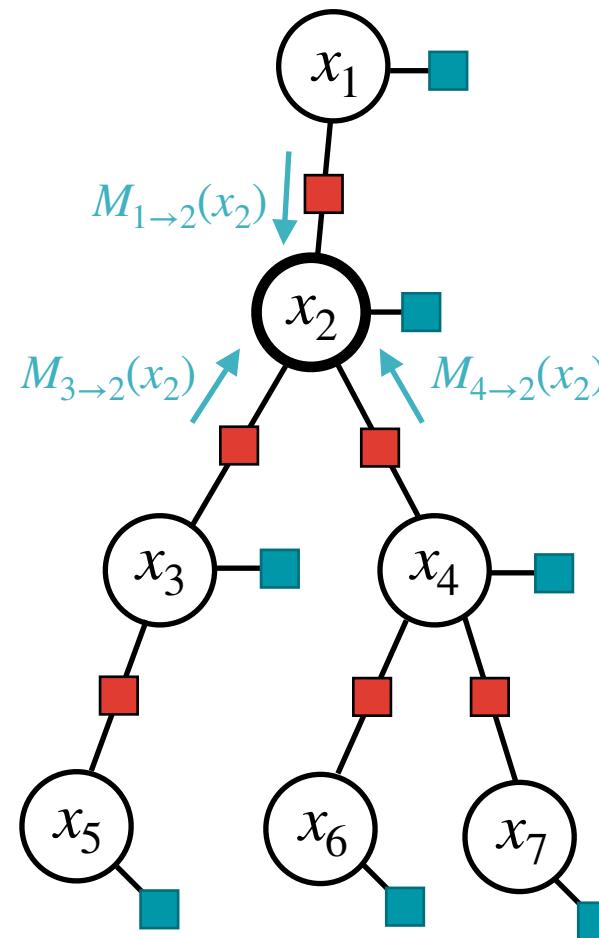


## Step 1. Message update



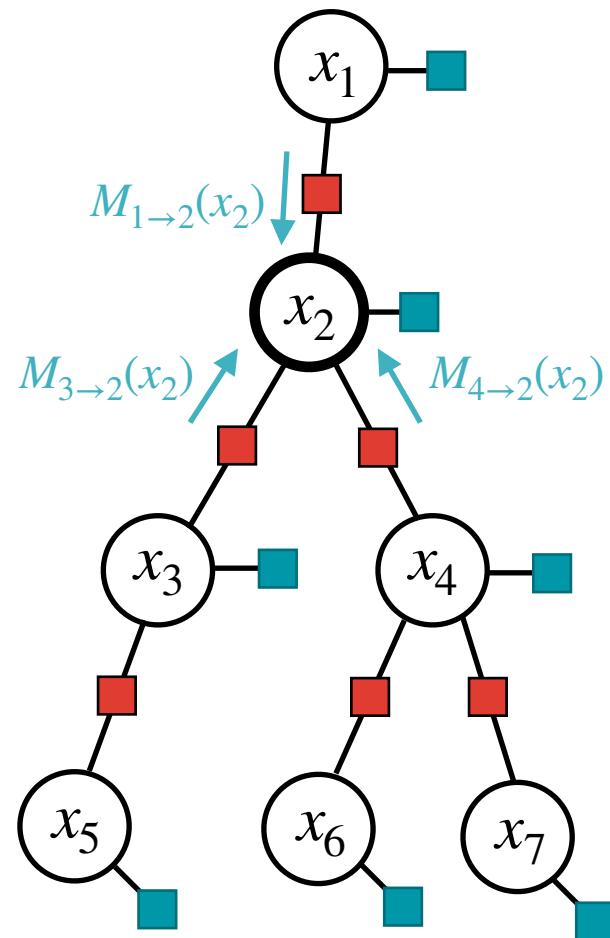
Let's say we want to compute  $p(x_2)$ .

# Step 1. Message update



Let's say we want to compute  $p(x_2)$ .

# Step 1. Message update

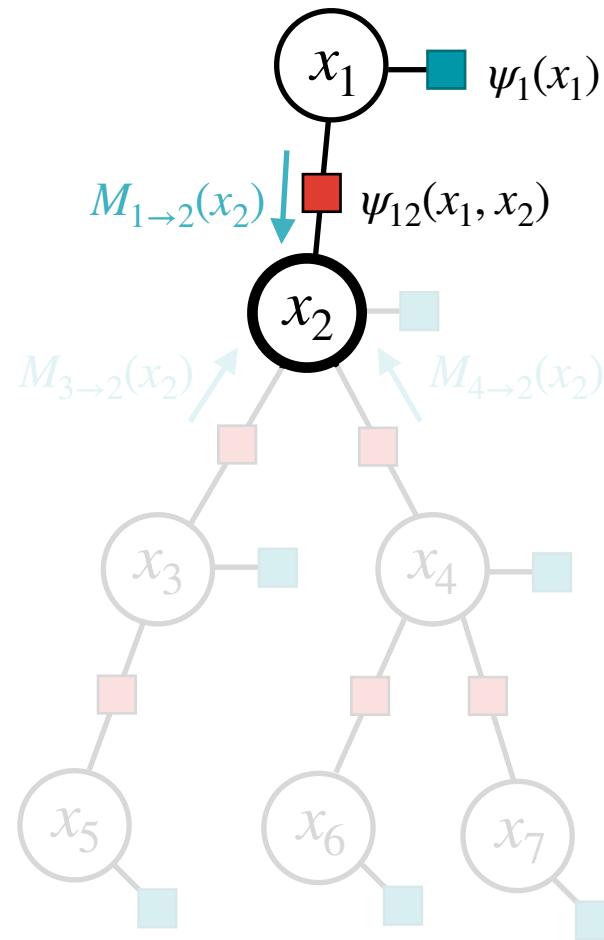


Let's say we want to compute  $p(x_2)$ .

Recall the message update step:

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

# Step 1. Message update



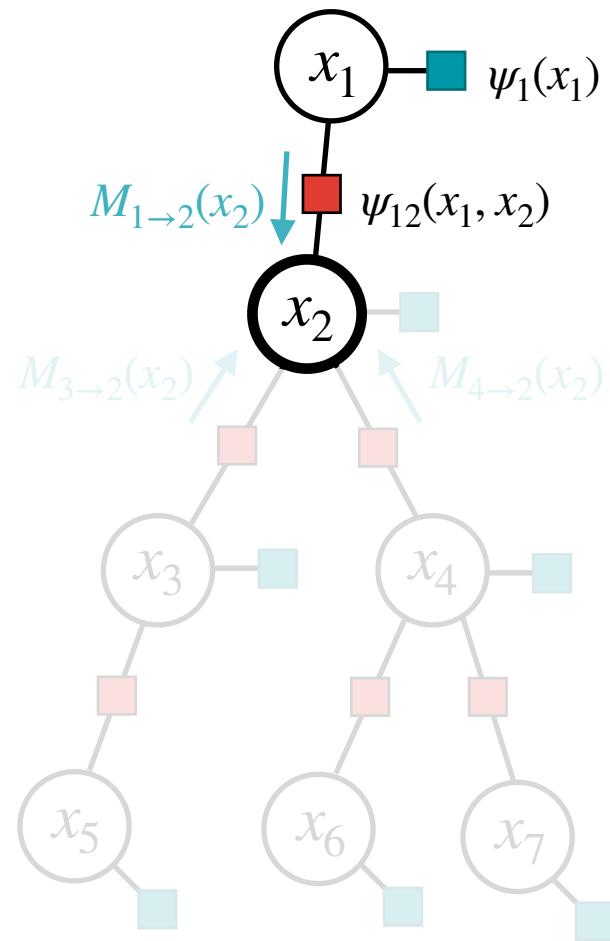
Let's say we want to compute  $p(x_2)$ .

Recall the message update step:

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

First, compute the message  $x_1 \rightarrow x_2$ :

# Step 1. Message update



Let's say we want to compute  $p(x_2)$ .

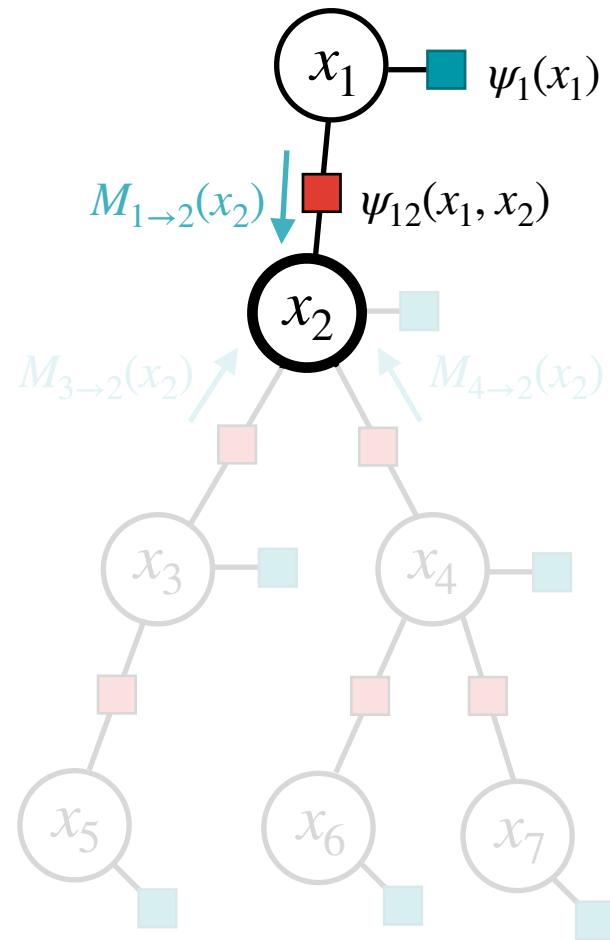
Recall the message update step:

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

First, compute the message  $x_1 \rightarrow x_2$ :

$$M_{1 \rightarrow 2}(x_2) = \sum_{x_1 \in \{1, \dots, K\}} \psi_{12}(x_1, x_2) \psi_1(x_1) \prod_{k \sim 1, k \neq 2} M_{k \rightarrow 1}(x_1)$$

# Step 1. Message update



Let's say we want to compute  $p(x_2)$ .

Recall the message update step:

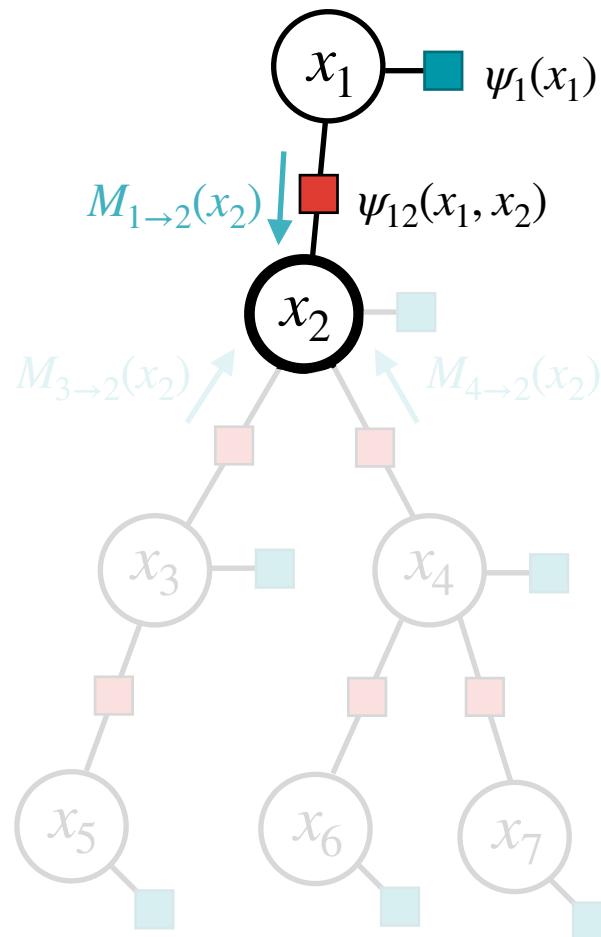
$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

First, compute the message  $x_1 \rightarrow x_2$ :

$$M_{1 \rightarrow 2}(x_2) = \sum_{x_1 \in \{1, \dots, K\}} \psi_{12}(x_1, x_2) \psi_1(x_1) \prod_{k \sim 1, k \neq 2} M_{k \rightarrow 1}(x_1)$$

??

# Step 1. Message update



Let's say we want to compute  $p(x_2)$ .

Recall the message update step:

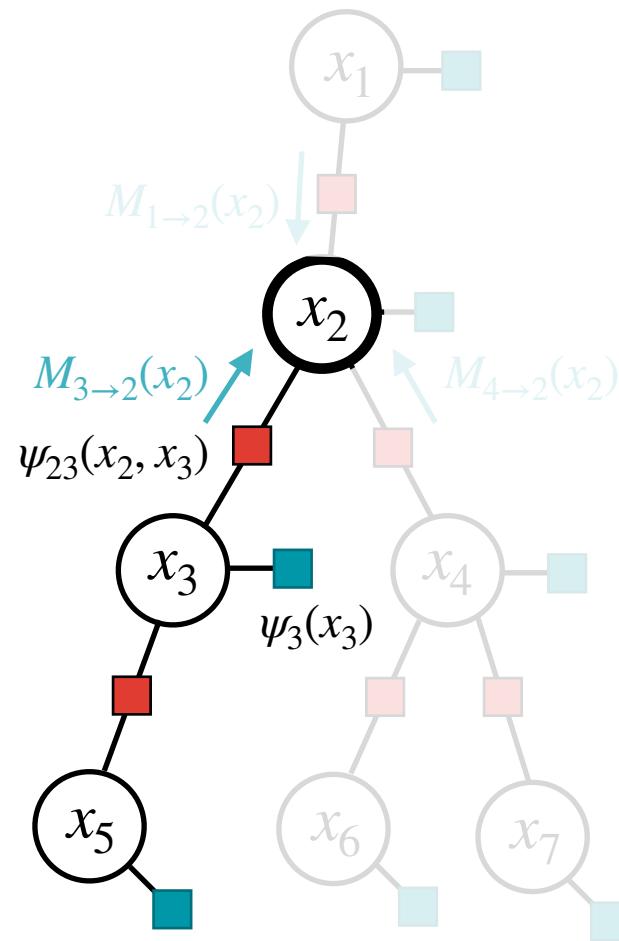
$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

First, compute the message  $x_1 \rightarrow x_2$ :

$$M_{1 \rightarrow 2}(x_2) = \sum_{x_1 \in \{1, \dots, K\}} \psi_{12}(x_1, x_2) \psi_1(x_1) \prod_{k \sim 1, k \neq 2} M_{k \rightarrow 1}(x_1)$$

**Rule:** Ignore “incoming messages” to node  $i$  if there are none

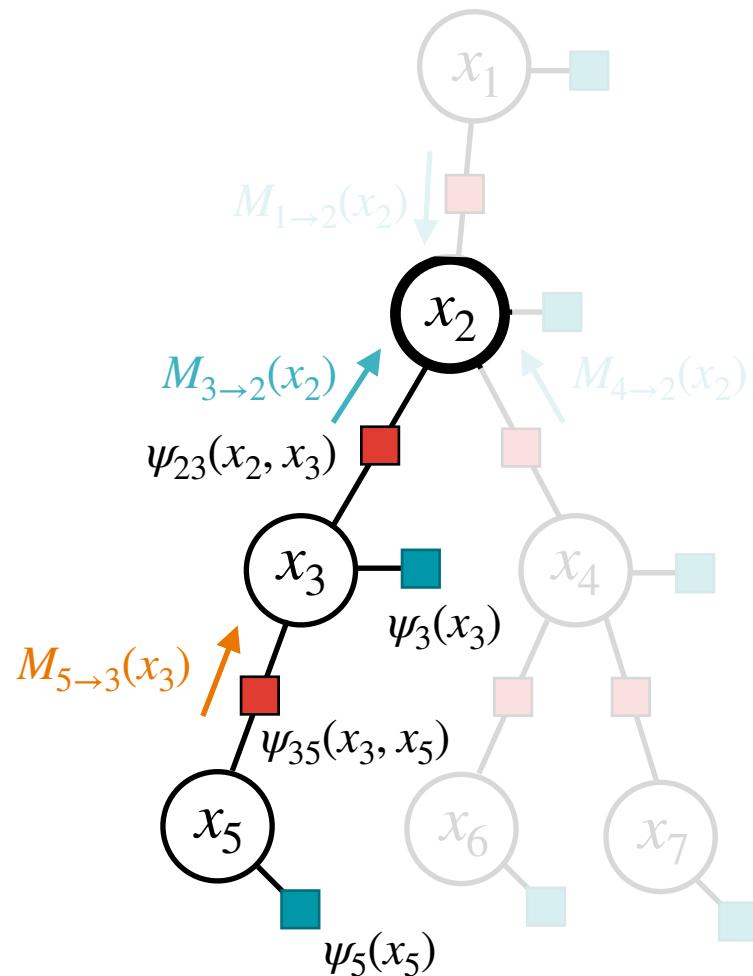
# Step 1. Message update



Next, compute the message  $x_3 \rightarrow x_2$ :

$$M_{3 \rightarrow 2}(x_2) = \sum_{x_3 \in \{1, \dots, K\}} \psi_{23}(x_2, x_3) \psi_3(x_3) \underbrace{\prod_{k \sim 3, k \neq 2} M_{k \rightarrow 3}(x_3)}_{??}$$

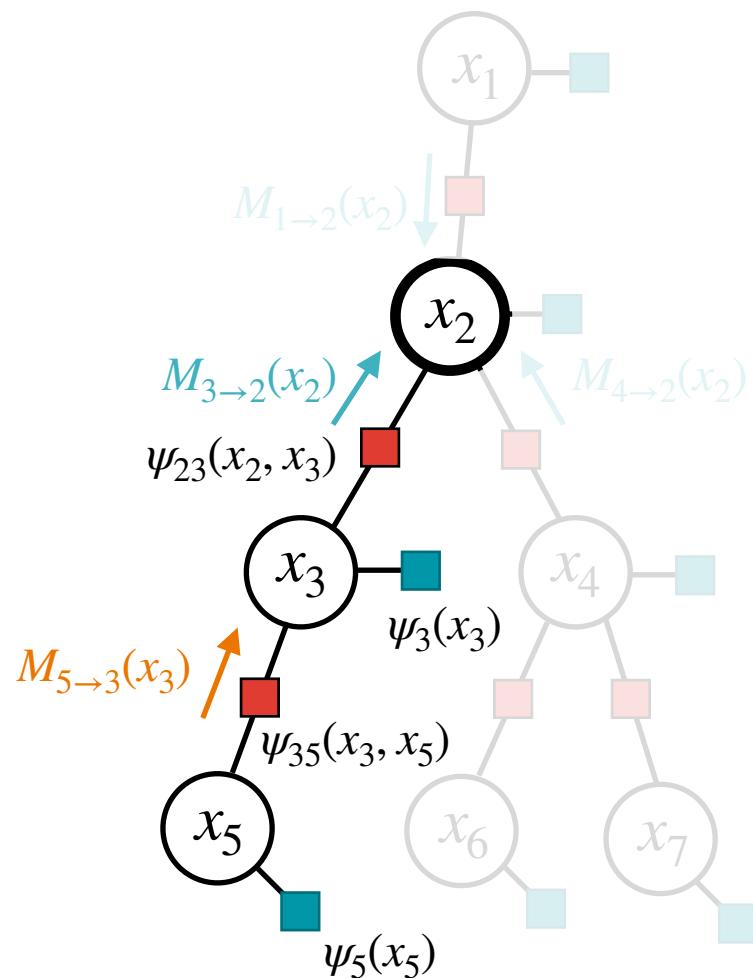
# Step 1. Message update



Next, compute the message  $x_3 \rightarrow x_2$ :

$$M_{3 \rightarrow 2}(x_2) = \sum_{x_3 \in \{1, \dots, K\}} \psi_{23}(x_2, x_3) \psi_3(x_3) M_{5 \rightarrow 3}(x_3)$$

# Step 1. Message update

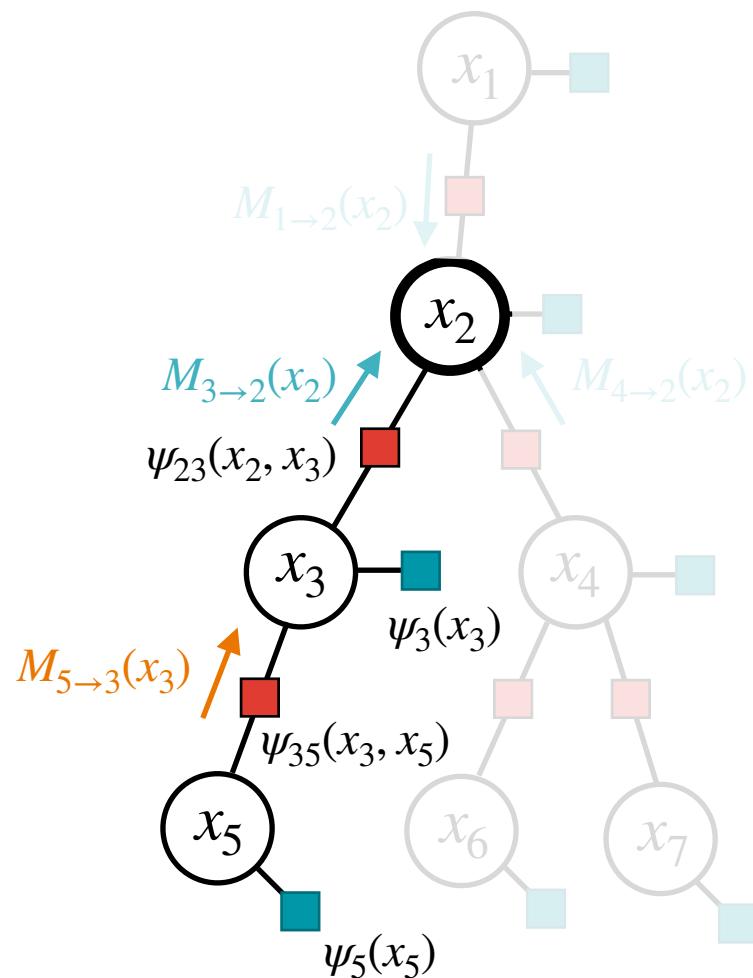


Next, compute the message  $x_3 \rightarrow x_2$ :

$$M_{3\rightarrow 2}(x_2) = \sum_{x_3 \in \{1, \dots, K\}} \psi_{23}(x_2, x_3) \psi_3(x_3) M_{5\rightarrow 3}(x_3)$$

$$M_{5\rightarrow 3}(x_3) = \sum_{x_5 \in \{1, \dots, K\}} \psi_{35}(x_3, x_5) \psi_5(x_5) \prod_{k \sim 5, k \neq 3} M_{k\rightarrow 5}(x_5)$$

# Step 1. Message update

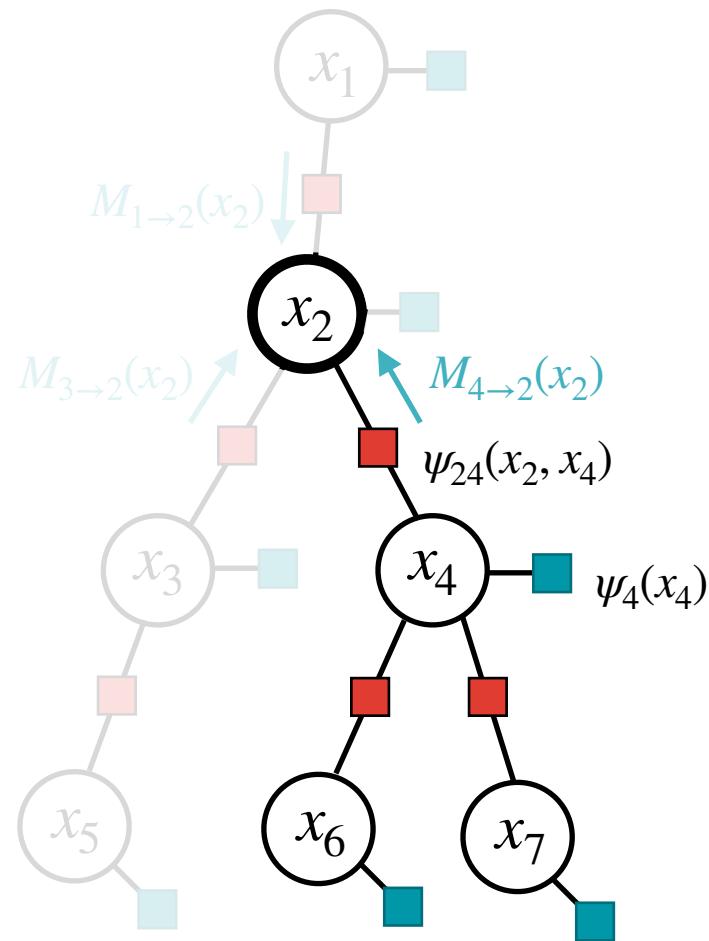


Next, compute the message  $x_3 \rightarrow x_2$ :

$$M_{3 \rightarrow 2}(x_2) = \sum_{x_3 \in \{1, \dots, K\}} \psi_{23}(x_2, x_3) \psi_3(x_3) M_{5 \rightarrow 3}(x_3)$$

$$M_{5 \rightarrow 3}(x_3) = \sum_{x_5 \in \{1, \dots, K\}} \psi_{35}(x_3, x_5) \psi_5(x_5) \prod_{k \sim 5, k \neq 3} M_{k \rightarrow 5}(x_5)$$

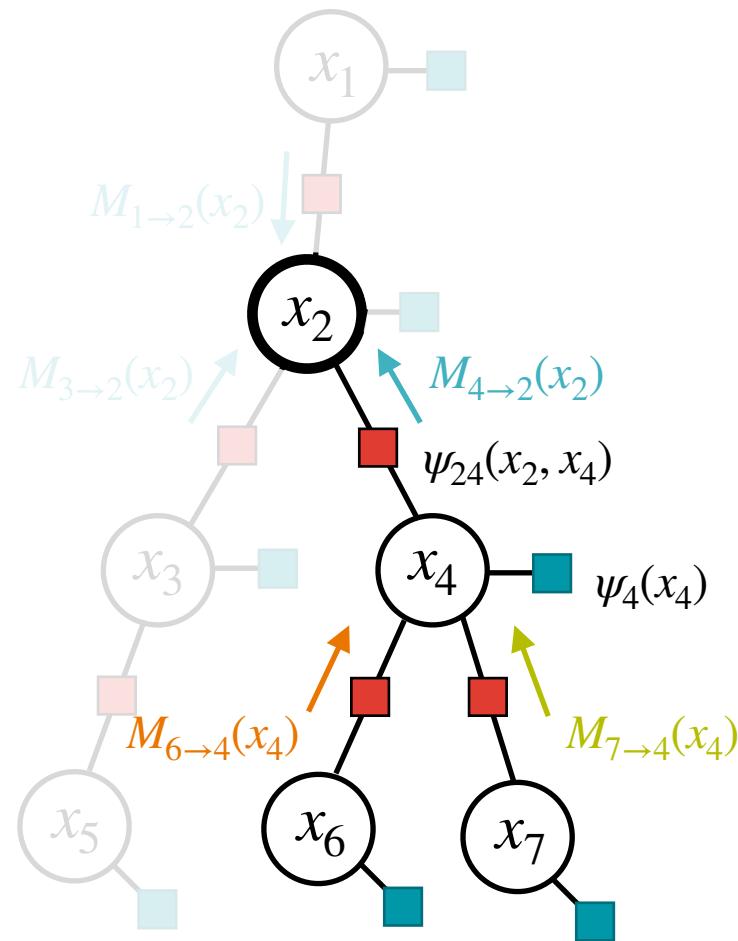
# Step 1. Message update



Finally, compute the message  $x_4 \rightarrow x_2$ :

$$M_{4 \rightarrow 2}(x_2) = \sum_{x_4 \in \{1, \dots, K\}} \psi_{24}(x_2, x_4) \psi_4(x_4) \underbrace{\prod_{k \sim 4, k \neq 2} M_{k \rightarrow 4}(x_4)}_{??}$$

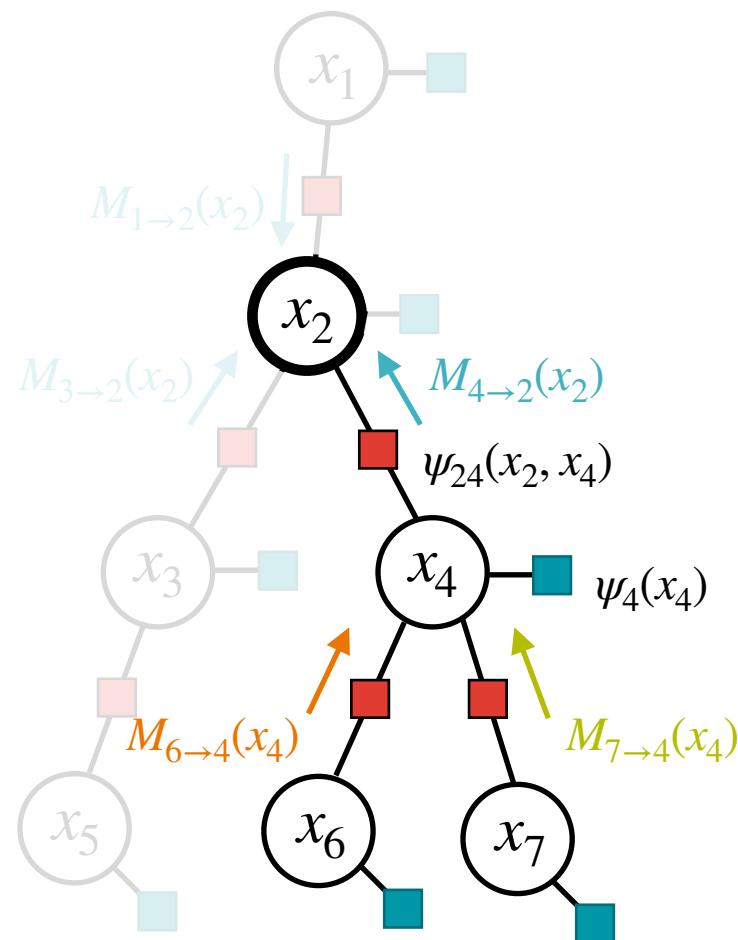
# Step 1. Message update



Finally, compute the message  $x_4 \rightarrow x_2$ :

$$M_{4 \rightarrow 2}(x_2) = \sum_{x_4 \in \{1, \dots, K\}} \psi_{24}(x_2, x_4) \psi_4(x_4) M_{6 \rightarrow 4}(x_4) M_{7 \rightarrow 4}(x_4)$$

# Step 1. Message update



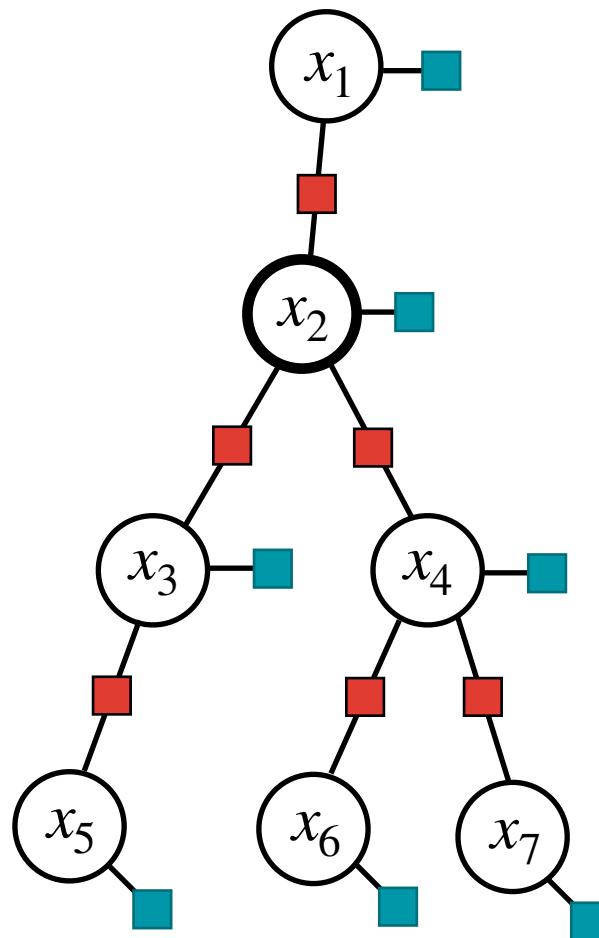
Finally, compute the message  $x_4 \rightarrow x_2$ :

$$M_{4 \rightarrow 2}(x_2) = \sum_{x_4 \in \{1, \dots, K\}} \psi_{24}(x_2, x_4) \psi_4(x_4) M_{6 \rightarrow 4}(x_4) M_{7 \rightarrow 4}(x_4)$$

$$M_{6 \rightarrow 4}(x_4) = \sum_{x_6 \in \{1, \dots, K\}} \psi_{46}(x_4, x_6) \psi_6(x_6)$$

$$M_{7 \rightarrow 4}(x_4) = \sum_{x_7 \in \{1, \dots, K\}} \psi_{47}(x_4, x_7) \psi_7(x_7)$$

# Belief propagation algorithm



BP proceeds by iteratively updating:

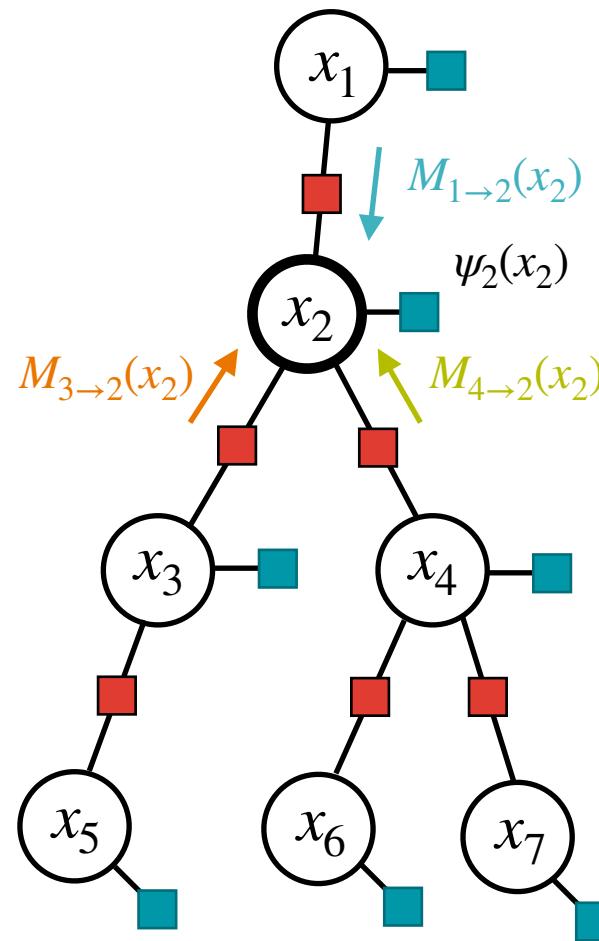
1. The “messages” between two nodes

$$M_{j \rightarrow i}(x_i) \rightarrow \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)$$

2. The “state” of each node

$$p(x_i) \rightarrow \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i)$$

## Step 2. State update



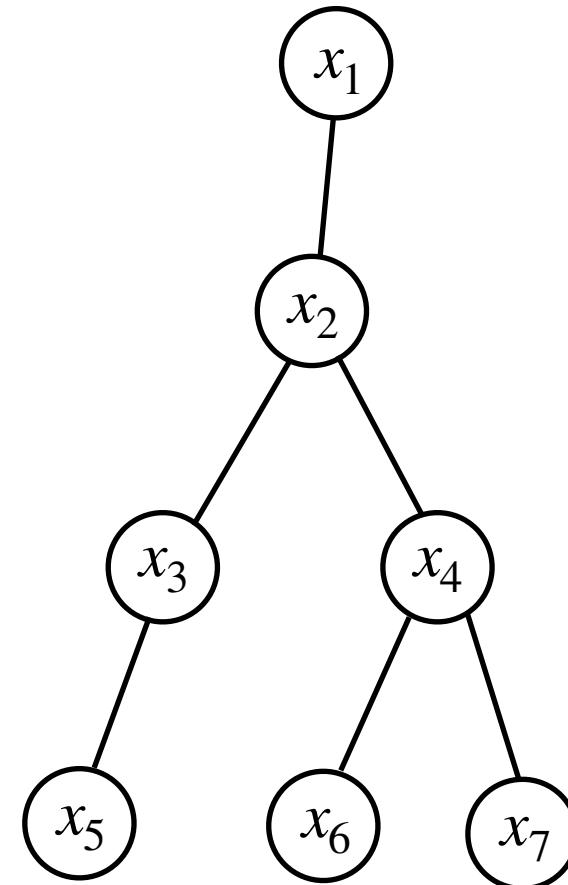
Now we can compute  $p(x_2)$ :

$$p(x_2) = \frac{1}{Z} \psi_2(x_2) \times M_{1 \rightarrow 2}(x_2) \times M_{3 \rightarrow 2}(x_2) \times M_{4 \rightarrow 2}(x_2)$$

where

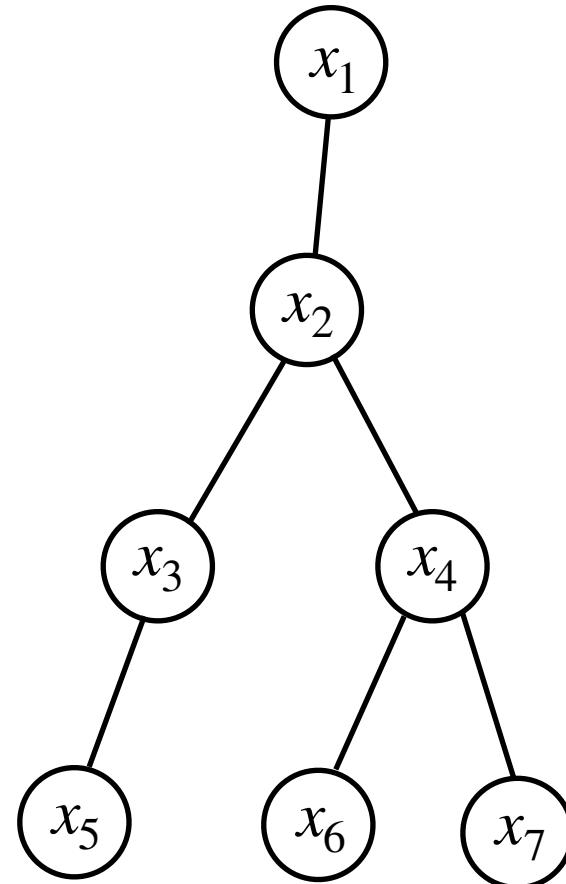
$$Z = \sum_{x_2 \in \{1, \dots, K\}} \psi_2(x_2) \times M_{1 \rightarrow 2}(x_2) \times M_{3 \rightarrow 2}(x_2) \times M_{4 \rightarrow 2}(x_2)$$

# Efficient implementation



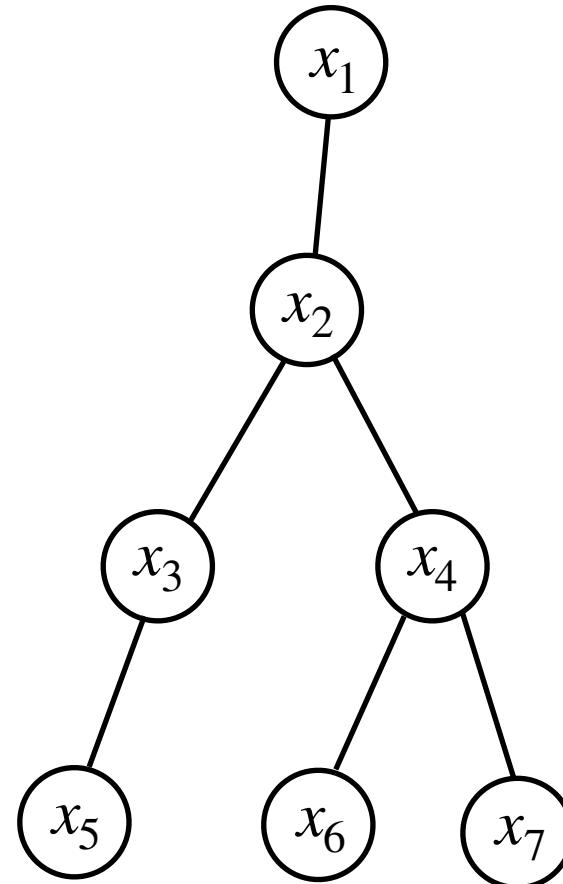
# Efficient implementation

Exploiting the tree-structure, we can compute *all* the marginals efficiently



# Efficient implementation

**Step 0.** Initialise



# Efficient implementation

**Step 0.** Initialise

- the **states** as

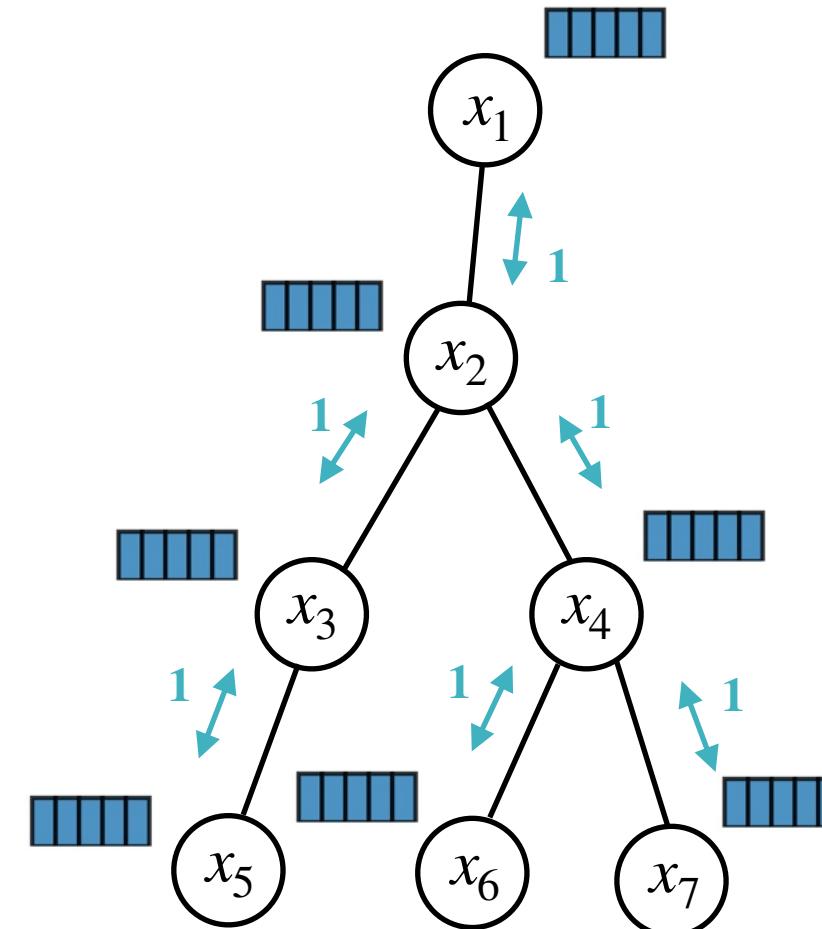
$$p(x_i) = \frac{1}{K} \mathbf{1},$$

for all  $i \in V$ , and

- the **messages** as

$$M_{j \rightarrow i}(x_i) = 1$$

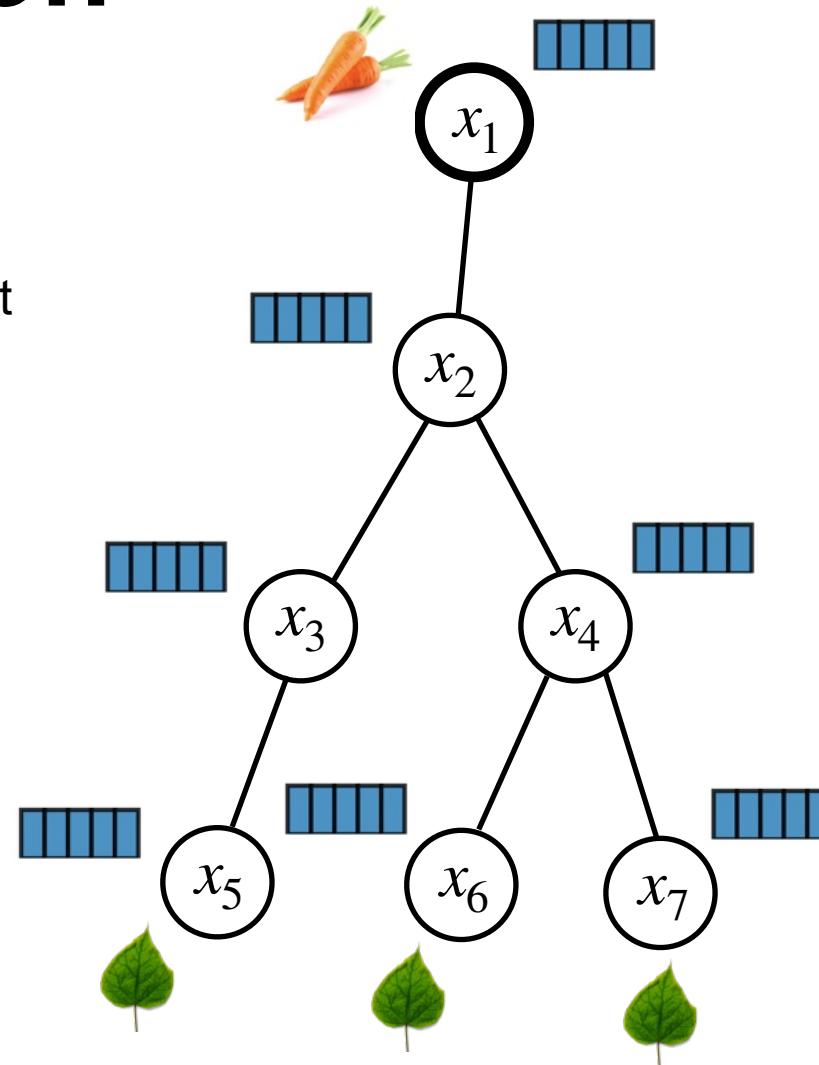
for all  $(i, j) \in E$ .



# Efficient implementation

**Step 1.** Choose a “**root**” node and identify the corresponding “**leaf**” nodes

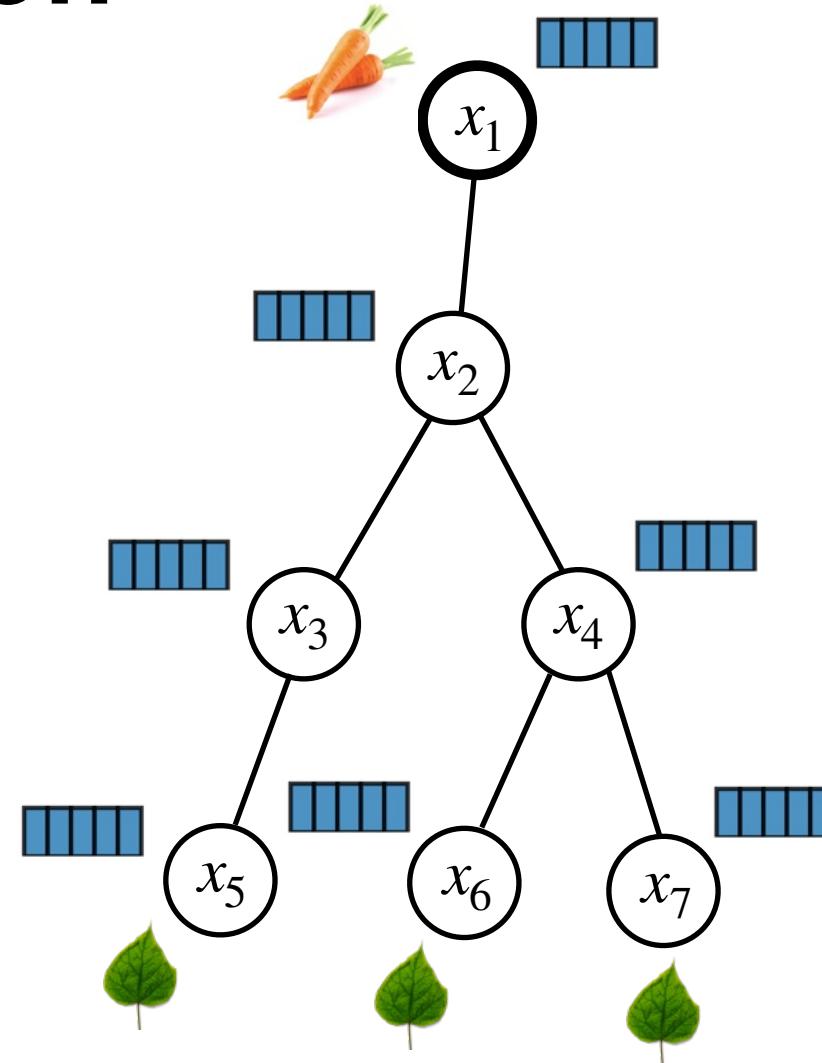
**Note:** The leaves are the furthest descendants of the root



# Efficient implementation

## Step 2. Update

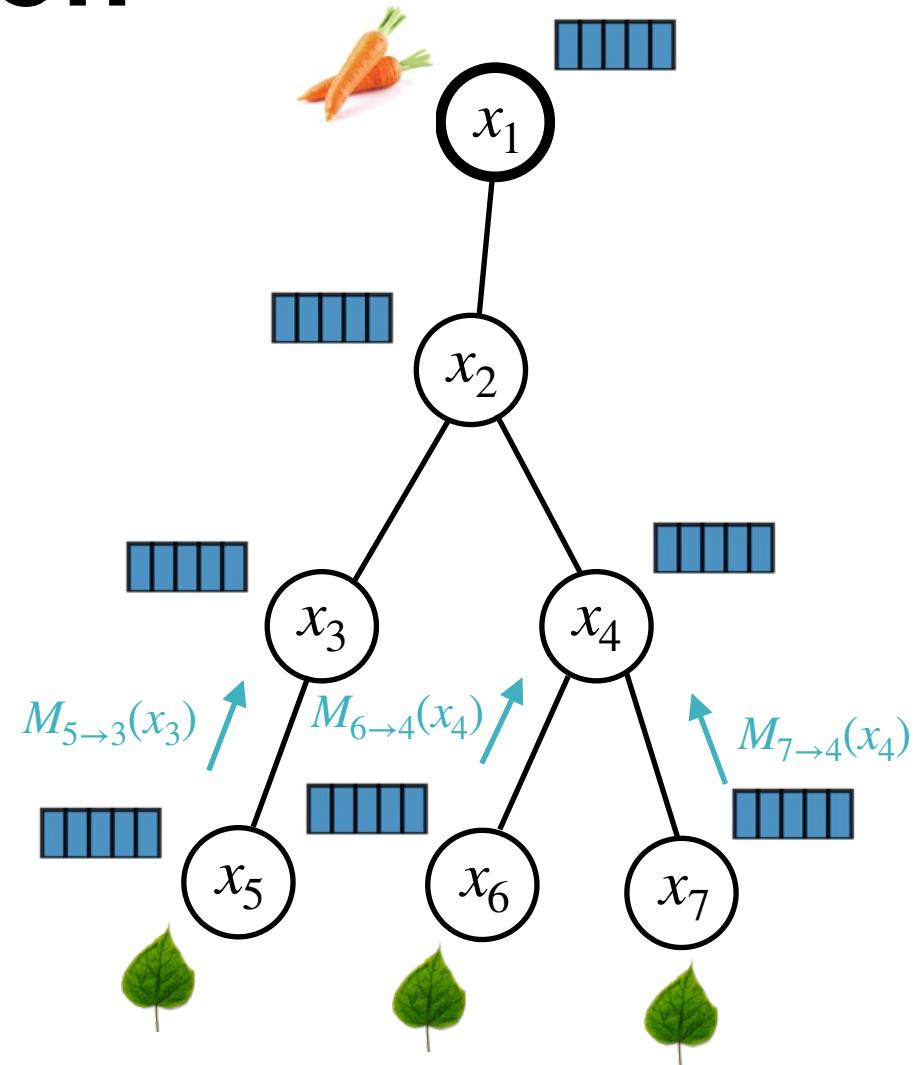
- all **messages** propagating from the leaf nodes, and
- all the **states** of their parent nodes



# Efficient implementation

## Step 2. Update

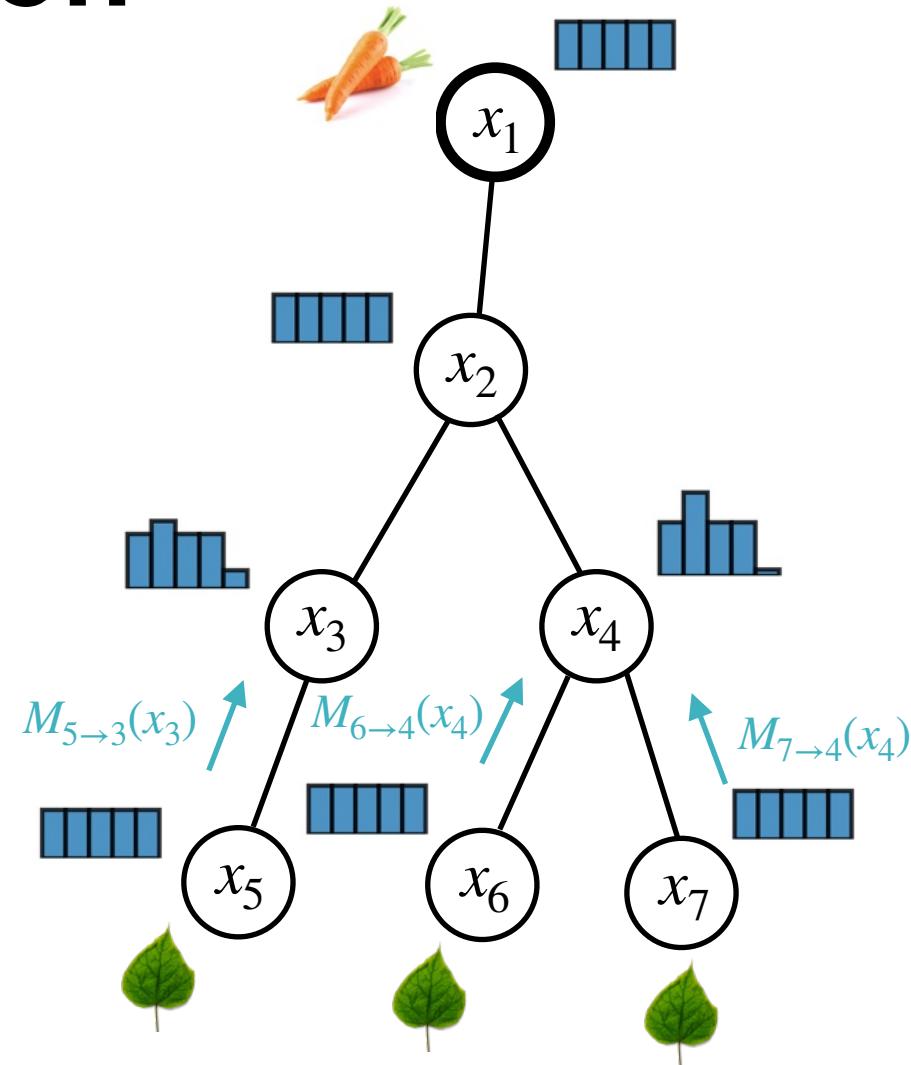
- all **messages** propagating from the leaf nodes, and
- all the **states** of their parent nodes



# Efficient implementation

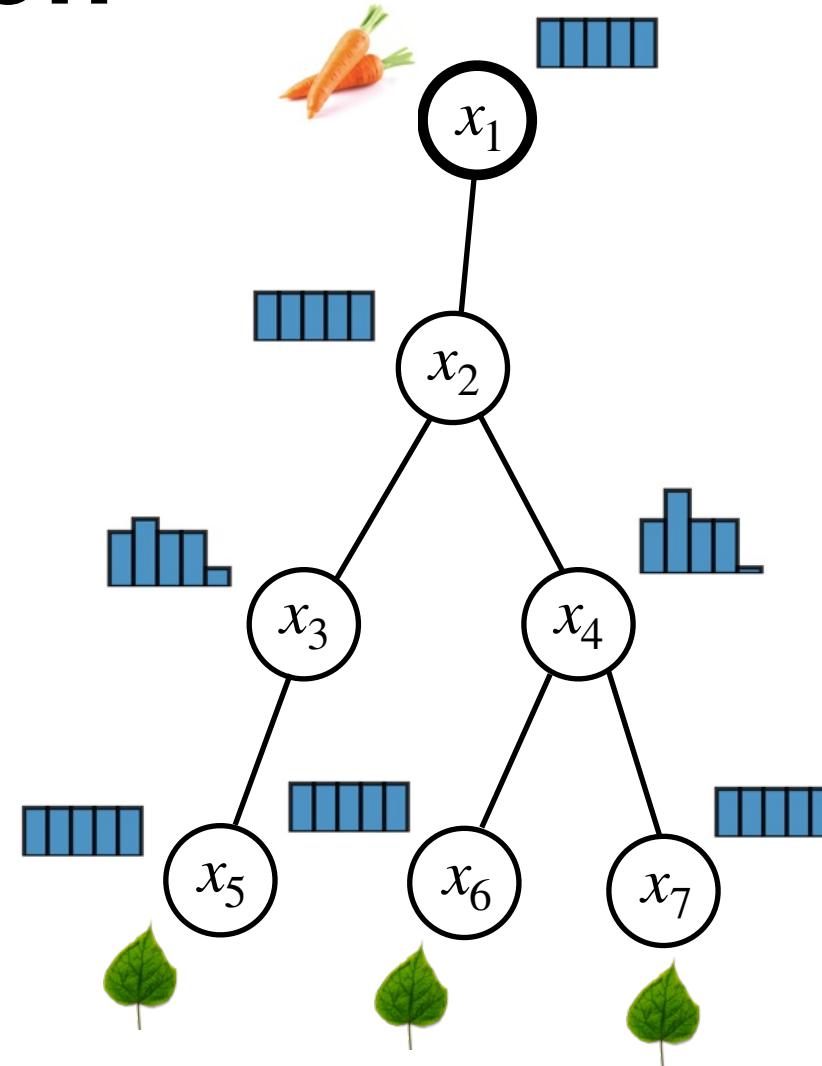
## Step 2. Update

- all **messages** propagating from the leaf nodes, and
- all the **states** of their parent nodes



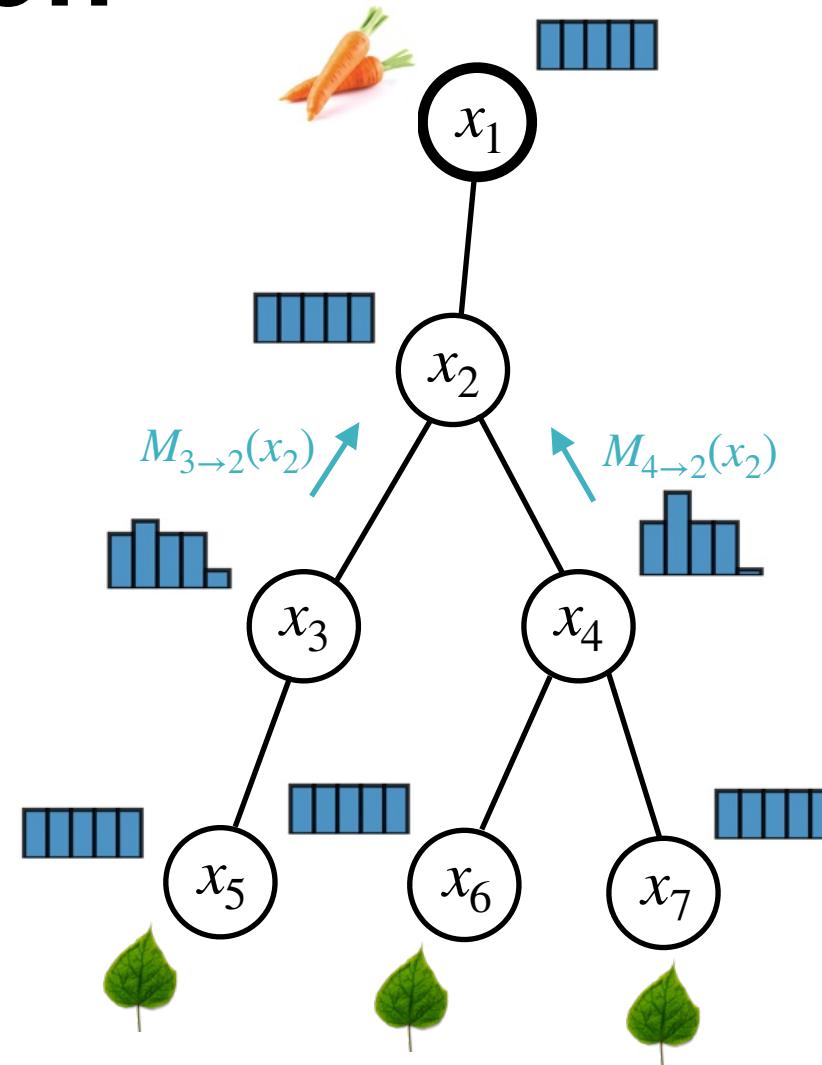
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



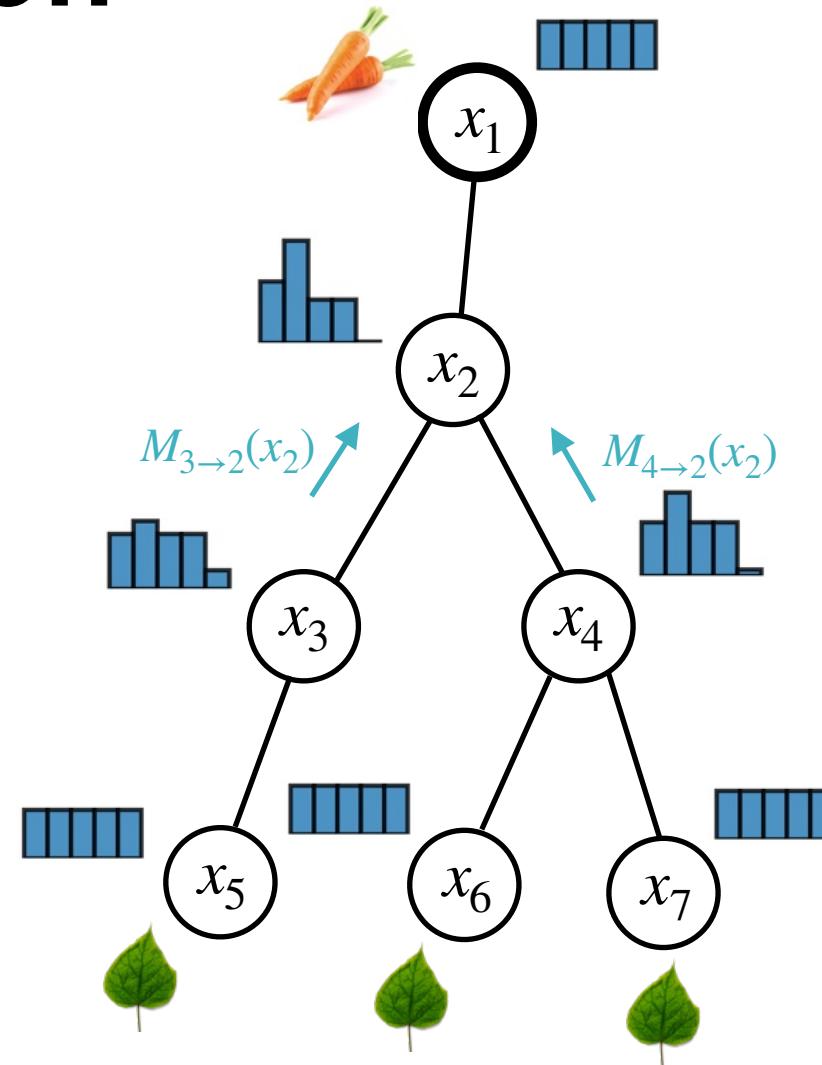
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



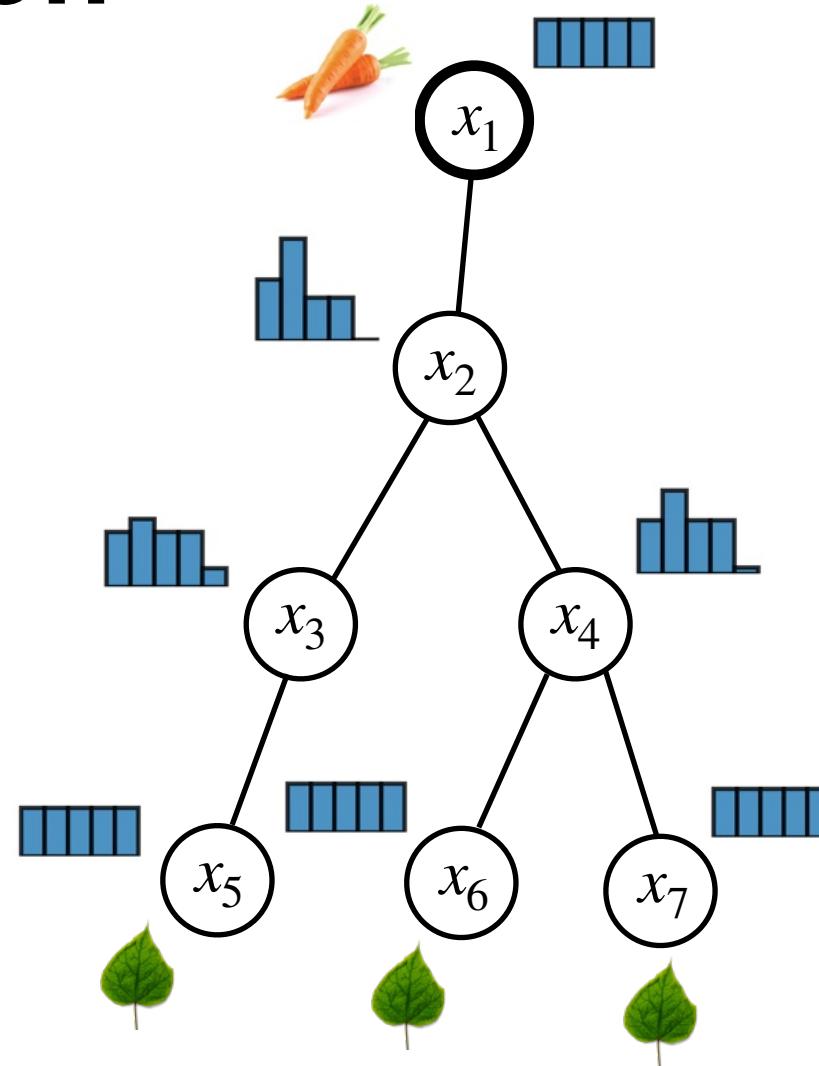
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



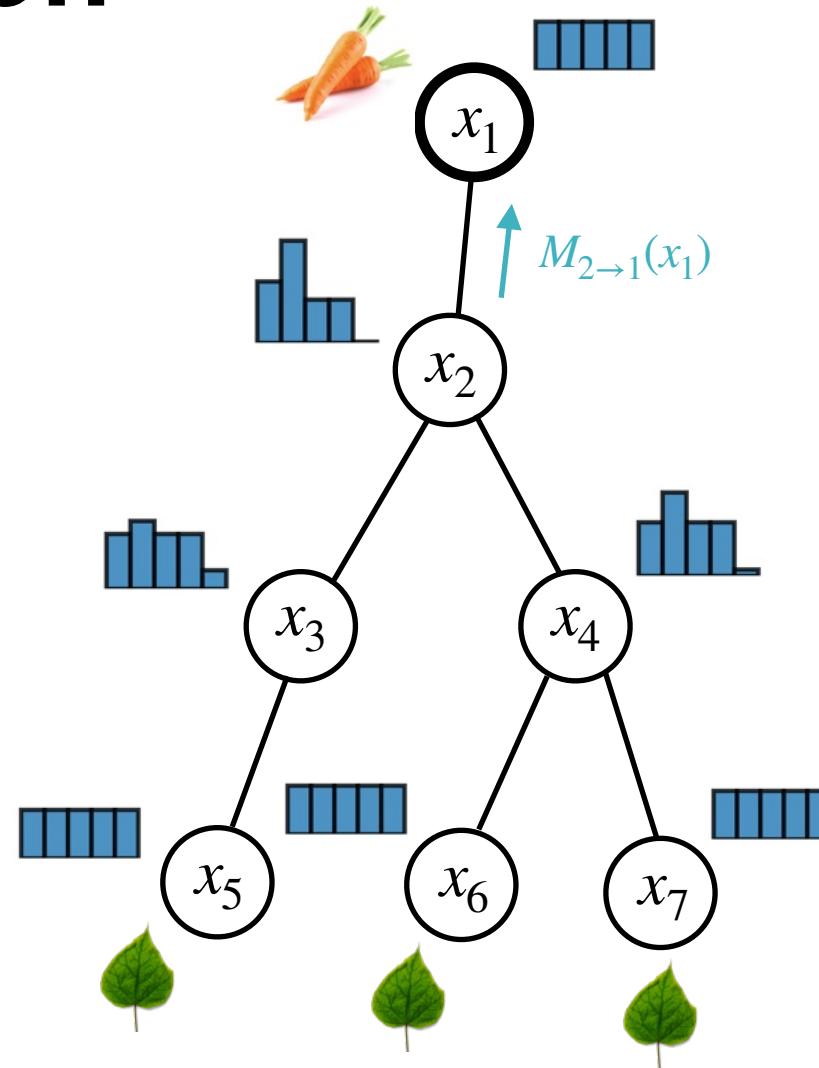
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



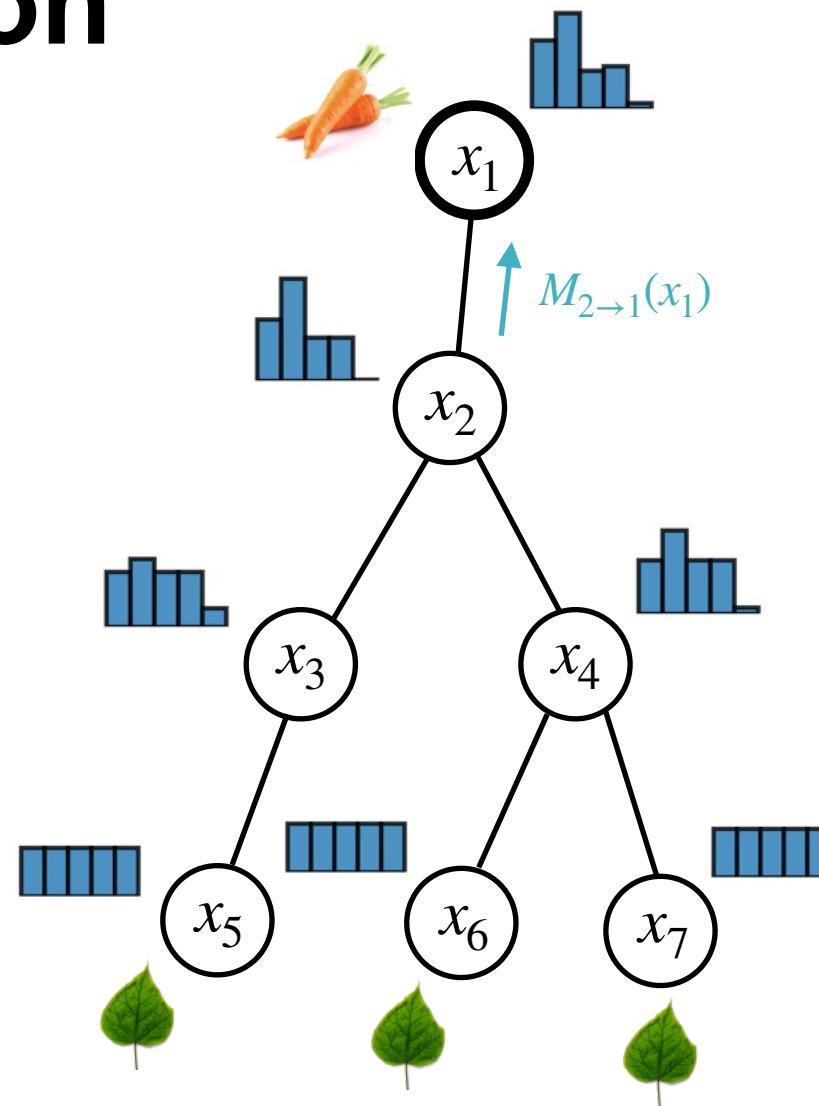
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



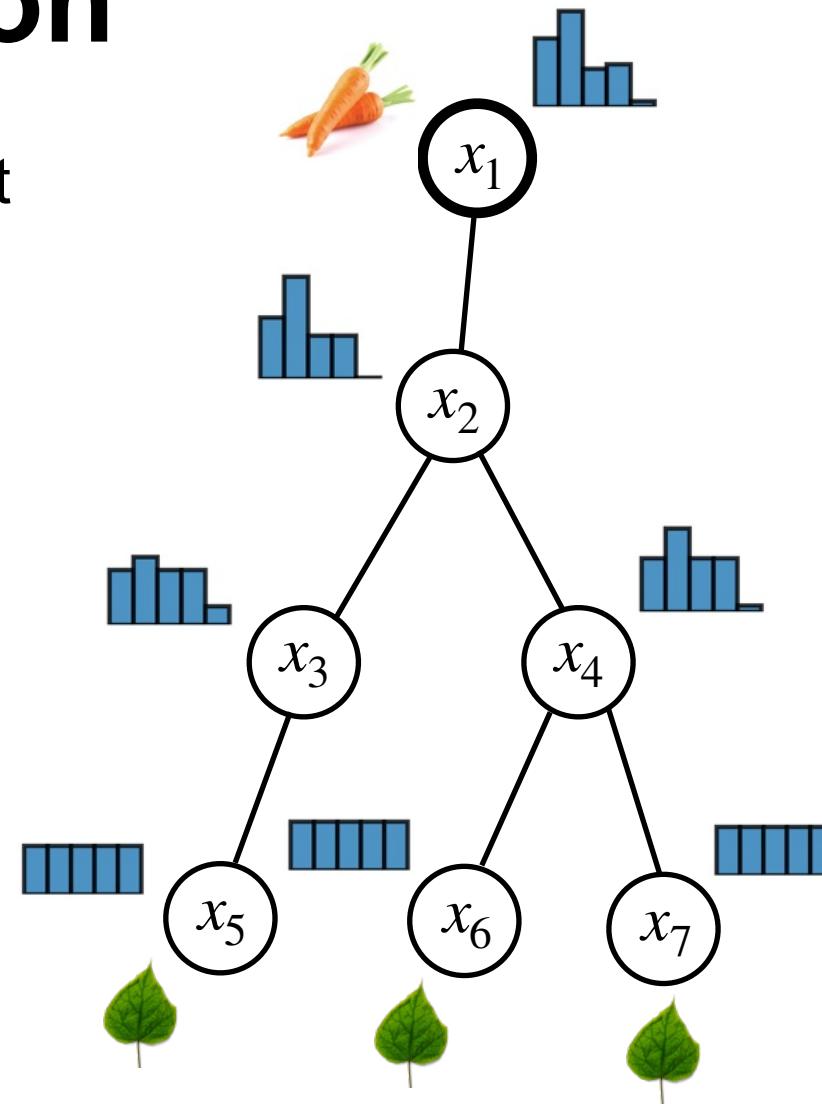
# Efficient implementation

**Step 3.** Update the **messages** and **states** all the way up to the root



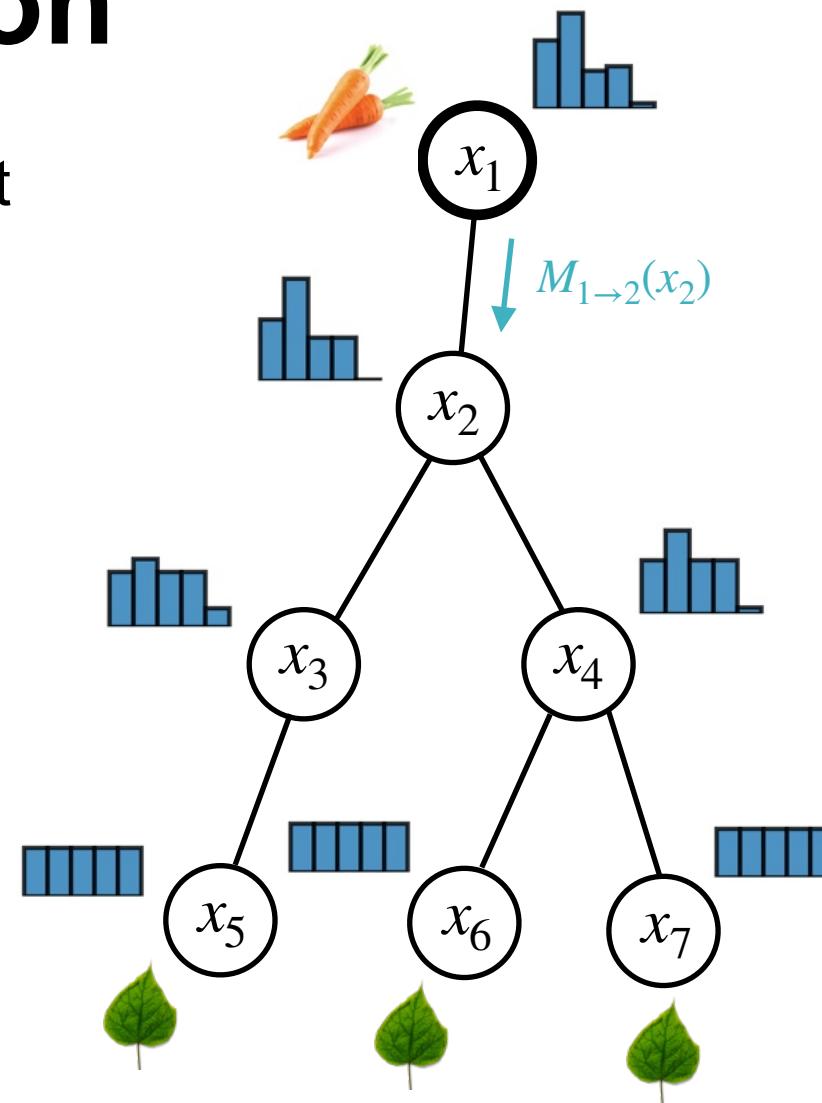
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



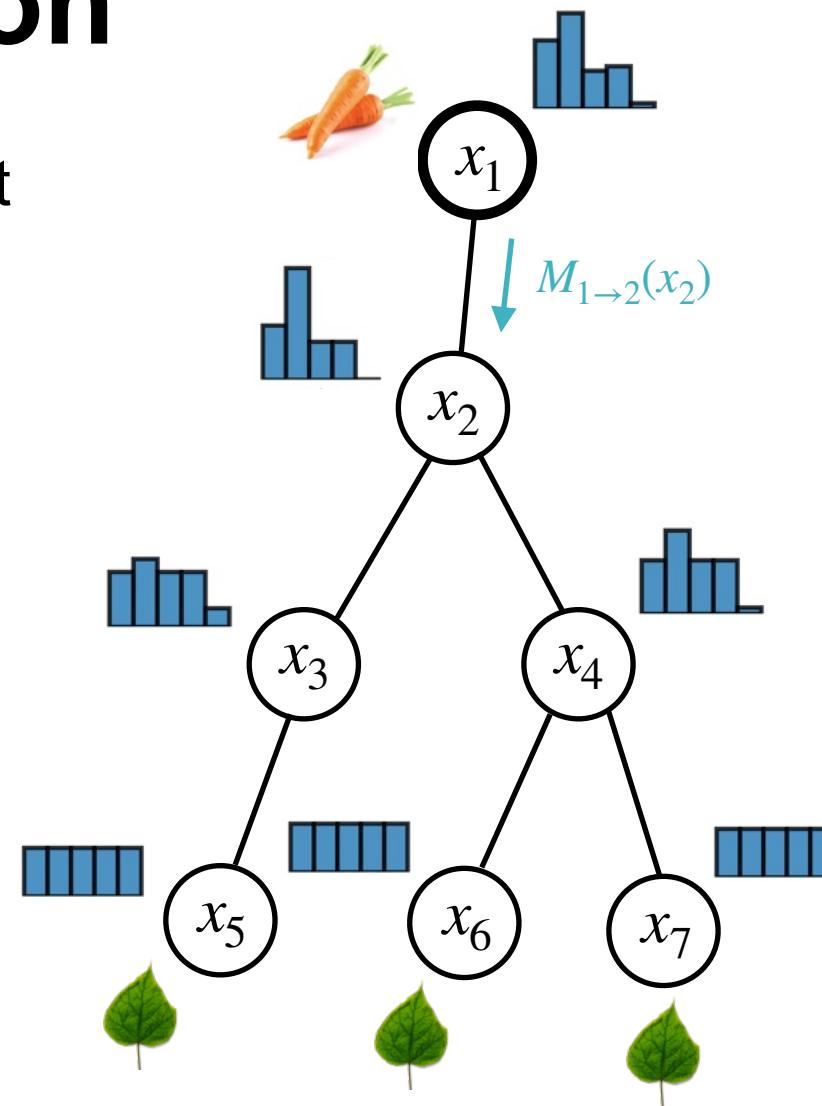
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



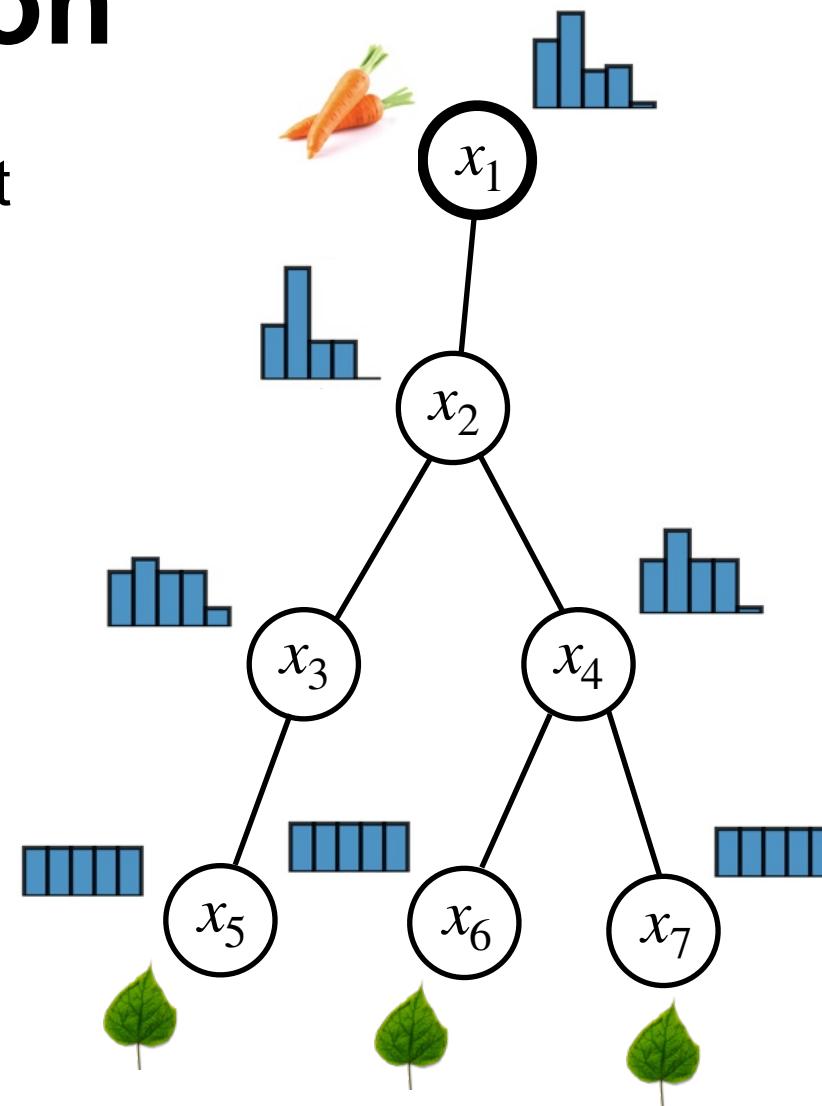
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



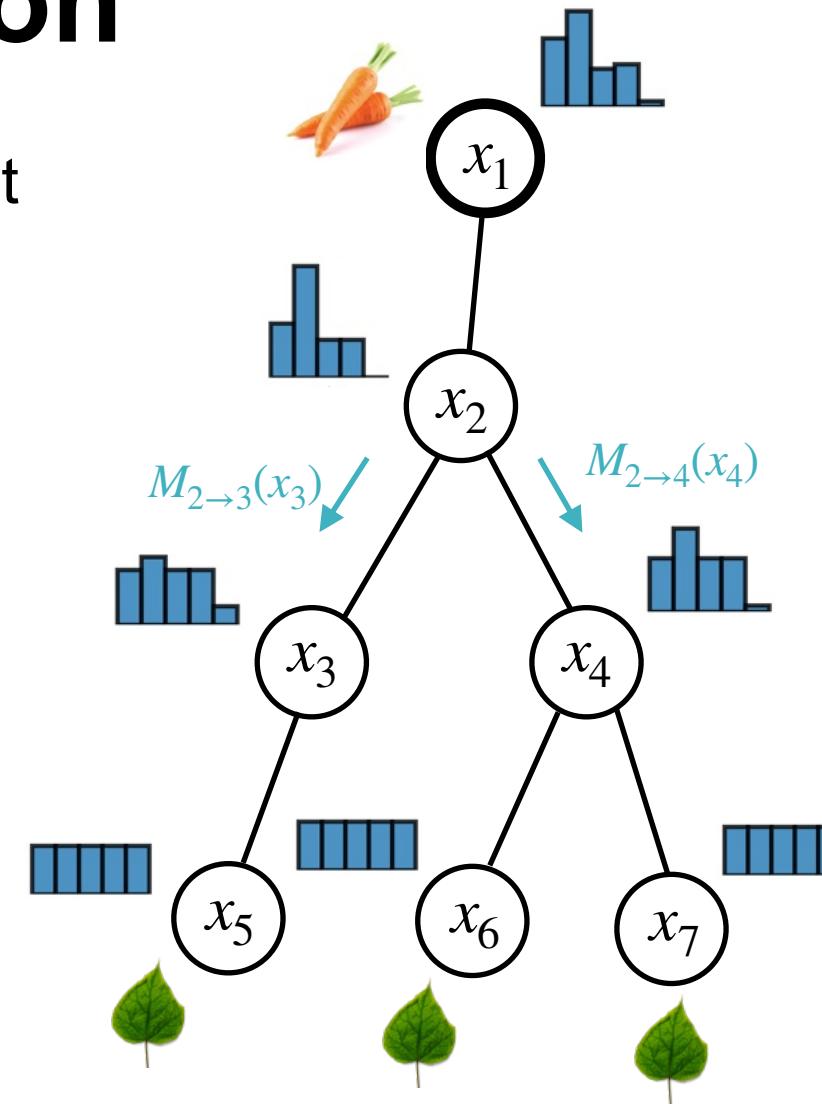
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



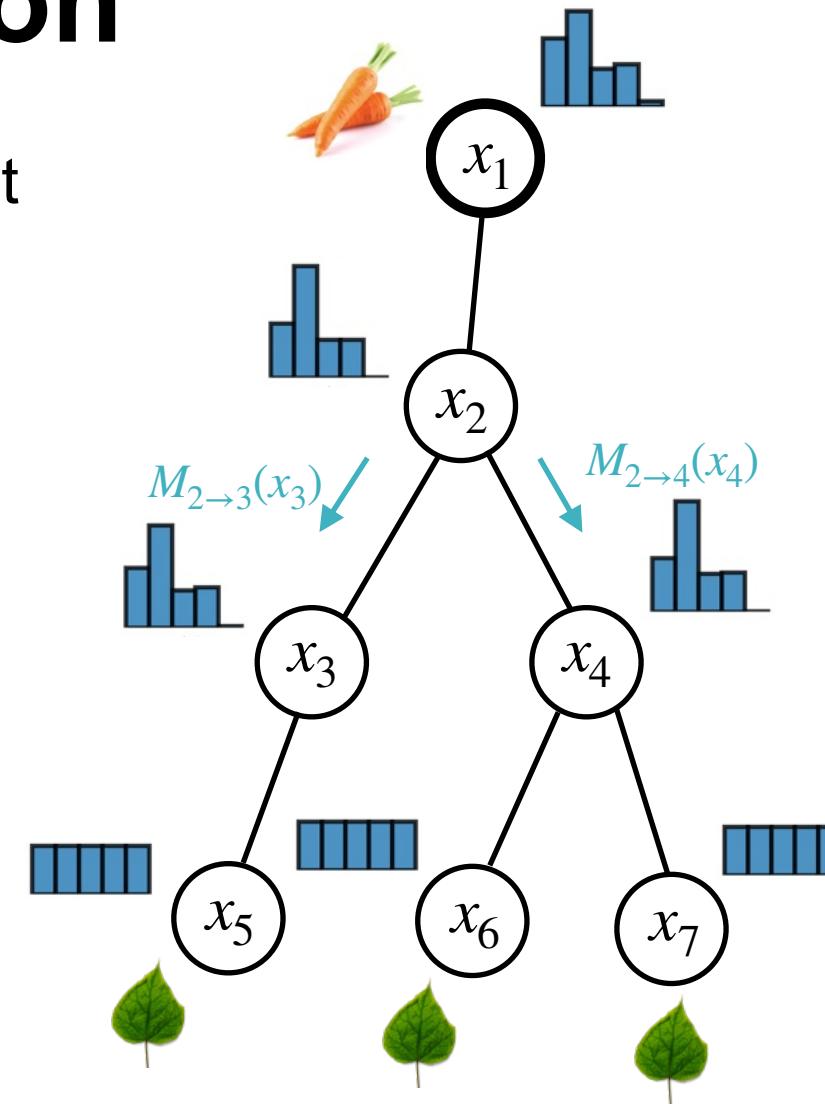
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



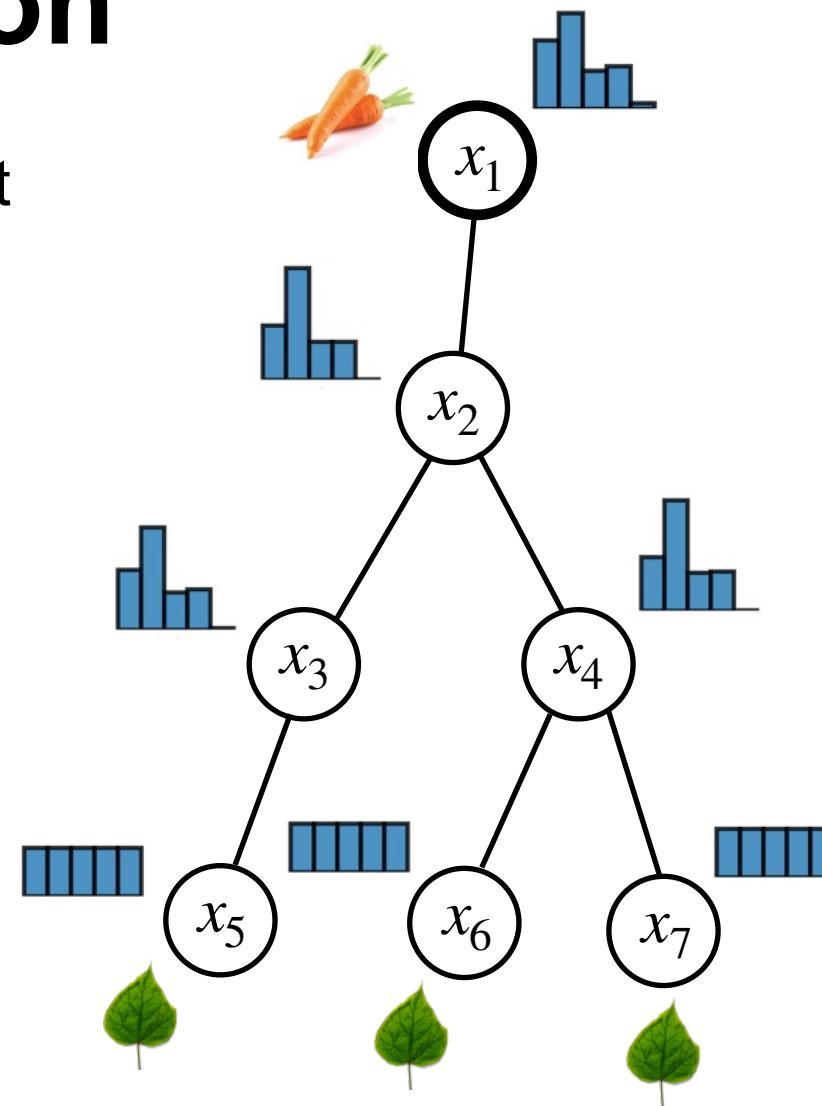
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



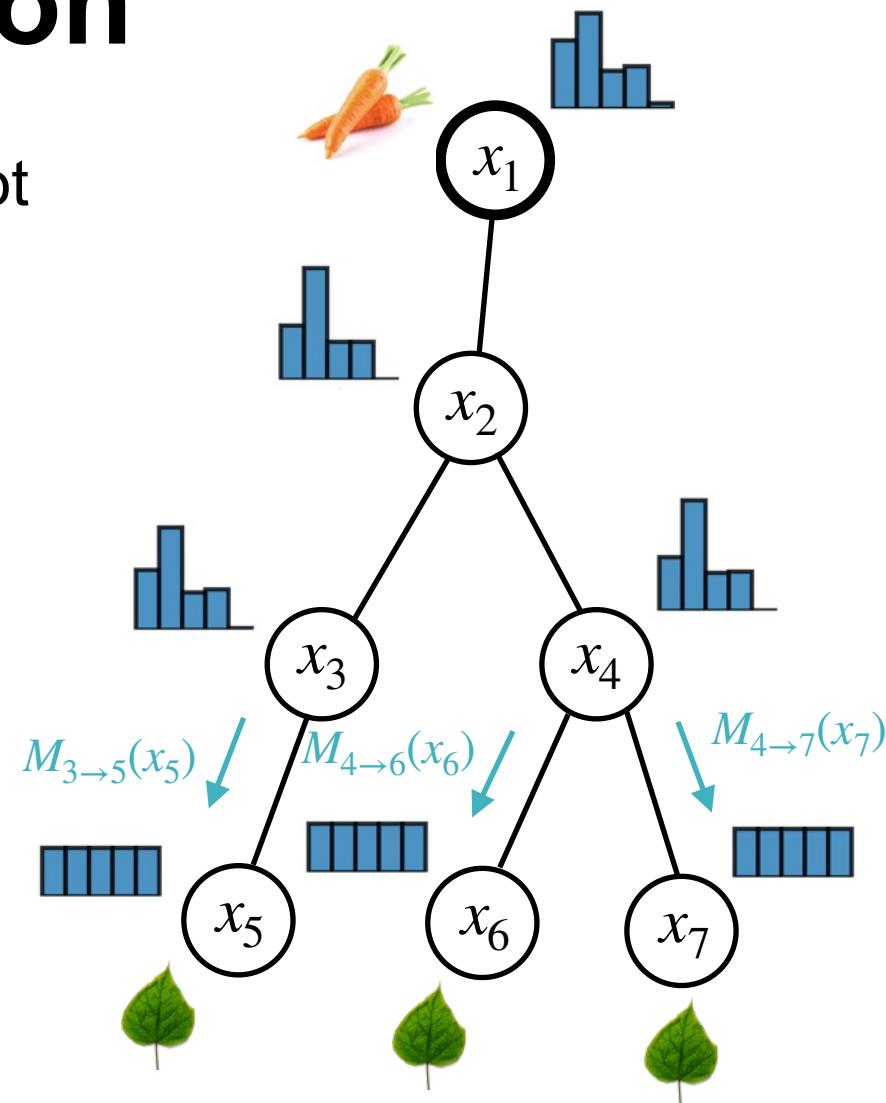
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



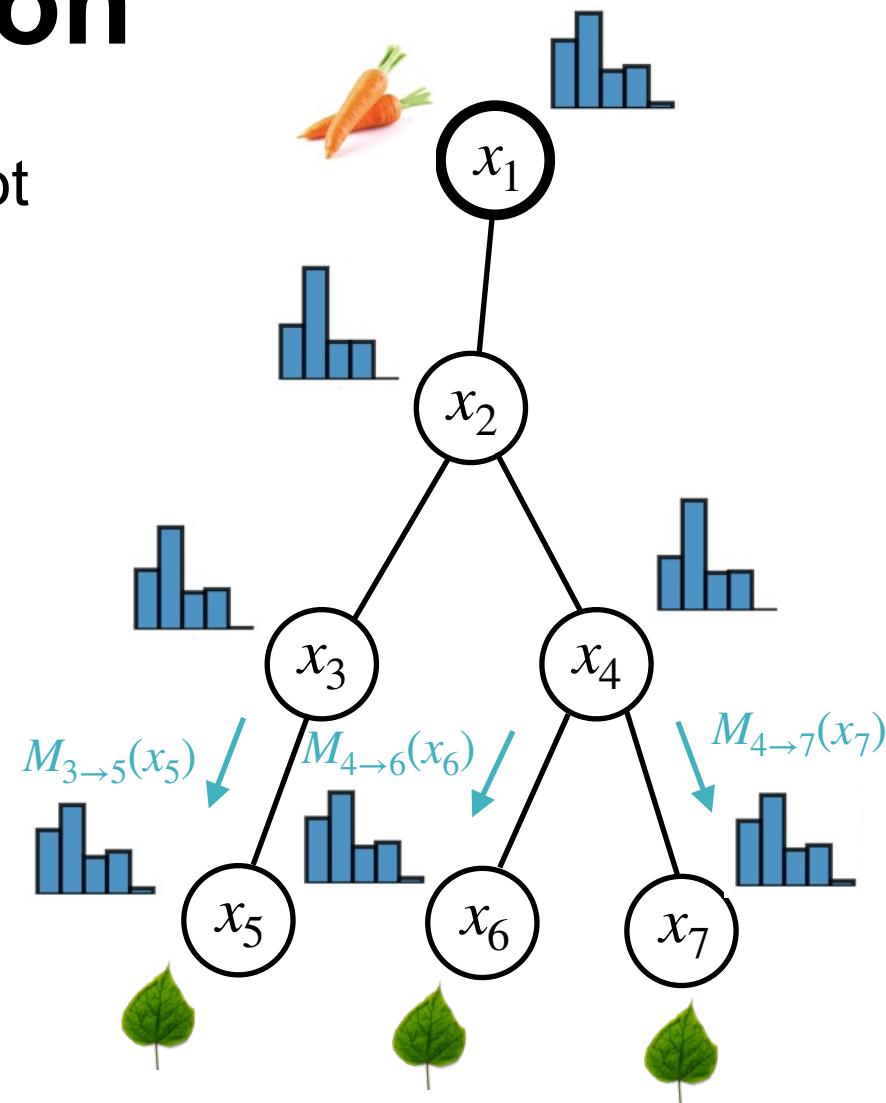
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves



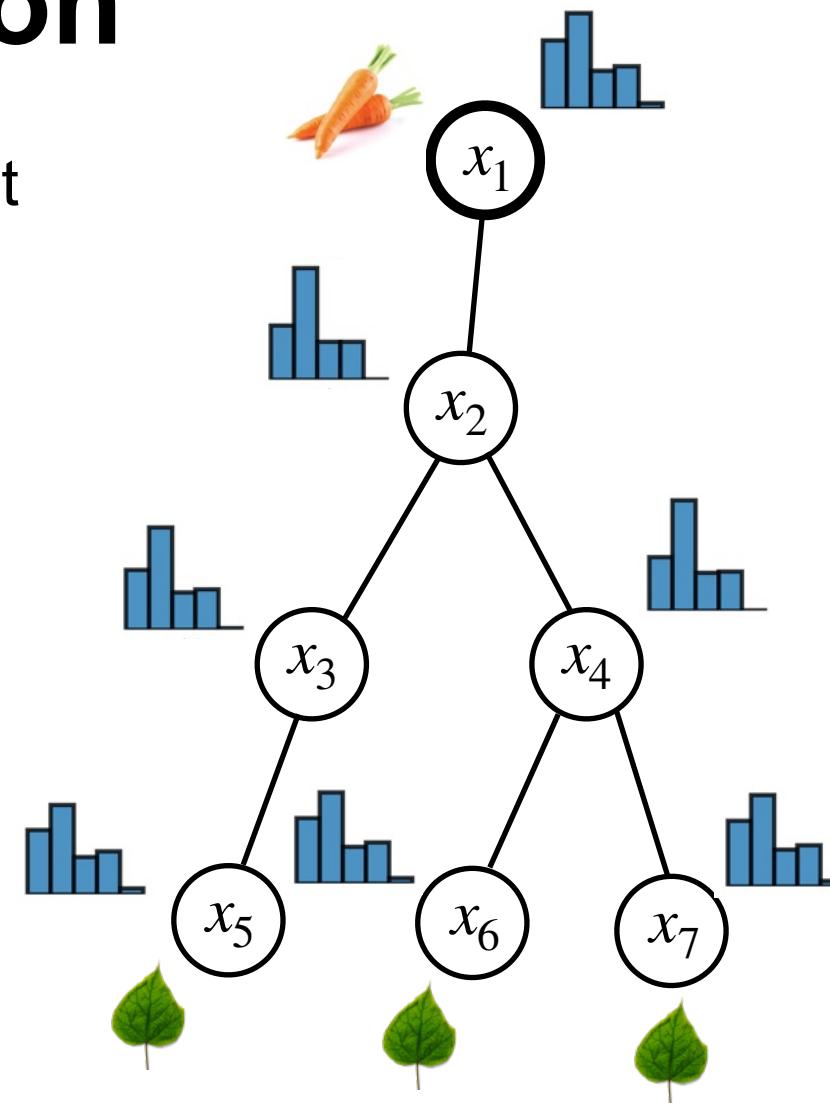
# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves

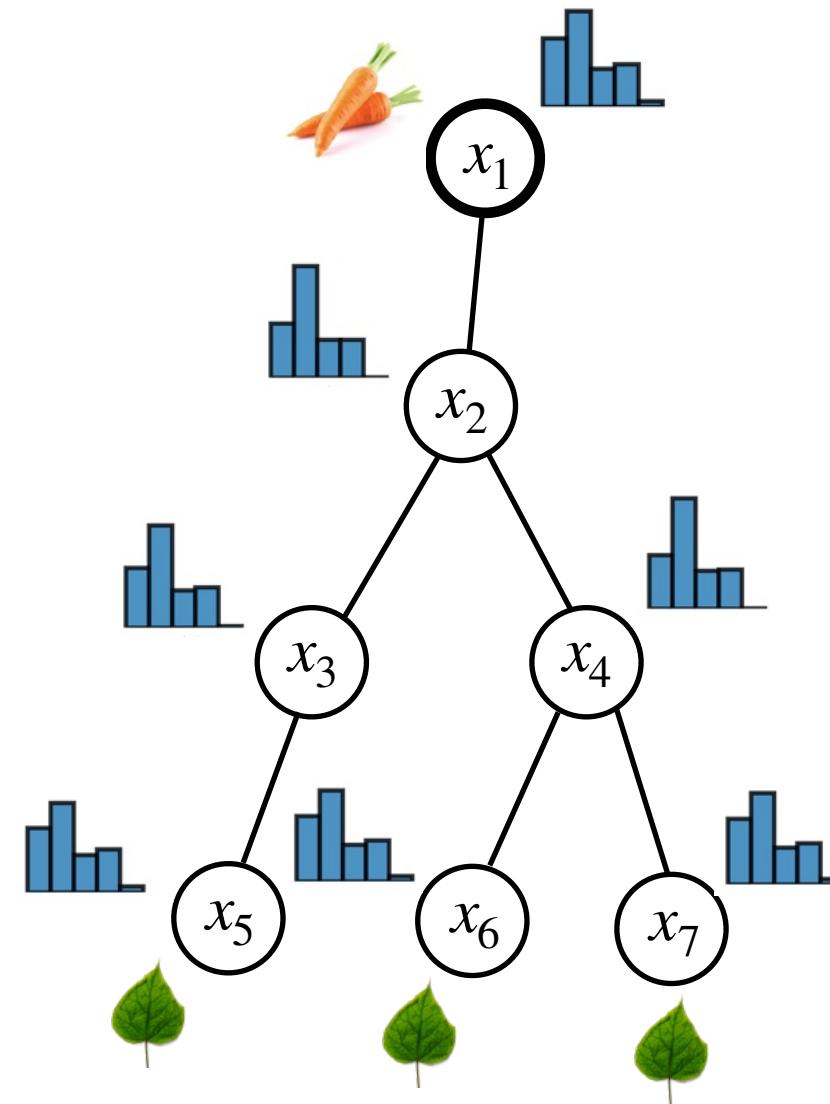


# Efficient implementation

**Step 4.** Do the same, starting from the root and going down to the leaves

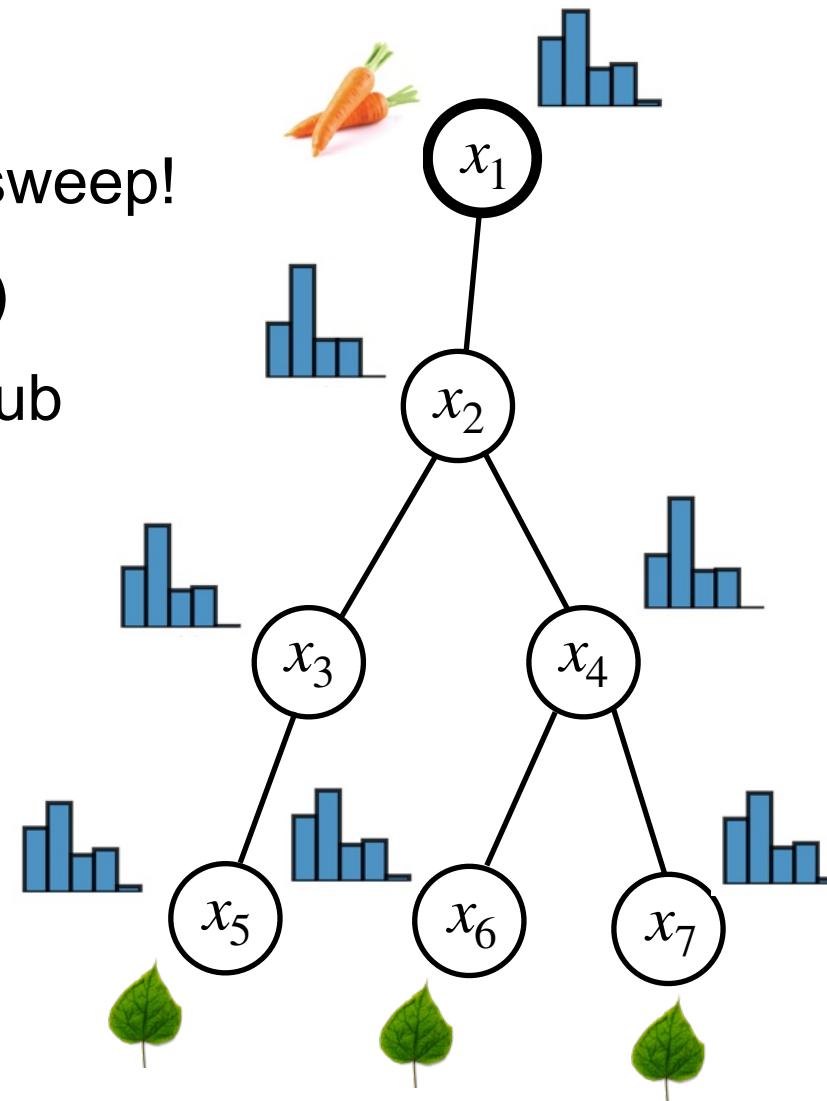


# Remarks



# Remarks

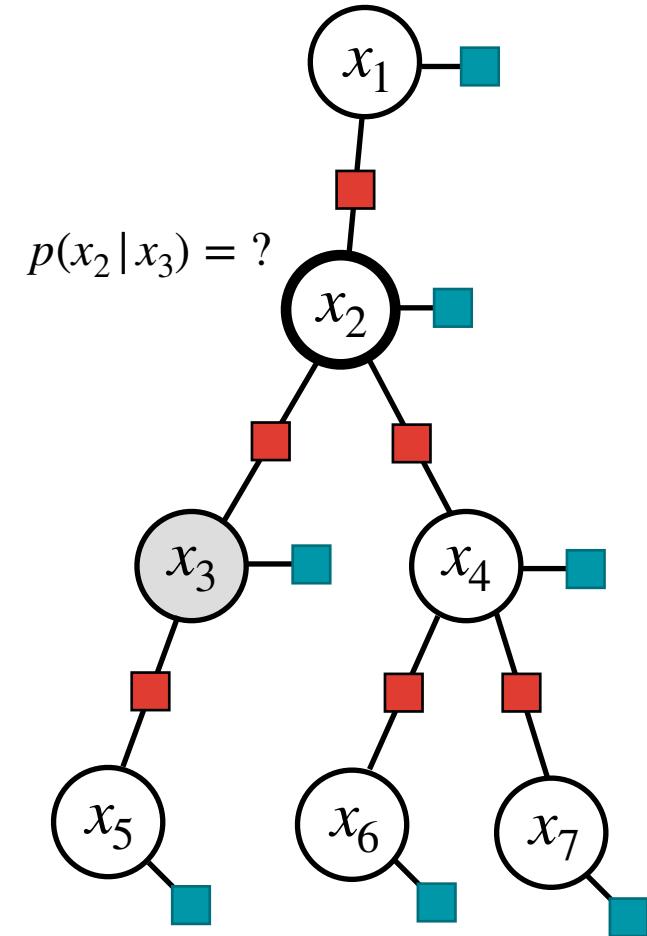
- Guaranteed convergence after a single sweep!
- *Linear* complexity in  $N$  (**not** exponential!)
- See example implementation in my GitHub



# Checklist

If  $G = (V, E)$  is a tree, can we compute:

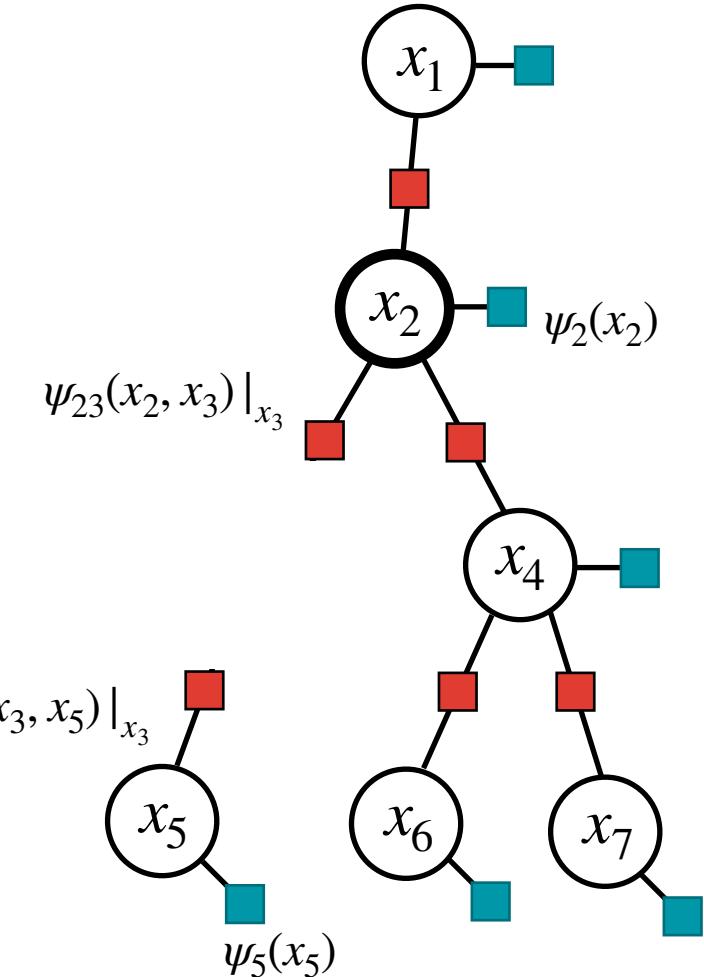
1. The marginal likelihood  $p(y)$  of observed data ✓
2. The marginal distribution  $p(z)$  of latent variables ✓
3. The conditional distribution  $p(x_i | x_j)$  for any  $i, j \in V$



# Checklist

If  $G = (V, E)$  is a tree, can we compute:

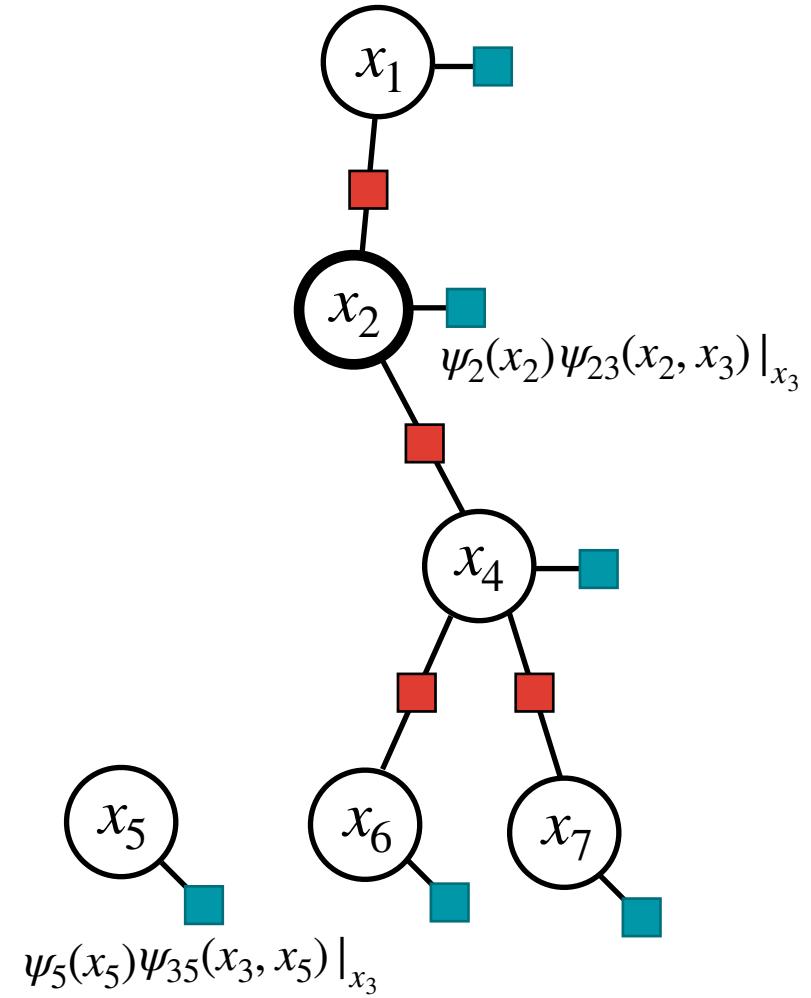
1. The marginal likelihood  $p(y)$  of observed data ✓
2. The marginal distribution  $p(z)$  of latent variables ✓
3. The conditional distribution  $p(x_i | x_j)$  for any  $i, j \in V$



# Checklist

If  $G = (V, E)$  is a tree, can we 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  $i, j \in V$

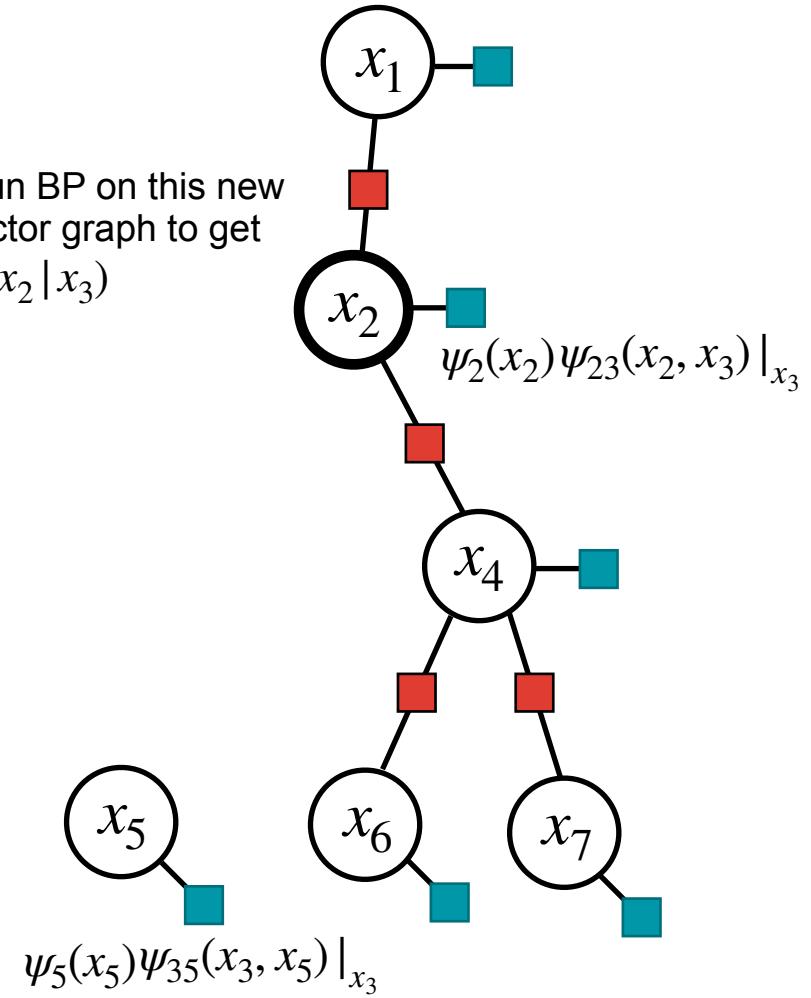


# Checklist

If  $G = (V, E)$  is a tree, can we 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  $i, j \in V$

Run BP on this new factor graph to get  
 $p(x_2 | x_3)$

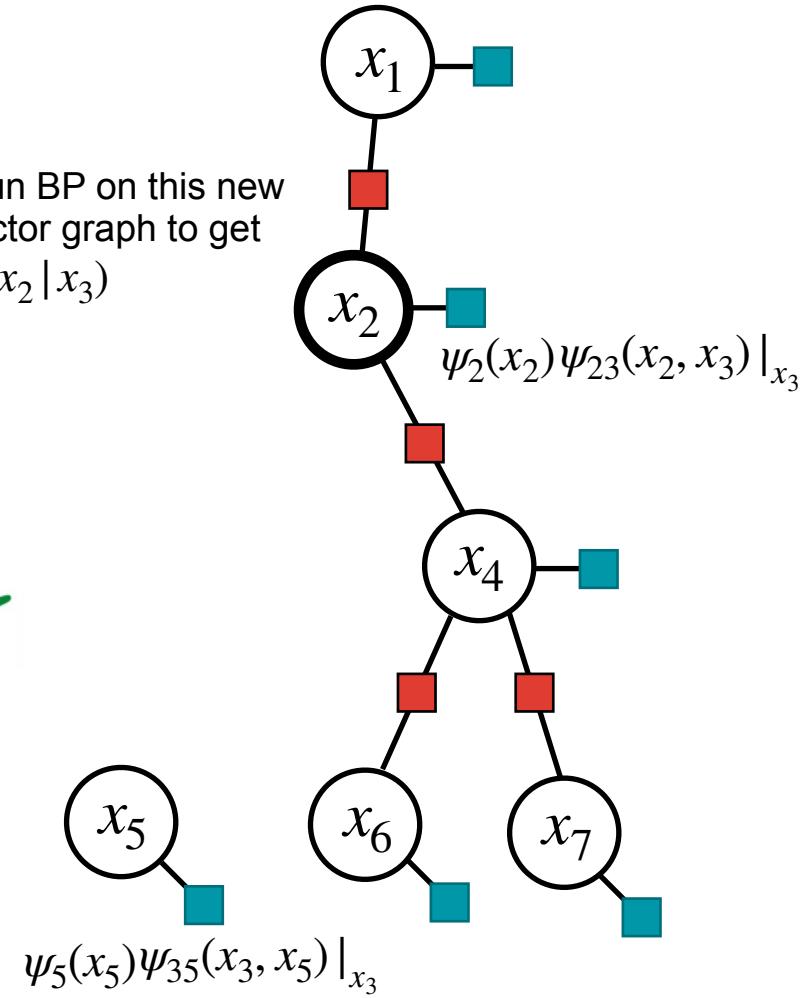


# Checklist

If  $G = (V, E)$  is a tree, can we 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  $i, j \in V$  ✓

Run BP on this new factor graph to get  
 $p(x_2 | x_3)$



# Checklist

If  $G = (V, E)$  is a tree, can we 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  $x^* = \operatorname{argmax}_x p(x)$

# Checklist

If  $G = (V, E)$  is a tree, can we 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  $x^* = \text{argmax}_x p(x)$  ✗

### 3. Some Extensions of Belief Propagation

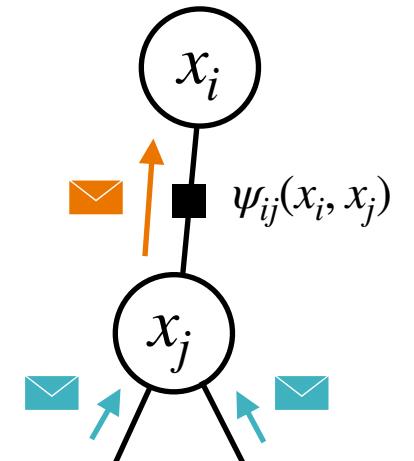
Recall the message passing protocol in BP:

**Message update:**

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

**State update:**

$$p(x_i) \propto \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i).$$



We assume that the graph is **tree-structured**.

What extensions can we consider?

# Extension 1. Continuous states

When states are *continuous*  $x_i \in \mathbb{R}^d$ , we replace the sum by an integral:

**Message update:**

$$M_{j \rightarrow i}(x_i) = \int_{\mathbb{R}^d} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j) dx_j,$$

**State update:**

$$p(x_i) \propto \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i).$$

The integral is generally intractable, except in some cases.

For e.g. **Gaussian belief propagation**.

# Gaussian belief propagation

## Properties of Gaussians:

1. Product of two Gaussians is Gaussian:

$$\mathcal{N}(x | a, A) \mathcal{N}(x | b, B) = \mathcal{N}(x | c, C),$$

where  $c = C(A^{-1}a + B^{-1}b)$ ,  $C = (A^{-1} + B^{-1})^{-1}$ .

## Message update:

$$M_{j \rightarrow i}(x_i) = \int_{\mathbb{R}^d} \psi_{ij}(x_i, x_j) \boxed{\psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j)} dx_j,$$

## State update:

$$p(x_i) \propto \boxed{\psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i)}.$$

# Gaussian belief propagation

## Properties of Gaussians:

1. Product of two Gaussians is Gaussian:

$$\mathcal{N}(x | a, A) \mathcal{N}(x | b, B) = \mathcal{N}(x | c, C),$$

where  $c = C(A^{-1}a + B^{-1}b)$ ,  $C = (A^{-1} + B^{-1})^{-1}$ .

## Message update:

$$M_{j \rightarrow i}(x_i) = \int_{\mathbb{R}^d} \psi_{ij}(x_i, x_j) \mathcal{N}(x_j | a, A) dx_j,$$

## State update:

$$p(x_i) = \mathcal{N}(x_i | \mu_i, \Sigma_i).$$

# Gaussian belief propagation

**Properties of Gaussians:**

2. Integral of Gaussians is Gaussian:

i.)  $\int_{\mathbb{R}^d} \mathcal{N}(x | Hx', R) \mathcal{N}(x' | a, A) dx' = \mathcal{N}(x | Ha, HAH^T + R),$

ii.)  $\int_{\mathbb{R}^d} \mathcal{N}(x | Hx', R) \mathcal{N}(x | a, A) dx = \mathcal{N}(Hx' | a, A + R).$

**Message update:**

$$M_{j \rightarrow i}(x_i) = \boxed{\int_{\mathbb{R}^d} \psi_{ij}(x_i, x_j) \mathcal{N}(x_j | a, A) dx_j},$$

**State update:**

$$p(x_i) = \mathcal{N}(x_i | \mu_i, \Sigma_i).$$

# Gaussian belief propagation

**Properties of Gaussians:**

2. Integral of Gaussians is Gaussian:

$$\begin{aligned} \text{i.) } & \int_{\mathbb{R}^d} \mathcal{N}(x | Hx', R) \mathcal{N}(x' | a, A) dx' = \mathcal{N}(x | Ha, HAH^T + R), \\ \text{ii.) } & \int_{\mathbb{R}^d} \mathcal{N}(x | Hx', R) \mathcal{N}(x | a, A) dx = \mathcal{N}(Hx' | a, A + R). \end{aligned}$$

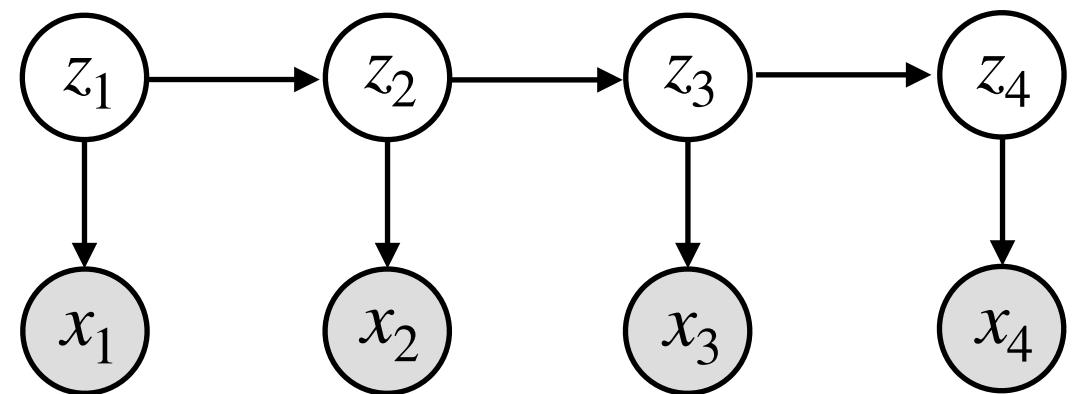
**Message update:**

$$M_{j \rightarrow i}(x_i) = \mathcal{N}(x_i | \mu_{j \rightarrow i}, \Sigma_{j \rightarrow i}),$$

**State update:**

$$p(x_i) = \mathcal{N}(x_i | \mu_i, \Sigma_i).$$

# Example: Timeseries modelling



Bayesian network representation of a  
state-space model

## Example: Timeseries modelling

Consider a linear state-space model:

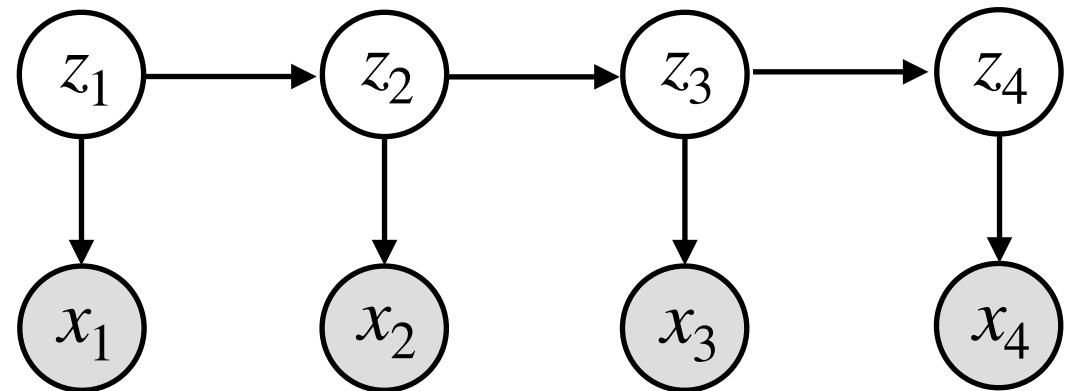
$$z_{n+1} = Mz_n + \epsilon_n, \quad \epsilon_n \sim \mathcal{N}(0, Q),$$

$$x_n = Hz_n + \eta_n, \quad \eta_n \sim \mathcal{N}(0, R).$$

Equivalently,

$$p(z_{n+1} | z_n) = \mathcal{N}(z_{n+1} | Mz_n, Q),$$

$$p(x_n | z_n) = \mathcal{N}(x_n | Hz_n, R).$$



Bayesian network representation of a state-space model

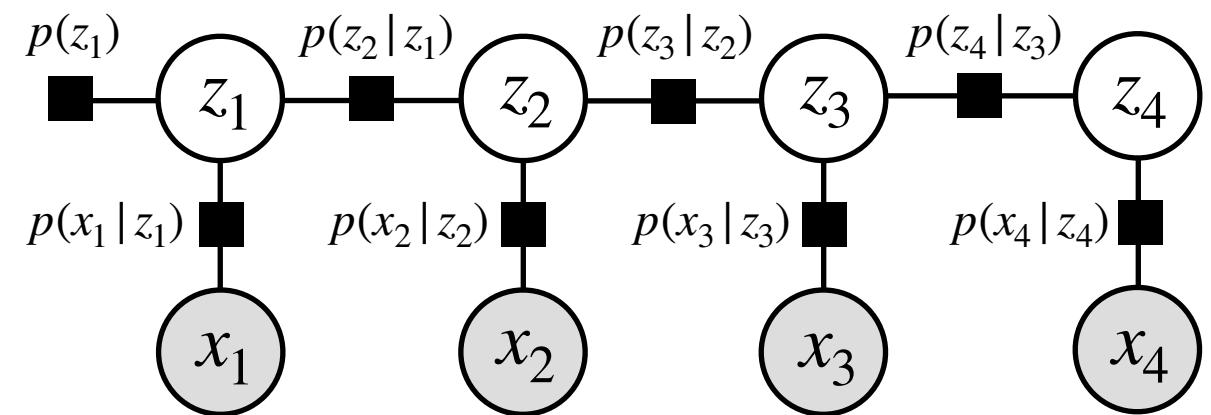
## Example: Timeseries modelling

Consider a linear state-space model:

$$\begin{aligned} z_{n+1} &= Mz_n + \epsilon_n, & \epsilon_n &\sim \mathcal{N}(0, Q), \\ x_n &= Hz_n + \eta_n, & \eta_n &\sim \mathcal{N}(0, R). \end{aligned}$$

Equivalently,

$$\begin{aligned} p(z_{n+1} | z_n) &= \mathcal{N}(z_{n+1} | Mz_n, Q), \\ p(x_n | z_n) &= \mathcal{N}(x_n | Hz_n, R). \end{aligned}$$



Factor graph representation of a state-space model

## Example: Timeseries modelling

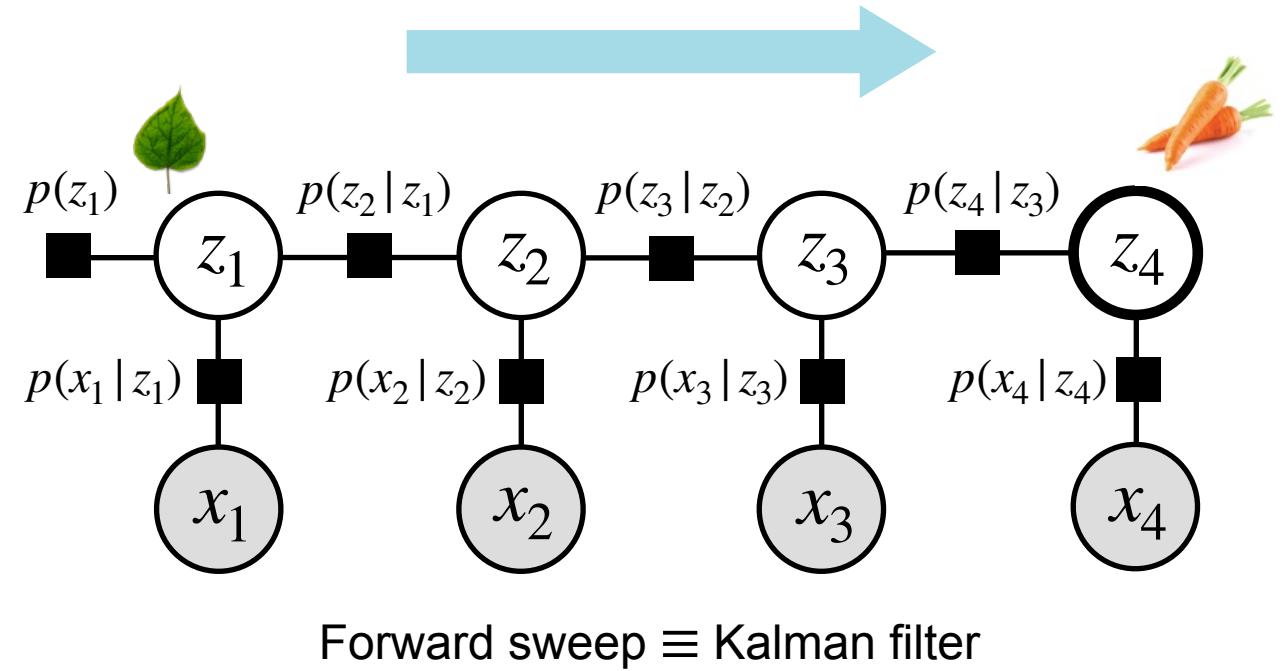
Consider a linear state-space model:

$$\begin{aligned} z_{n+1} &= Mz_n + \epsilon_n, & \epsilon_n &\sim \mathcal{N}(0, Q), \\ x_n &= Hz_n + \eta_n, & \eta_n &\sim \mathcal{N}(0, R). \end{aligned}$$

Equivalently,

$$p(z_{n+1} | z_n) = \mathcal{N}(z_{n+1} | Mz_n, Q),$$

$$p(x_n | z_n) = \mathcal{N}(x_n | Hz_n, R).$$



- Running only the forward sweep of BP is equivalent to the *Kalman filter*

## Example: Timeseries modelling

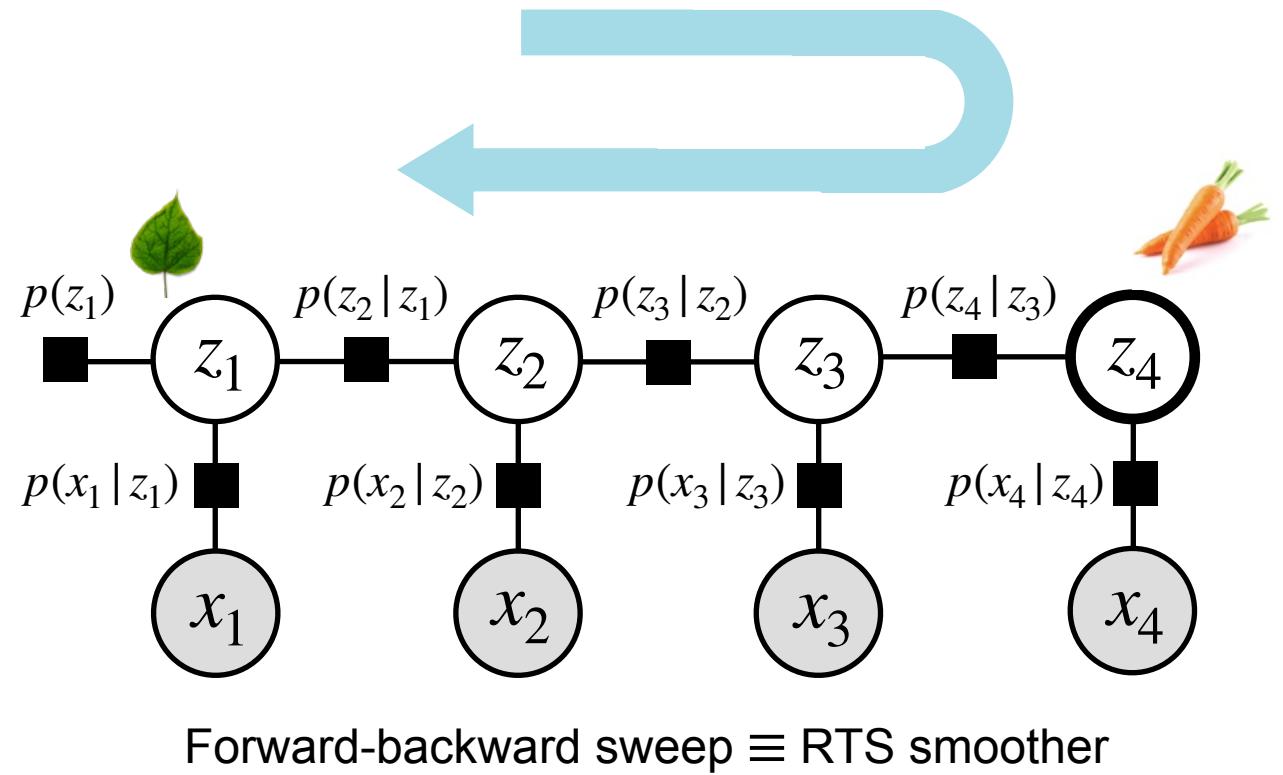
Consider a linear state-space model:

$$\begin{aligned} z_{n+1} &= Mz_n + \epsilon_n, & \epsilon_n &\sim \mathcal{N}(0, Q), \\ x_n &= Hz_n + \eta_n, & \eta_n &\sim \mathcal{N}(0, R). \end{aligned}$$

Equivalently,

$$p(z_{n+1} | z_n) = \mathcal{N}(z_{n+1} | Mz_n, Q),$$

$$p(x_n | z_n) = \mathcal{N}(x_n | Hz_n, R).$$



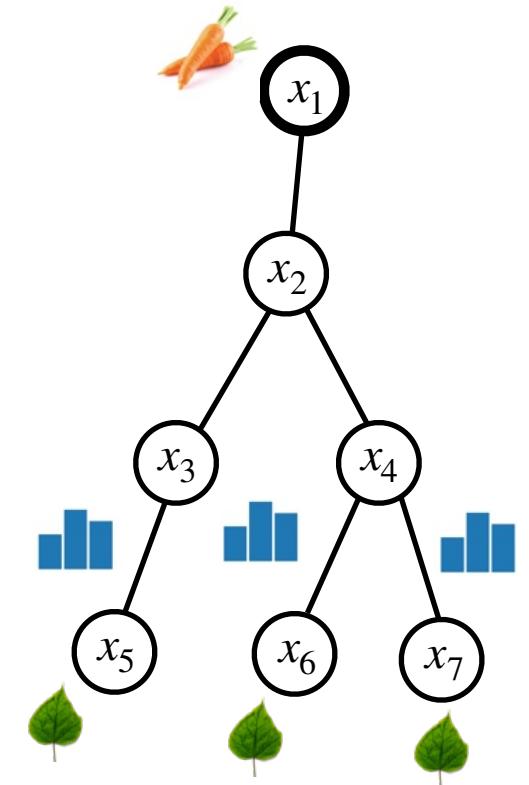
- Running only the forward sweep of BP is equivalent to the *Kalman filter*
- Running a full BP is equivalent to the *Rauch-Tung Striebel smoother*

# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

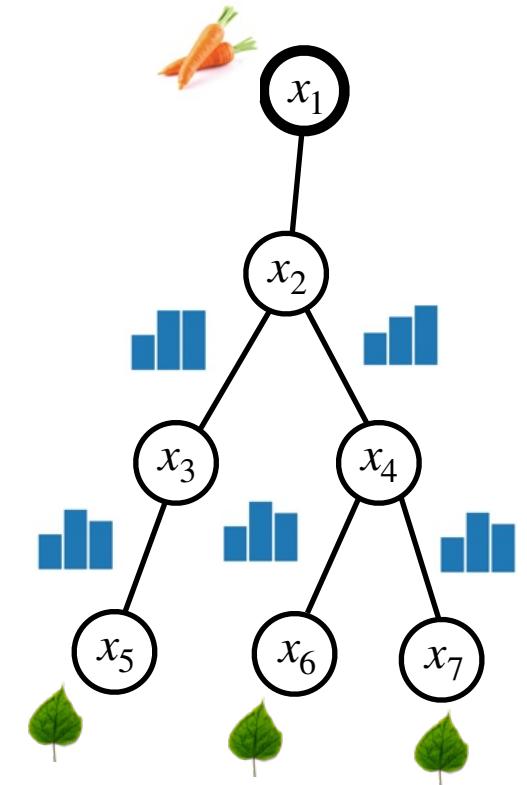


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

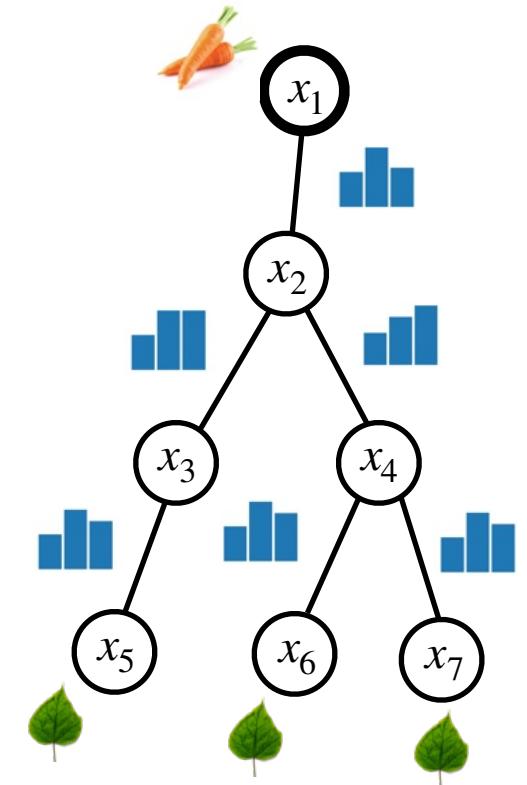


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

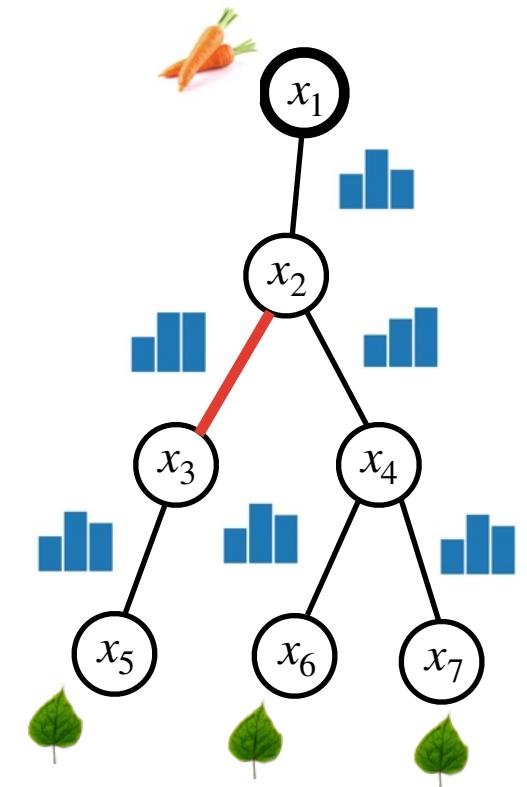


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

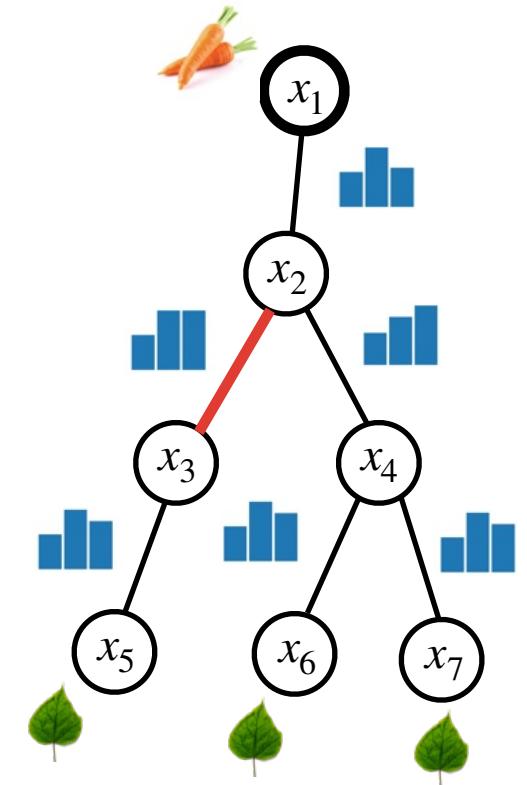


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

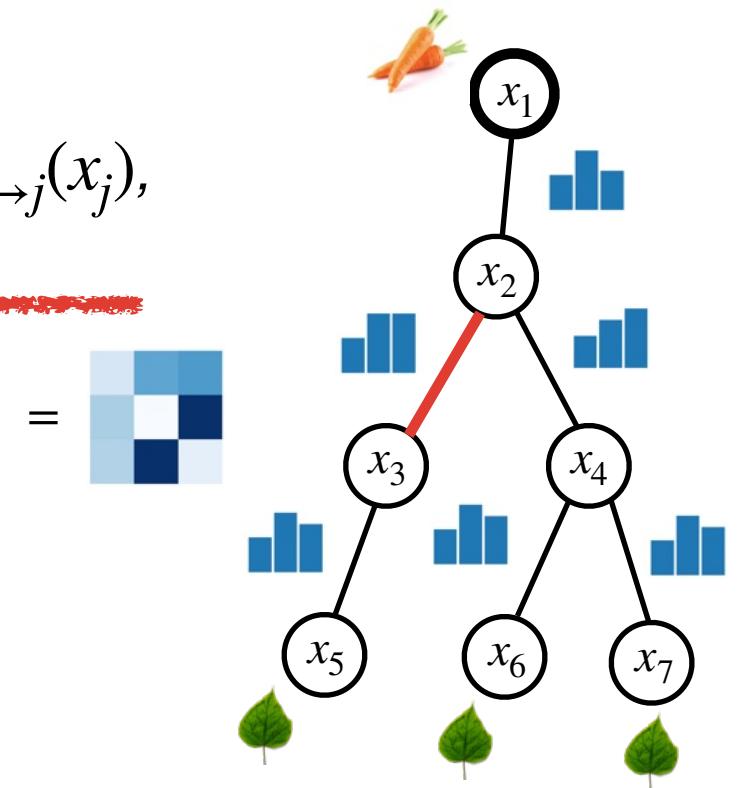


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

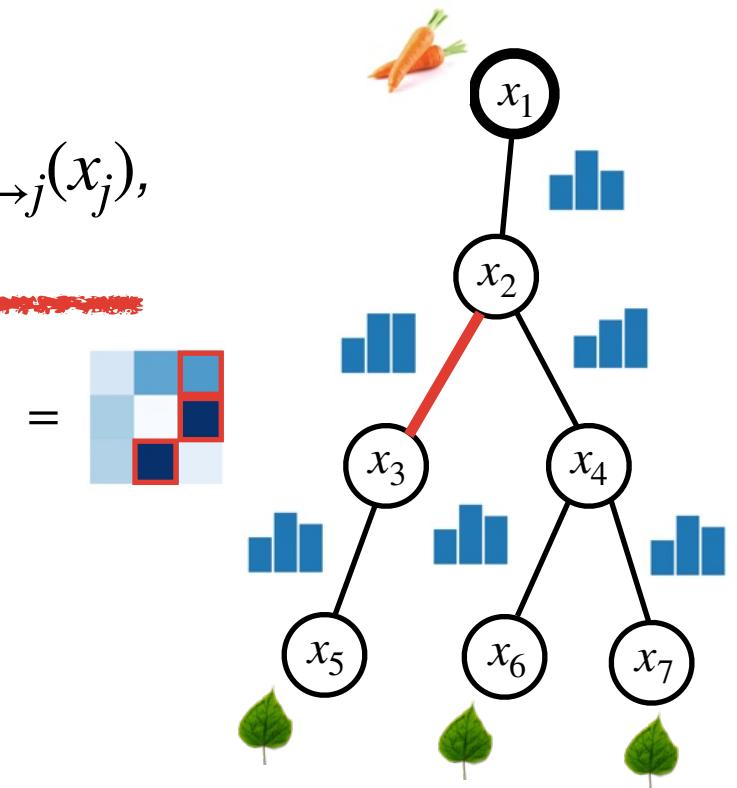


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

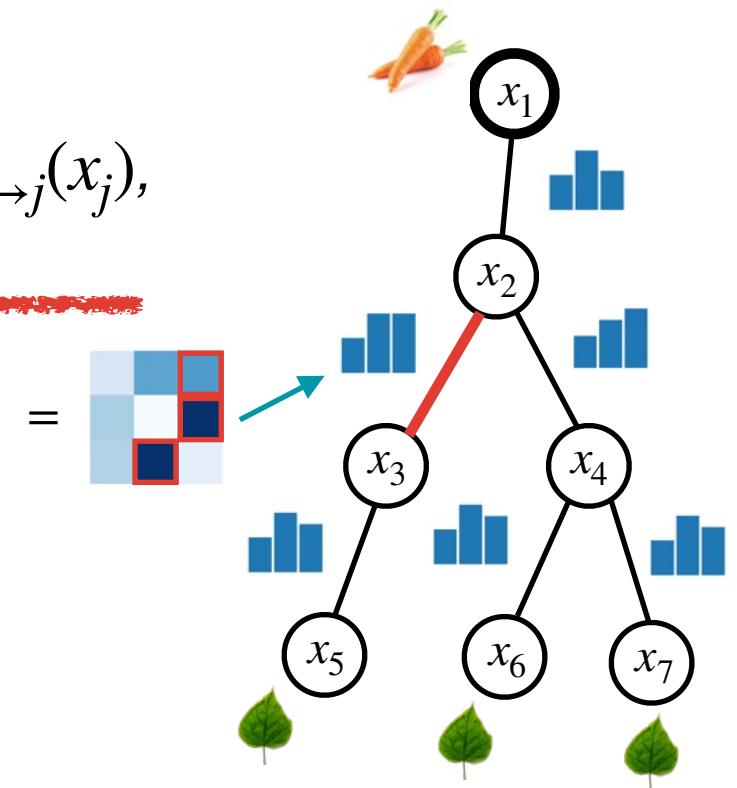


# Extension 2. Max-product algorithm

Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.



# Extension 2. Max-product algorithm

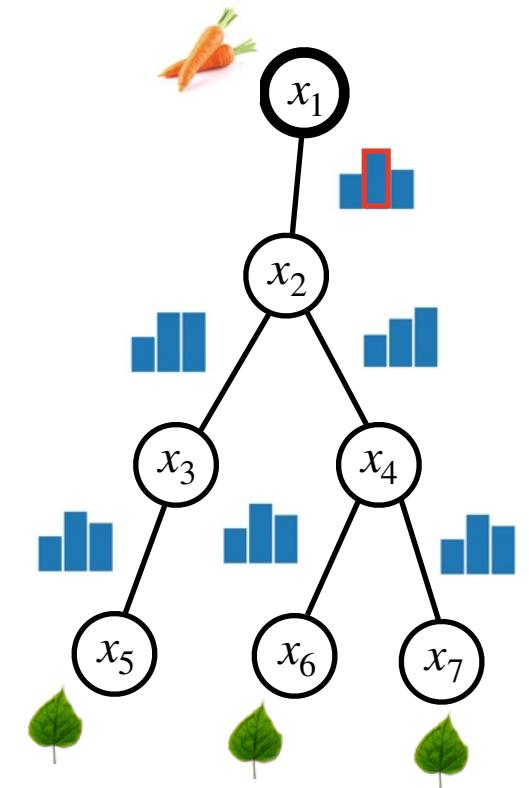
Replacing the sum in the message update by a max operator, we obtain the **max-product algorithm**:

$$M_{j \rightarrow i}(x_i) = \max_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

Iterate from leaf nodes up to the root node.

Then, we get

$$\max_x p(\mathbf{x}) = \max_{x_{\text{root}} \in \{1, \dots, K\}} \frac{1}{Z} \prod_{j \sim \text{root}} M_{j \rightarrow \text{root}}(x_{\text{root}}).$$

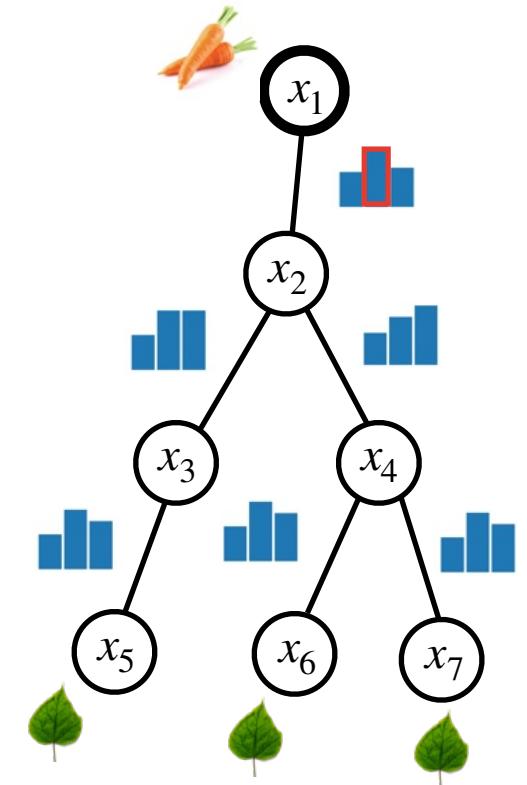


# Extension 2. Max-product algorithm

Going from the root node back to the leaf nodes, we can find the **mode**:

$$x^* = \operatorname{argmax}_x p(x),$$

using a procedure called *back-tracking*:



# Extension 2. Max-product algorithm

Going from the root node back to the leaf nodes, we can find the **mode**:

$$\boldsymbol{x}^* = \operatorname{argmax}_{\boldsymbol{x}} p(\boldsymbol{x}),$$

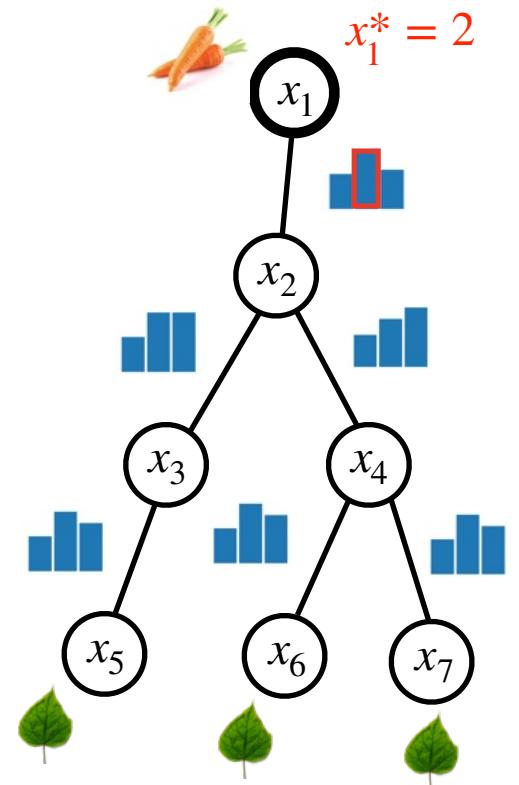
using a procedure called *back-tracking*:

1. At the root node, compute

$$x_{\text{root}}^* = \operatorname{argmax}_{x_{\text{root}}} \frac{1}{Z} \prod_{j \sim \text{root}} M_{j \rightarrow \text{root}}(x_{\text{root}}).$$

2. From the root node back to the leaf nodes, compute

$$x_j^* = \operatorname{argmax}_{x_j} \psi_{ij}(x_i^*, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j), \quad j \sim i.$$



# Extension 2. Max-product algorithm

Going from the root node back to the leaf nodes, we can find the **mode**:

$$\boldsymbol{x}^* = \operatorname{argmax}_{\boldsymbol{x}} p(\boldsymbol{x}),$$

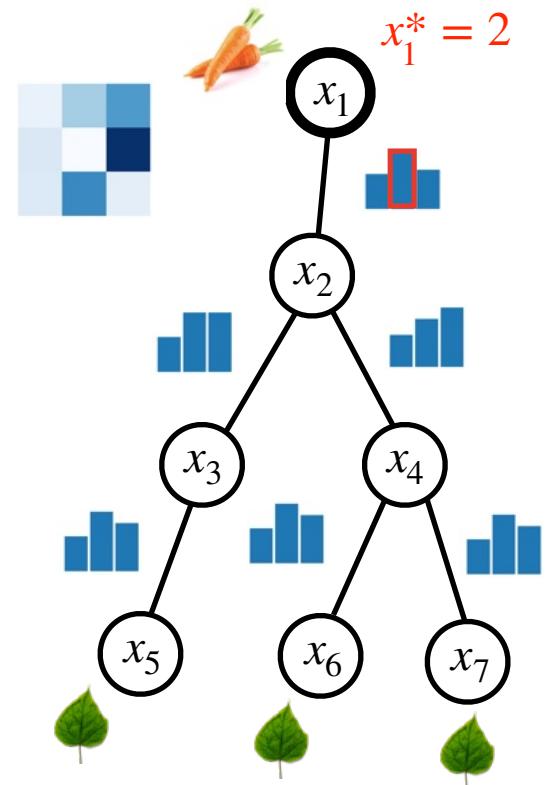
using a procedure called *back-tracking*:

1. At the root node, compute

$$x_{\text{root}}^* = \operatorname{argmax}_{x_{\text{root}}} \frac{1}{Z} \prod_{j \sim \text{root}} M_{j \rightarrow \text{root}}(x_{\text{root}}).$$

2. From the root node back to the leaf nodes, compute

$$x_j^* = \operatorname{argmax}_{x_j} \psi_{ij}(x_i^*, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j), \quad j \sim i.$$



# Extension 2. Max-product algorithm

Going from the root node back to the leaf nodes, we can find the **mode**:

$$\boldsymbol{x}^* = \operatorname{argmax}_{\boldsymbol{x}} p(\boldsymbol{x}),$$

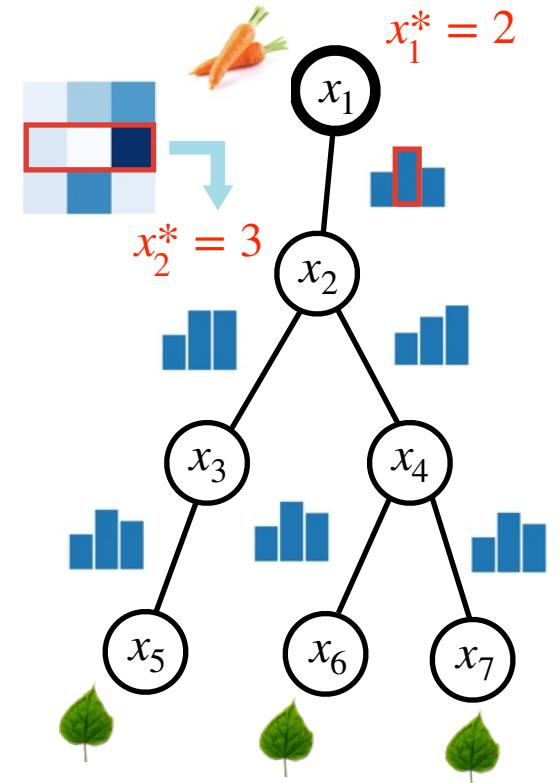
using a procedure called *back-tracking*:

1. At the root node, compute

$$x_{\text{root}}^* = \operatorname{argmax}_{x_{\text{root}}} \frac{1}{Z} \prod_{j \sim \text{root}} M_{j \rightarrow \text{root}}(x_{\text{root}}).$$

2. From the root node back to the leaf nodes, compute

$$x_j^* = \operatorname{argmax}_{x_j} \psi_{ij}(x_i^*, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j), \quad j \sim i.$$



# Extension 2. Max-product algorithm

Going from the root node back to the leaf nodes, we can find the **mode**:

$$\mathbf{x}^* = \operatorname{argmax}_{\mathbf{x}} p(\mathbf{x}),$$

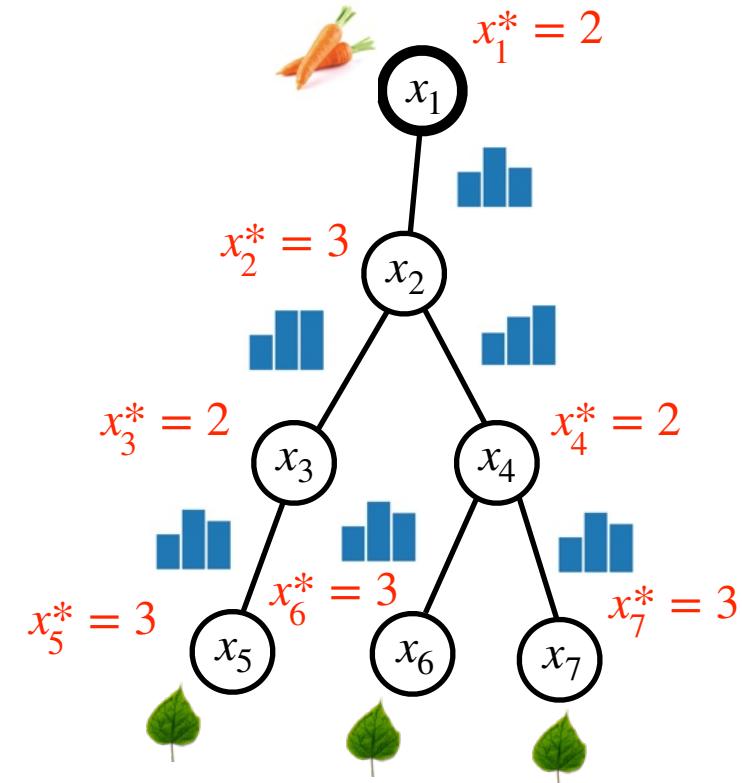
using a procedure called *back-tracking*:

1. At the root node, compute

$$x_{root}^* = \operatorname{argmax}_{x_{root}} \frac{1}{Z} \prod_{j \sim \text{root}} M_{j \rightarrow \text{root}}(x_{root}).$$

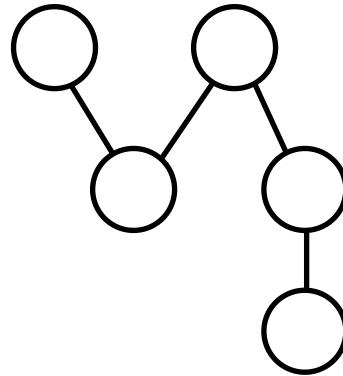
2. From the root node back to the leaf nodes, compute

$$x_j^* = \operatorname{argmax}_{x_j} \psi_{ij}(x_i^*, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j), \quad j \sim i.$$

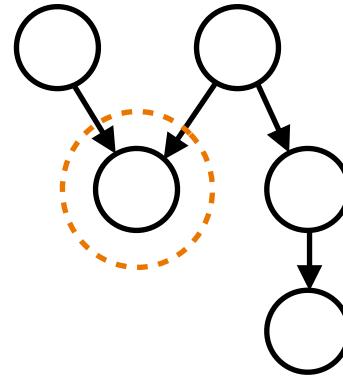


# Extension 3. Polytrees and other graphs

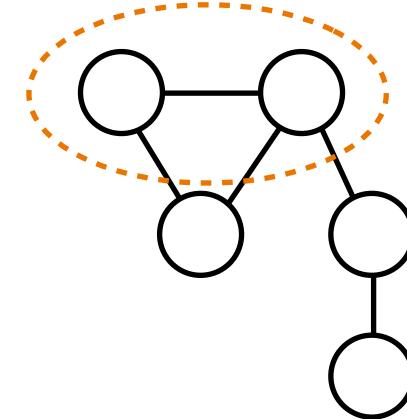
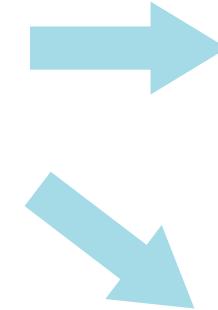
A **polytree** is a directed tree



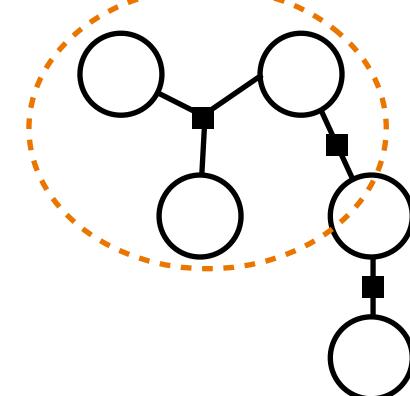
A tree



A polytree



A polytree as a MRF



A polytree as a factor graph

**Note:** factors are not necessarily pairwise!

# Extension 3. Polytrees and other graphs

On trees, the message passing updates read:

**Message update:**

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

**State update:**

$$p(x_i) \propto \psi_i(x_i) \prod_{j \sim i} M_{j \rightarrow i}(x_i).$$

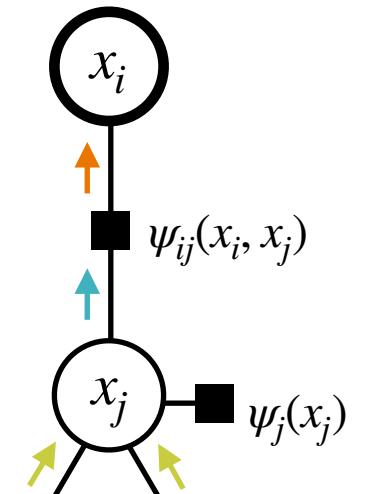
# Extension 3. Polytrees and other graphs

First, break down the message update step into two sub-steps:

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

$$1. \mu_{x_j \rightarrow \psi_{ij}}(x_j) = \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j). \quad (\text{variable-to-factor message})$$

$$2. \underbrace{\mu_{\psi_{ij} \rightarrow x_i}(x_i)}_{\equiv M_{j \rightarrow i}(x_i)} = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \mu_{x_j \rightarrow \psi_{ij}}(x_j). \quad (\text{factor-to-variable message})$$

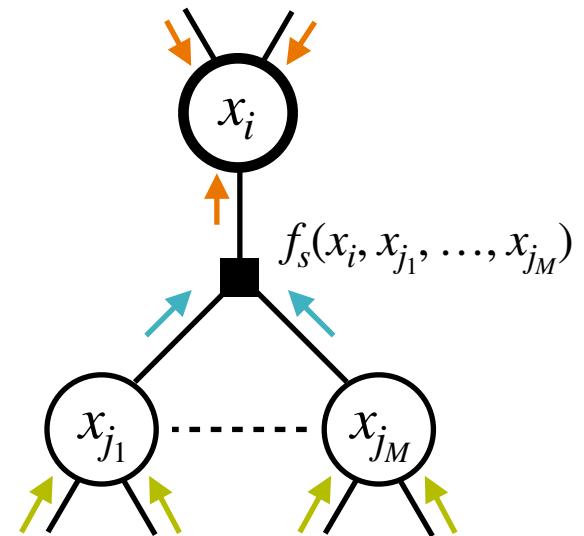


# Extension 3. Polytrees and other graphs

Extending to polytrees:

$$1. \mu_{x_j \rightarrow f_s}(x_j) = \prod_{l \in \text{ne}(x_j) \setminus s} \mu_{f_l \rightarrow x_j}(x_j)$$

$$2. \mu_{f_s \rightarrow x_i}(x_i) = \sum_{x_{j_1}, \dots, x_{j_M}} f_s(x_i, x_{j_1}, \dots, x_{j_M}) \prod_{k=1}^M \mu_{x_{j_k} \rightarrow f_s}(x_{j_k})$$



The state updates read:

$$p(x_i) = \prod_{s \in \text{ne}(x_i)} \mu_{f_s \rightarrow x_i}(x_i).$$

# Extension 3. Polytrees and other graphs

We can apply the same update rules to more general graphs with loops.

This is called **Loopy Belief Propagation (LBP)**.

**Message update (same as before):**

$$1. \mu_{x_j \rightarrow f_s}(x_j) = \prod_{l \in \text{ne}(x_j) \setminus s} \mu_{f_l \rightarrow x_j}(x_j)$$

$$2. \mu_{f_s \rightarrow x_i}(x_i) = \sum_{x_{j_1}, \dots, x_{j_M}} f_s(x_i, x_{j_1}, \dots, x_{j_M}) \prod_{k=1}^M \mu_{x_{j_k} \rightarrow f_s}(x_{j_k})$$

**State update (same as before):**

$$p(x_i) = \prod_{s \in \text{ne}(x_i)} \mu_{f_s \rightarrow x_i}(x_i).$$

- LBP is iterative and can be started off by setting

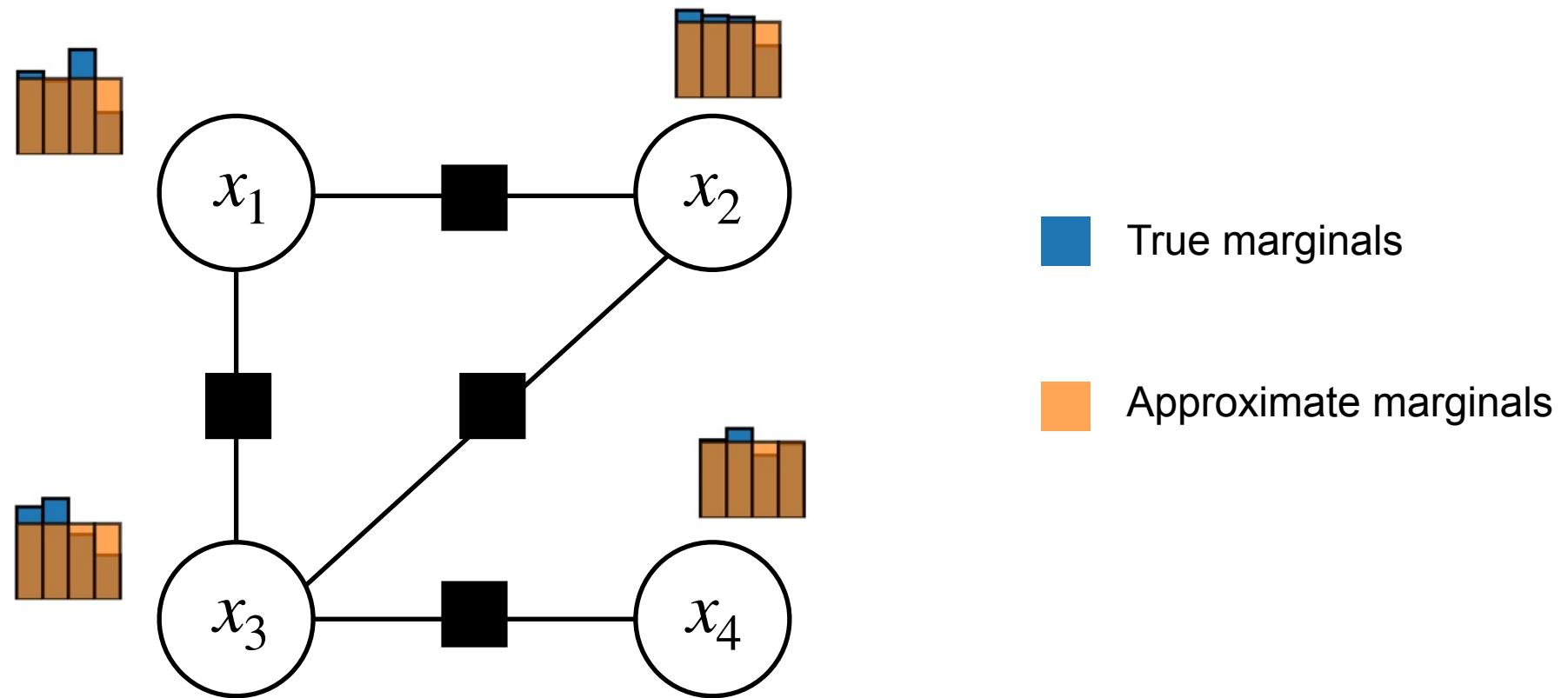
$$\mu_{x \rightarrow f}(x) = 1,$$

for all variables  $x$  and factors  $f$ .

- Updates can be done in parallel (flooding schedule).

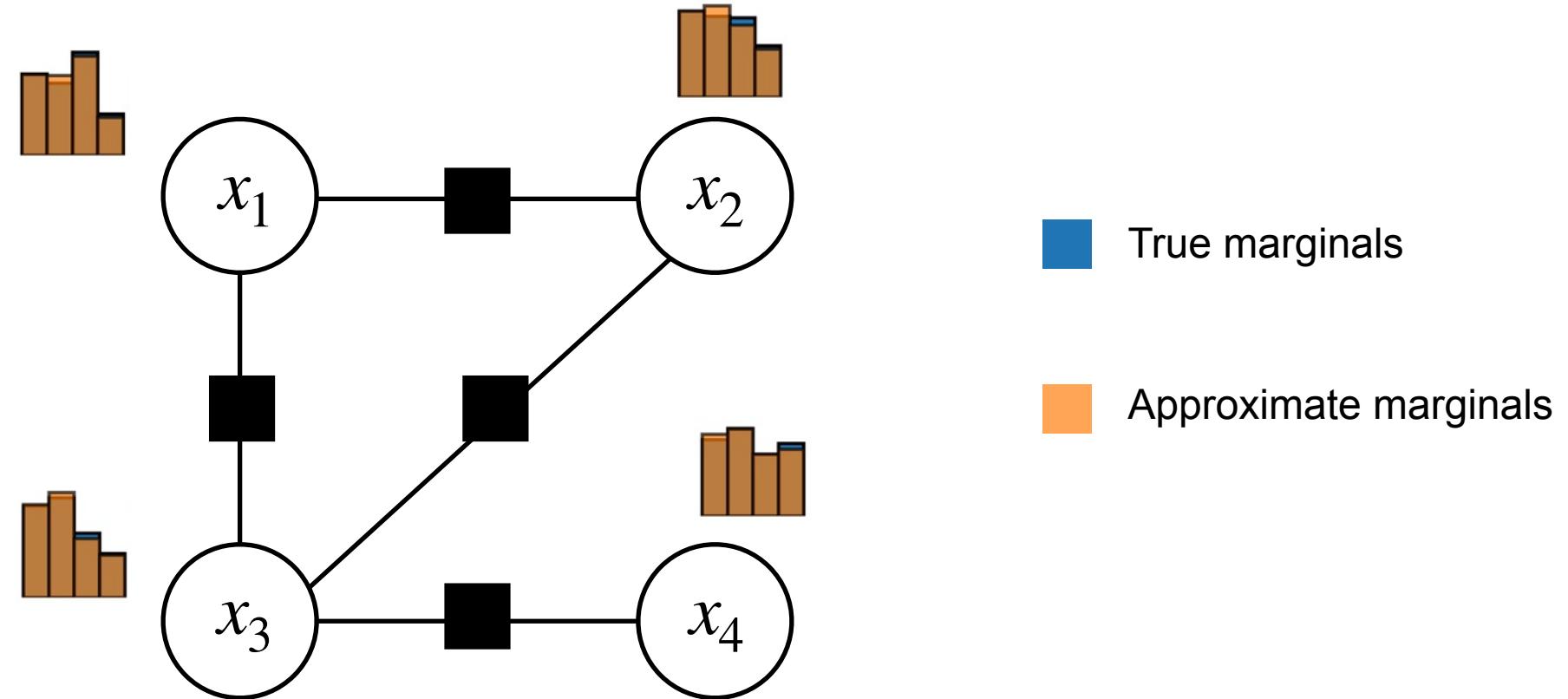
# Implementation (flooding schedule)

Iteration 1



# Implementation (flooding schedule)

Iteration 1



# References

- [1] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. New York: springer, 2006.
- [2] Wainwright, Martin J., and Michael I. Jordan. *Graphical Models, Exponential Families, and Variational Inference*. Foundations and Trends in Machine Learning, 2008.
- [3] Ortiz, Joseph, Talfan Evans, and Andrew J. Davison. *A Visual Introduction to Gaussian Belief Propagation*. 2021. (<https://gaussianbp.github.io/>)
- [4] Minka, Thomas P. *Expectation propagation for approximate Bayesian inference*. Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence, 2001.
- [5] Yedidia, Jonathan S., William T. Freeman, and Yair Weiss. *Understanding belief propagation and its generalizations*. Exploring artificial intelligence in the new millennium, 2003.

# 4. Message Passing Neural Networks

# Neural networks

Neural networks have dominated ML in the past decade.

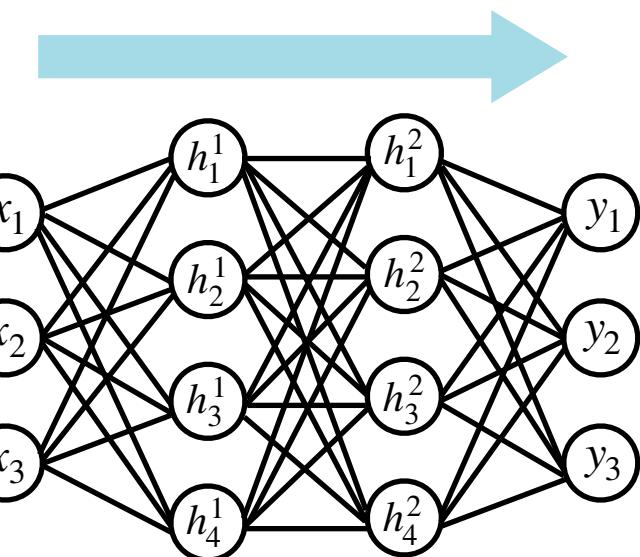
They are:

- Extremely flexible for modelling
- Able to process complex data structures
- Composed of simple, parallelisable components
- Automatically differentiable

$$h^0 = x$$

$$h^{l+1} = \text{ReLU}(W h^l + b), \quad t = 0, \dots, L - 1$$

$$y = \text{Softmax}(W h^L + b)$$



Multilayer perceptron

# A zoo of graphs in the real world

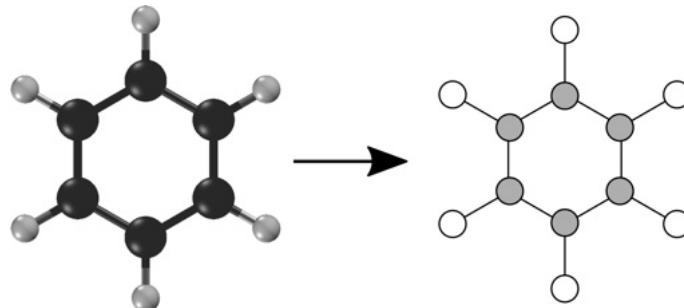


Image from: <https://www.oreilly.com/library/view/deep-learning-for/9781492039822/ch04.html>



Image from: <https://medium.com/analytics-vidhya/social-network-analytics-f082f4e21b16>

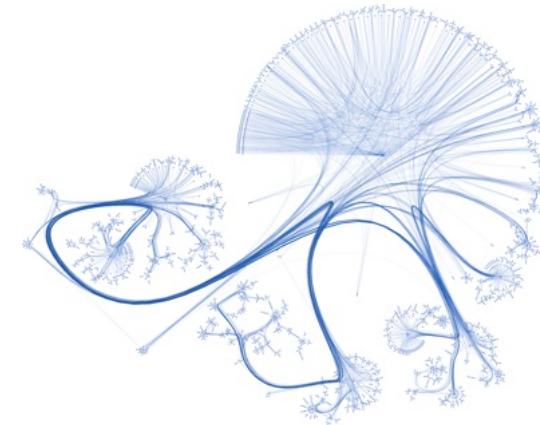


Image from: <https://graphsandnetworks.com/the-cora-dataset/>

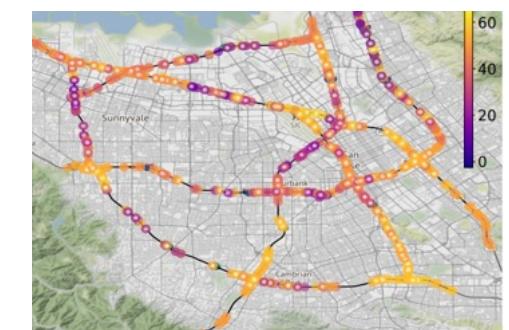
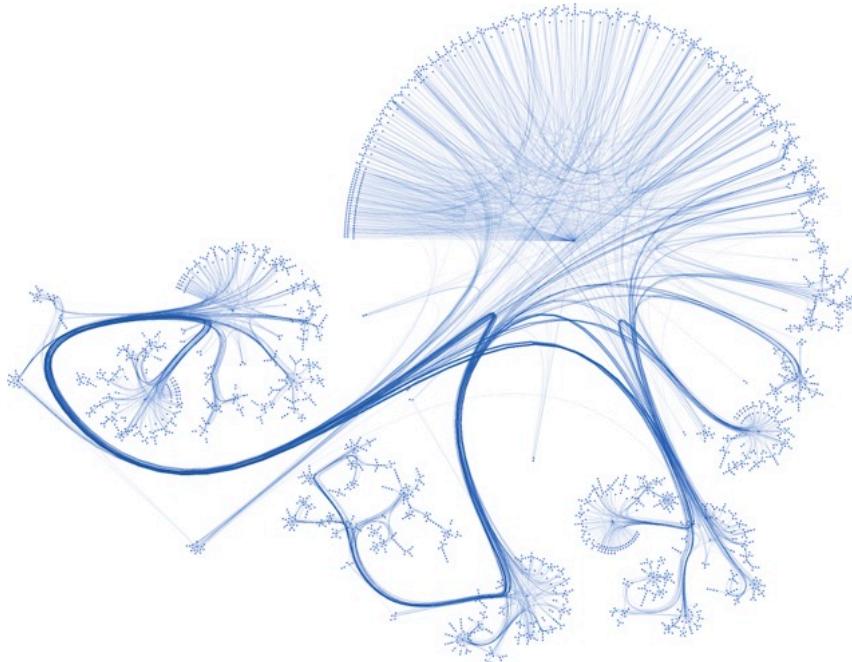


Image from: <http://proceedings.mlr.press/v130/borovitskiy21a/borovitskiy21a.pdf>

# Example: Cora dataset

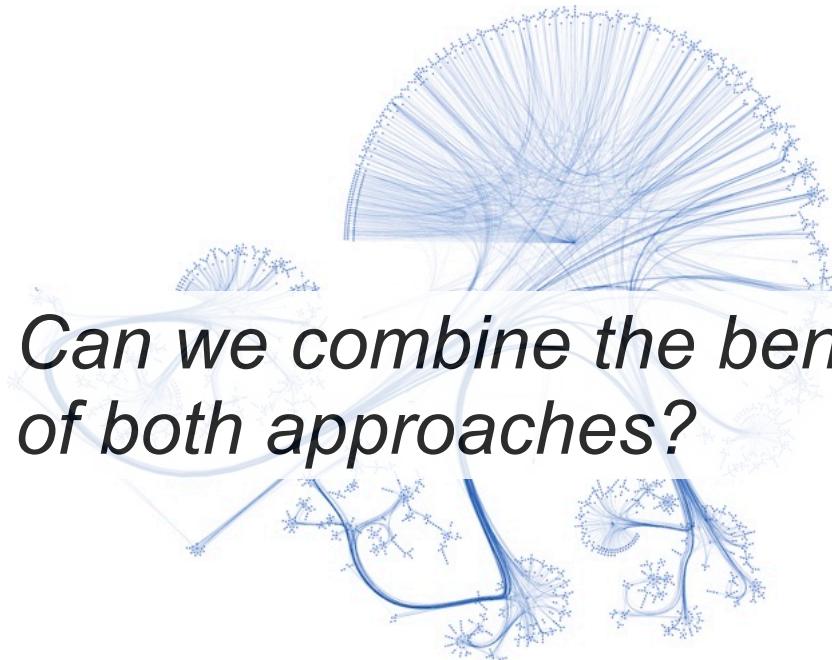


Overview of dataset:

- 2708 ML publications
- 5429 citation links
- Node feature size: 1433
- Seven classes

**Task:** classify nodes according to topic

# Example: Cora dataset



## Using belief propagation:

- Create a MRF with pairwise potential [12]

$$\psi_{ij}(x_i, x_j) = \begin{cases} 0.9, & x_i = x_j \\ 0.0166..., & x_i \neq x_j \end{cases}$$

- Perform LBP to compute  $p(x_i | x^{obs})$
- However,
- This does not consider node features
  - Pairwise potential is arbitrary

# Convolutional neural networks

- Incorporates inductive bias of grid-inputs
- Sparse connectivity owing to local receptive field
- Shared parameters

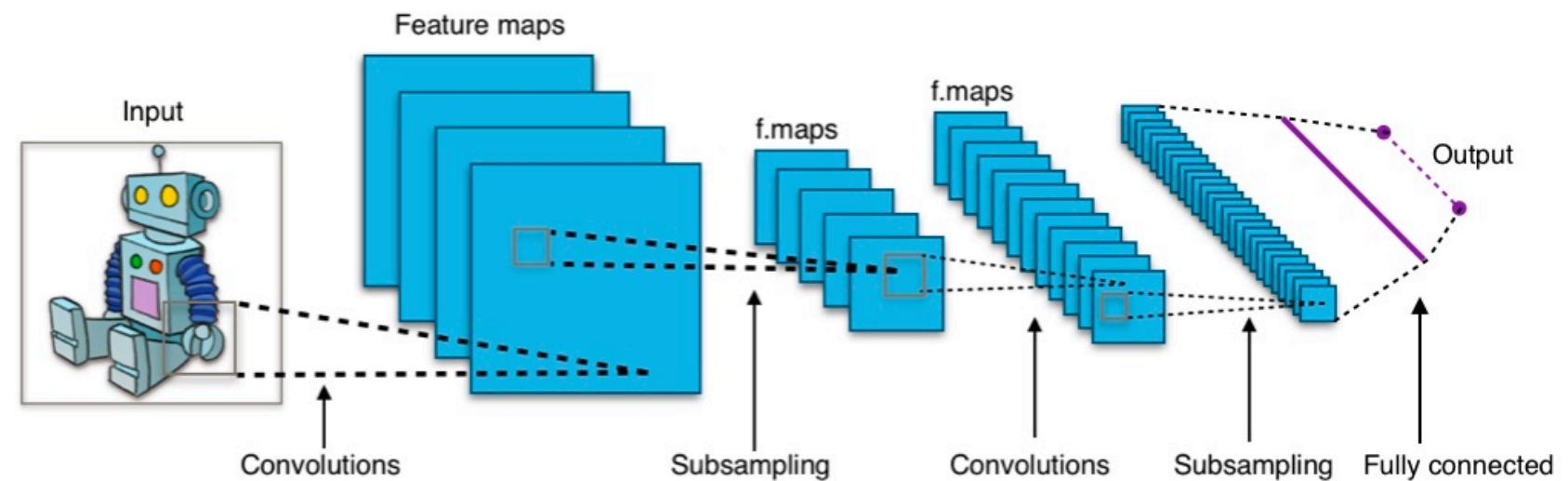


Image from: [https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network](https://en.wikipedia.org/wiki/Convolutional_neural_network)

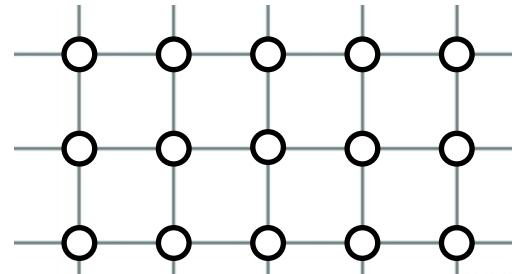
# Criteria for an “ideal” graph NN

1.  $\mathcal{O}(|V| + |E|)$  computational and storage efficiency
2. Parameter size independent of input size
3. Use local information to construct hidden features
4. Can use edge features in addition to node features

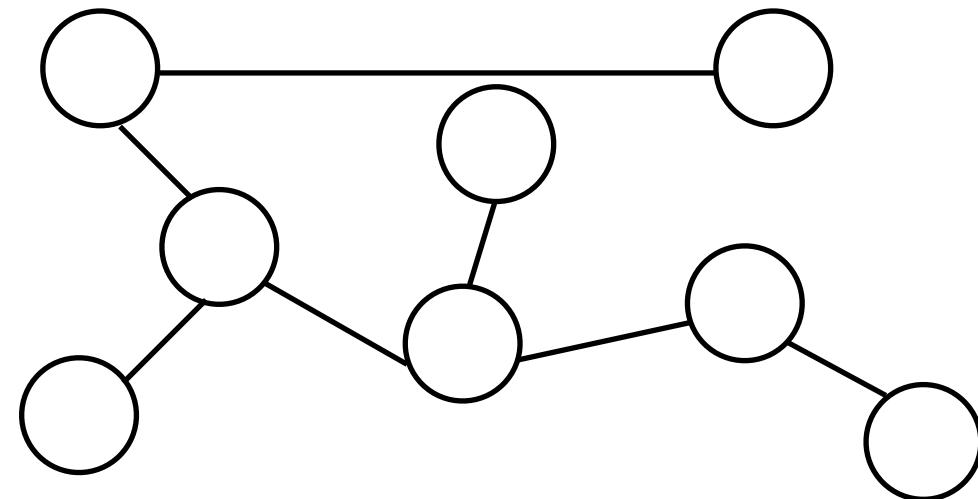
# Extending convolutions to graphs?

CNNs are based on discretisation of the *convolution operator*

$$\begin{aligned} f \star \psi_\theta(x) &= \int_{\mathbb{R}^2} f(y) \psi_\theta(x - y) dy \\ &\approx \sum_{y \in \mathbb{Z}^2} f(y) \psi_\theta(x - y) \end{aligned}$$



Convolution applies to grids



Can we define convolutions on graphs?

# Spectral graph convolution

Bruna et al. [1] introduced SpectralNet based on the following property of  $\star$

$$f \star \psi_\theta(x) = \mathcal{F}^{-1} (\mathcal{F}f \odot \widetilde{\mathcal{F}\psi_\theta})(x),$$

where  $\mathcal{F}$  denotes the Fourier transform.  $\widetilde{\psi_\theta} = \theta$

**Observation:** Fourier transform can be defined on general graphs!

1. Construct the *graph Laplacian*  $\mathbf{L} = \mathbf{D} - \mathbf{A}$
2. Diagonalise  $\mathbf{L}$  to get  $\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^\top$
3. Define  $\mathcal{F}\mathbf{f} := \mathbf{U}^\top \mathbf{f}$  and  $\mathcal{F}^{-1}\hat{\mathbf{f}} := \mathbf{U}\hat{\mathbf{f}}$

“Spectral graph convolution”

# Spectral graph convolution

How good is SpectralNet?

1.  $\mathcal{O}(|V| + |E|)$  computational and storage efficiency ✗
  - ▶ Computational and storage cost for Fourier transform is  $\mathcal{O}(|V|^2)$
2. Parameter size independent of input size ✗
  - ▶ Parameter size is  $|V|$
3. Use local information to construct hidden features ✗
  - ▶ Diagonal features in Fourier space are non-local
4. Can use edge features in addition to node features ✗
  - ▶ Does not use edge features

# Graph Convolutional Networks

Alternatively, consider a “spatial” approach (Duvenaud et al. [3]):

$$h_{v_i}^{l+1} = \sigma \left( \sum_{j \in \mathcal{N}_i} h_{v_j}^l W_{|\mathcal{N}_i|}^l \right), \quad v_i \in V.$$

Kipf and Welling [4] introduced the **Graph Convolutional Network (GCN)**:

$$h_{v_i}^{l+1} = \text{ReLU} \left( \sum_{j \in \mathcal{N}_i} h_{v_j}^l \frac{W^l}{\sqrt{|\mathcal{N}_i| |\mathcal{N}_j|}} \right), \quad v_i \in V.$$

- Works well in practice
- Can be derived from *ChebNet* [2], a variant of spectral graph convolution

# Graph Convolutional Networks

How good is GCN?

1.  $\mathcal{O}(|V| + |E|)$  computational and storage efficiency ✓

- ▶ Computational cost is  $\mathcal{O}(|V|CF)$  (multiplication  $h_{v_j}^l W^l$  performed  $|V|$  times)
- ▶ Storage cost is  $\mathcal{O}(|E|)$  (to store adjacency matrix  $\mathbf{A}$ )

2. Parameter size independent of input size ✓

- ▶ Parameter size is  $\mathcal{O}(CF)$  per layer to store  $W^l \in \mathbb{R}^{C \times F}$

3. Use local information to construct hidden features ✓

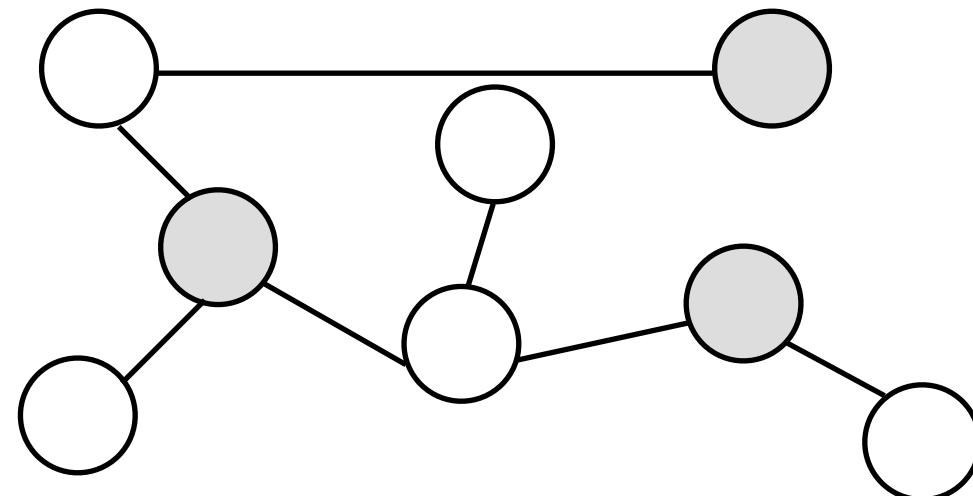
- ▶ By construction, hidden features only depend on local neighbours

4. Can use edge features in addition to node features ✗

- ▶ Does not use edge features in original formulation

# Semi-supervised learning

- Applies when the number of labelled datapoints are *small*
- But relations between labelled and unlabelled data exist



# Semi-supervised learning

Experiment with Cora dataset:

- Use only 140 nodes for training data
- 1000 nodes for testing

Train with cross-entropy loss over labelled data  $\mathcal{D}_L$  (i.e. training data):

$$L = - \sum_{(y, X) \in \mathcal{D}_L} y \log \text{GCN}(X).$$

Kipf and Welling [4] reports accuracy of:

- 81.5 % using GCN
- 55.1 % using MLP

# Message Passing Neural Networks

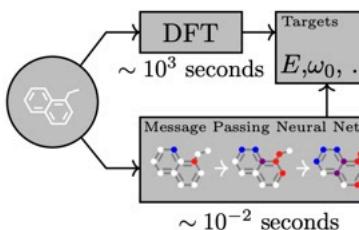
---

**Neural Message Passing for Quantum Chemistry**

---

Justin Gilmer<sup>1</sup> Samuel S. Schoenholz<sup>1</sup> Patrick F. Riley<sup>2</sup> Oriol Vinyals<sup>3</sup> George E. Dahl<sup>1</sup>

**Abstract**  
Supervised learning on molecules has incredible potential to be useful in chemistry, drug discovery, and materials science. Luckily, several promising and closely related neural network models invariant to molecular symmetries have already been described in the literature. These models learn a message passing algorithm and aggregation procedure to compute a function of their entire input graph. At this point, the next step is to find a particularly effective variant of this general approach and apply it to chemical prediction benchmarks until we either solve them or reach the limits of the approach. In this paper, we reformulate existing models into a single common framework we call Message Passing Neural Networks (MPNNs) and explore additional novel variations within this framework. Using MPNNs we demonstrate state of the art results on an important molecular property prediction benchmark; these results are strong enough that we believe future work should focus on



The diagram illustrates the computational efficiency of Message Passing Neural Networks (MPNNs) compared to Density Functional Theory (DFT). A molecule icon is connected to two parallel processing paths. The top path, labeled 'DFT', leads to 'Targets' ( $E, \omega_0, \dots$ ) and is annotated with ' $\sim 10^3$  seconds'. The bottom path, labeled 'Message Passing Neural Net', leads to the same targets and is annotated with ' $\sim 10^{-2}$  seconds'.

*Figure 1.* A Message Passing Neural Network predicts quantum properties of an organic molecule by modeling a computationally expensive DFT calculation.

Rupp et al., 2012; Rogers & Hahn, 2010; Montavon et al., 2012; Behler & Parrinello, 2007; Schoenholz et al., 2016) has revolved around feature engineering. While neural networks have been applied in a variety of situations (Merkwirth & Lengauer, 2005; Micheli, 2009; Lusci et al., 2013;

- Developed to predict properties of molecules
- Introduces a general framework for learning features on graphs based on message passing
- Can handle graph data containing both node and edge features

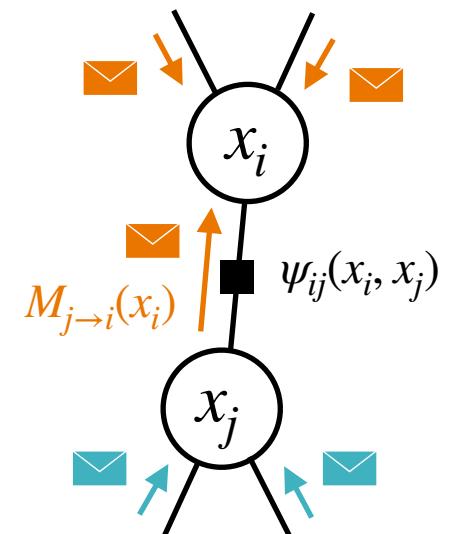
Recall the message passing protocol in BP:

**Message update:**

$$M_{j \rightarrow i}(x_i) = \sum_{x_j \in \{1, \dots, K\}} \psi_{ij}(x_i, x_j) \psi_j(x_j) \prod_{k \sim j, k \neq i} M_{k \rightarrow j}(x_j),$$

**State update:**

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



Message passing in MPNN [6]:

**Message update:**

$$M_{j \rightarrow i}^l = M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}),$$

**State update:**

$$h_{v_i}^{l+1} = U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l).$$

**Readout:**

$$y = R_\theta(\{h_{v_i}^L \mid v_i \in V\}).$$

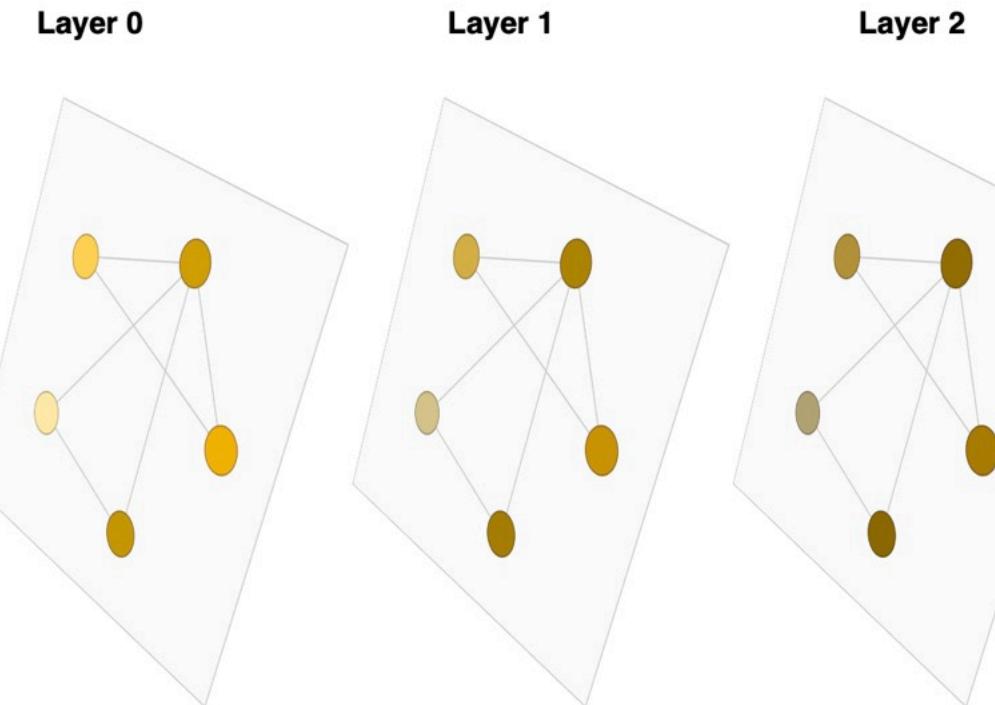


Image from: <https://distill.pub/2021/gnn-intro/>

Message passing in MPNN [6]:

**Message update:**

$$M_{j \rightarrow i}^l = M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}),$$

**State update:**

$$h_{v_i}^{l+1} = U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l).$$

**Readout:**

$$y = R_\theta(\{h_{v_i}^L \mid v_i \in V\}).$$

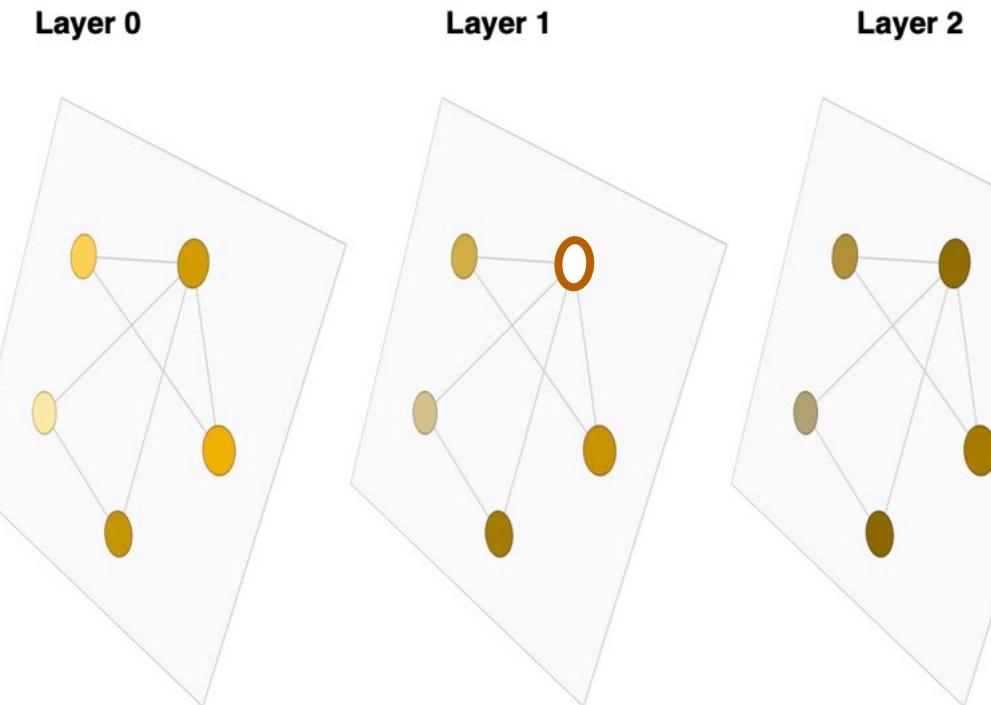


Image from: <https://distill.pub/2021/gnn-intro/>

Message passing in MPNN [6]:

**Message update:**

$$M_{j \rightarrow i}^l = M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}),$$

**State update:**

$$h_{v_i}^{l+1} = U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l).$$

**Readout:**

$$y = R_\theta(\{h_{v_i}^L \mid v_i \in V\}).$$

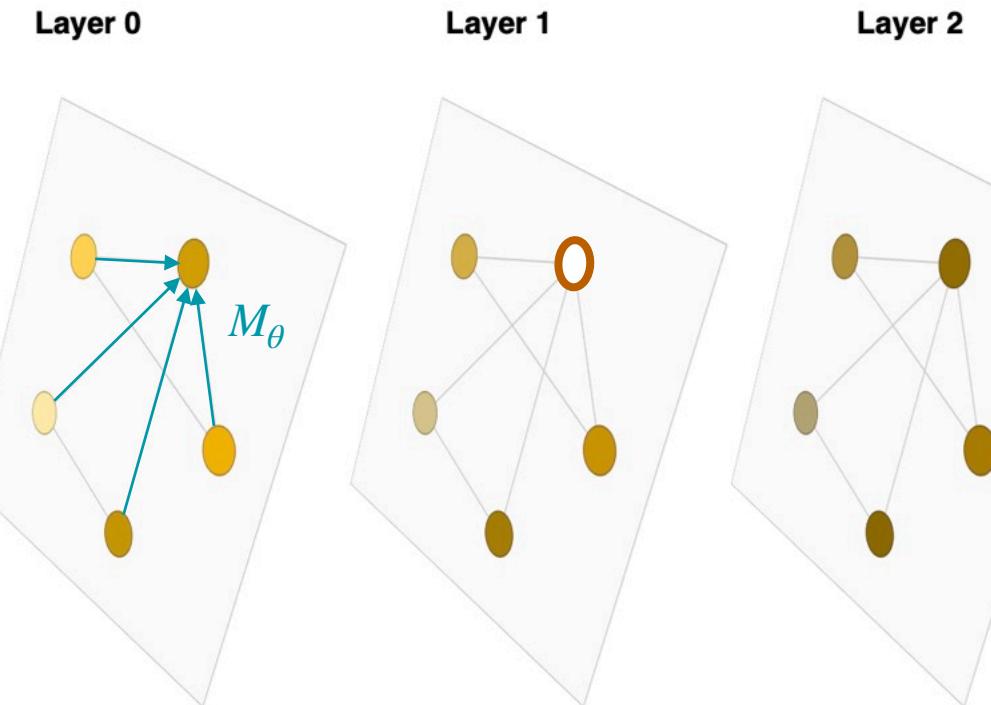


Image from: <https://distill.pub/2021/gnn-intro/>

Message passing in MPNN [6]:

**Message update:**

$$M_{j \rightarrow i}^l = M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}),$$

**State update:**

$$h_{v_i}^{l+1} = U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l).$$

**Readout:**

$$y = R_\theta(\{h_{v_i}^L \mid v_i \in V\}).$$

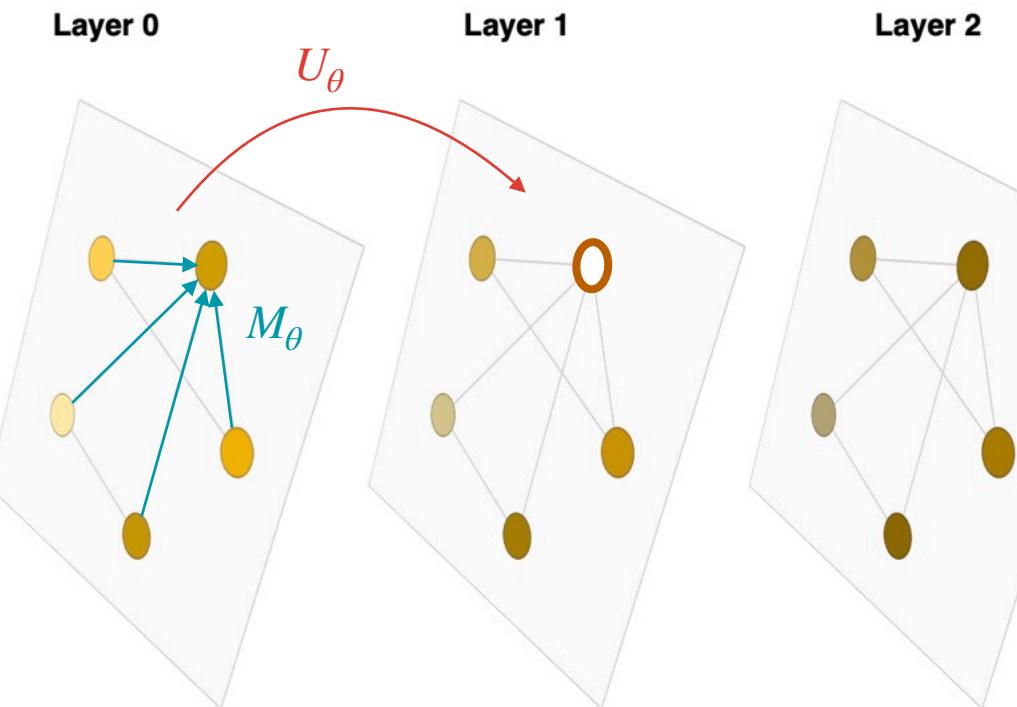


Image from: <https://distill.pub/2021/gnn-intro/>

Message passing in MPNN [6]:

**Message update:**

$$M_{j \rightarrow i}^l = M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}),$$

**State update:**

$$h_{v_i}^{l+1} = U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l).$$

**Readout:**

$$y = R_\theta(\{h_{v_i}^L \mid v_i \in V\}).$$

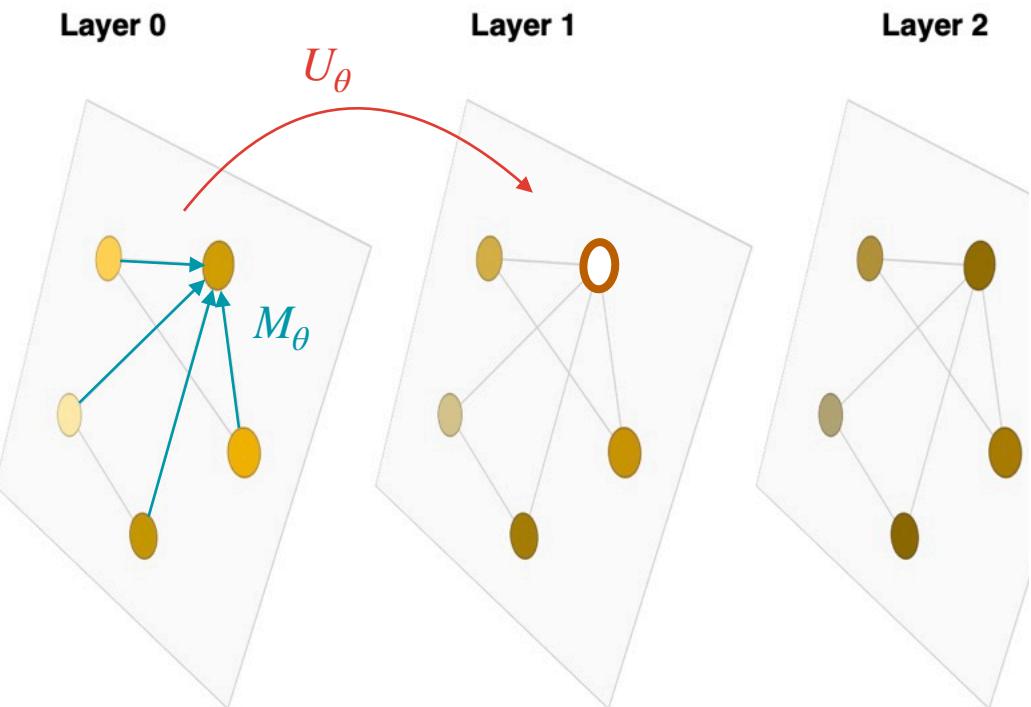


Image from: <https://distill.pub/2021/gnn-intro/>

Most GNN architectures can be expressed as an MPNN!

# Example 1: GCNs as MPNN

Recall the GCN architecture:

$$h_{v_i}^{l+1} = \text{ReLU} \left( \sum_{j \in \mathcal{N}_i} h_{v_j}^l \frac{W^l}{\sqrt{|\mathcal{N}_i| |\mathcal{N}_j|}} \right), \quad v_i \in V.$$

This can be expressed as an MPNN with:

- $M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}) = \frac{1}{\sqrt{|\mathcal{N}_i| |\mathcal{N}_j|}} h_{v_j}^l$
- $U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l) = \text{ReLU} \left( \left( \frac{1}{|\mathcal{N}_i|} h_{v_i}^l + \sum_{j \sim i} M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}) \right) W^l \right)$

# Example 2: MPNN in Gilmer et al. [5]

The original work of Gilmer et al. [5] used the following MPNN model

- $M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}) = \text{MLP}(e_{ij}) h_{v_j}^l$
- $U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l) = \text{GRU}\left(h_{v_i}^l, \sum_{j \sim i} M_{j \rightarrow i}^l\right)$

to predict 13 quantum properties of molecules in the QM9 dataset.

Model performs extremely well with 11 out of 13 properties reaching “chemical accuracy”.

# Example 3: Transformers

MPNNs also encompass the transformer [9] model:

- $M_\theta^l(h_{v_i}^l, h_{v_j}^l, e_{ij}) = \text{MultiheadAttention}(h_{v_i}^l, h_{v_j}^l)$   
 $= \{w_{ij}^k(h_{v_i}^l, h_{v_j}^l), V_j^k(h_{v_j}^l)\}_{k=1}^K$
- $U_\theta^l(h_{v_i}^l, \square_{j \sim i} M_{j \rightarrow i}^l) = \text{LN}\left(\text{MLP}\left(\text{LN}\left(\sum_{j \sim i} w_{ij}^k V_j^k\right)\right)\right)$

where the graph is assumed to be *fully-connected*.

(See blogpost [8] for more details)

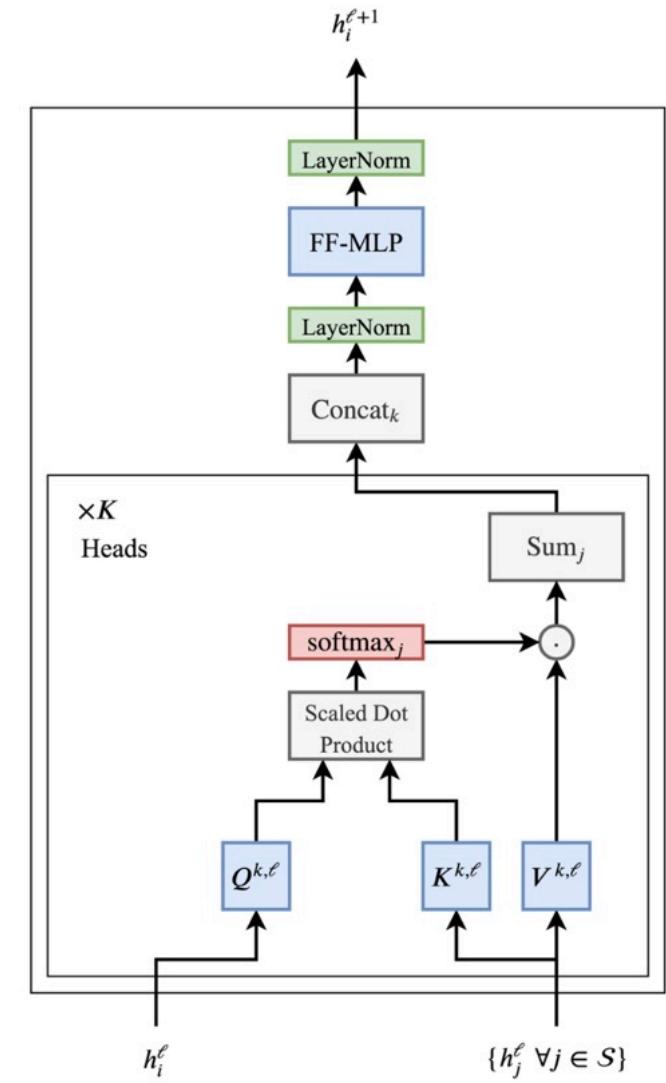


Image from [8]

# Comparison of MPNN with LBP

LBP	MPNN
<b>Bayesian.</b> Coupling between neighbours arise from prior knowledge of model. Message passing rule follows from laws of probability.	<b>Frequentist.</b> Message and state update rules are learned from data to obtain useful feature representations.
<b>Iterative.</b> States are updated iteratively to obtain better estimates of marginals.	<b>Deep.</b> Uses the power of deep learning to extract increasingly complex features with depth.
<b>Interpretable.</b> Prior assumptions are usually quite simple, making predictions interpretable.	<b>Flexible.</b> Processes high-dimensional node and edge features easily to model complex relations between inputs and outputs.

Many recent works aim to combine benefits of both approaches ([10] - [14])!

# Introduction to meta-learning

Brooks Paige  
ML Seminar

Term 2, 2021

# What is meta-learning?

Most machine learning models are

- ... trained to solve a single, specific task
- ... from a single, fixed dataset
- ... using a specific learning algorithm.

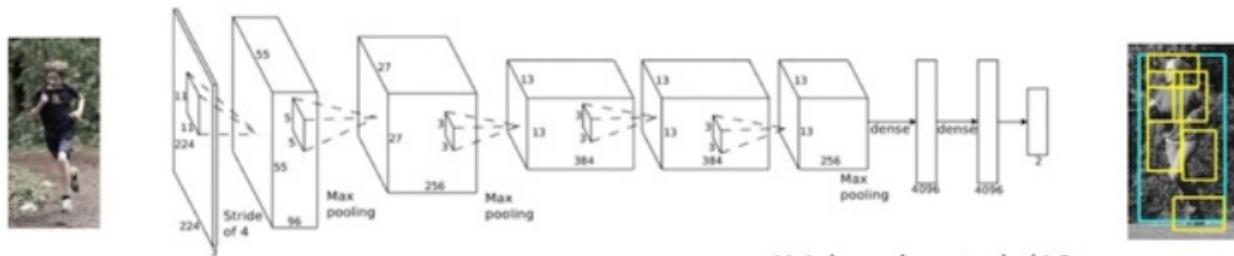
# What is meta-learning?

**Meta-learning** asks how we can **learn to learn**:

- If we have already solved  $n$  problems, shouldn't solving the  $(n + 1)^{\text{th}}$  be somehow easier?
- How can our learning algorithms gain "experience" over the course of many different but related learning problems?
- We no longer need to hand-design learning rules, or representations. Why do we hand-design learning algorithms?

# End-to-end learning in computer vision

Replacing hand-engineered features with end-to-end learning:



There's still a lot of hard-coded inductive bias in that architecture — and the learning algorithm is pre-defined

# Challenges

- Deep learning (and machine learning generally) requires huge amounts of labelled training data
- This is acceptable for solving isolated problems, but has a large real-world expense
- Are giant training sets really the path to solving “AI”?
- Never going to be appropriate in low-data regimes: personalized medicine, personalized recommendations, translating rare languages, . . .
- Humans don’t have this problem — we learn quickly!

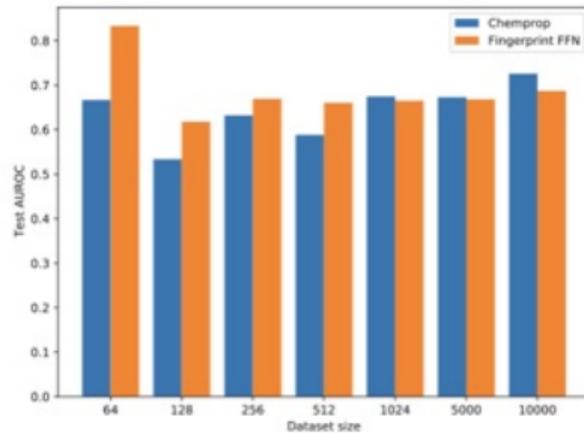
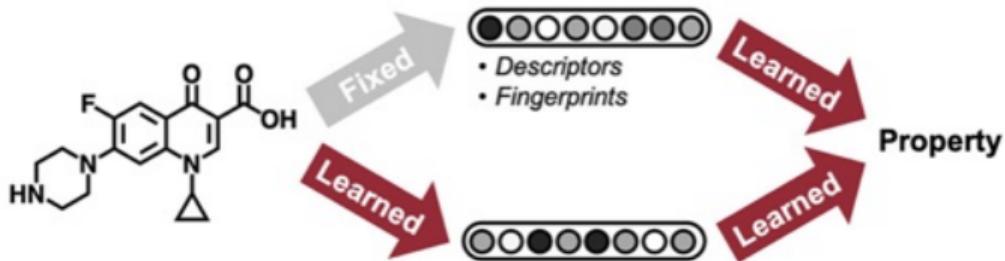
# Learning when we don't have much data



Why are you able to do this? **Prior experience.**

Figure: Chelsea Finn (cs330)

# Learning when data is expensive to collect



**Molecular property prediction:** on small datasets, hand-crafted features can still outperform deep learning.

[MSc project last year: addressing this with meta-learning!]

# Difficulties in definitions

The term “meta-learning” is used in many different communities, to mean many different things, and with varying terminology.

In general, for machine learning problems, **more data  $\Rightarrow$  better predictions**.

The goal in meta-learning: **more tasks  $\Rightarrow$  better learning algorithms**.

# Learning to learn: general framework

1. In “conventional” machine learning, a **model** improves its performance (e.g. its predictions) as it acquires more instances data
  - ▶ This machine learning model is called our **base learner**, which is trained to solve a specific task (e.g. a single classification problem) using an *inner* or *base* learning algorithm.
2. In meta-learning, a **learning algorithm** improves its performance over the course of multiple “learning episodes”
  - ▶ This is called our **meta-learning** or *outer learning* algorithm. It is responsible for training the inner learning algorithm!
  - ▶ A **learning episode** consists of a base algorithm, a model, and some measure of performance (e.g. generalization error, or convergence speed). Multiple learning episodes taken together provide the training data for the outer learning algorithm.

# Terminology

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 an algorithm  $A$  or its parameters  $\omega$ ,

- 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]*

*[alternative terminology]*

# Learning and meta-learning

Anatomy of a typical ML learning scenario:

- Dataset  $\mathcal{D}$ , e.g.  $\mathcal{D} = \{\mathbf{x}_1, y_1\}, \dots, \{\mathbf{x}_N, y_N\}$  for supervised learning
- Model specification to learn, e.g. parameters  $\theta$  of a predictor  $\hat{y} = f_{\theta}(\mathbf{x})$
- Loss  $\mathcal{L}(\mathcal{D}; \theta, \omega)$  which we minimize, e.g. error between true labels  $y$  and predicted labels  $\hat{y}$ .

This definition of the loss makes explicit a dependence on  $\omega$ , which are all the parameters of the learning process, including

- the learning algorithm  $A$ ,
- the function class or architecture of the model  $f$ ,
- the explicit form of the loss function,
- ...

# Cost per episode

- A **task**  $\mathcal{T}$  is defined as a dataset  $\mathcal{D}$  and a loss function  $\mathcal{L}$ , i.e.  $\mathcal{T} = \{\mathcal{D}, \mathcal{L}\}$ .
- An **episode** is defined by minimizing the loss  $\mathcal{L}$  on a single dataset  $\mathcal{D} = \{\mathcal{D}_{train}, \mathcal{D}_{test}\}$ . For example, assuming a probabilistic model with parameters  $\boldsymbol{\theta}$ , and a classification problem, this loss is

$$\mathcal{L}(\mathcal{D}; \omega) = \frac{1}{|\mathcal{D}_{test}|} \sum_{\{\mathbf{x}_i, y_i\} \in \mathcal{D}_{test}} -\log p(y_i | \mathbf{x}_i, \boldsymbol{\theta}^*)$$

- The value of  $\boldsymbol{\theta}^*$  for a particular episode is found by running the base learning algorithm  $A$ , with parameters  $\omega$ ; i.e.

$$\boldsymbol{\theta}^* = A(\mathcal{L}, \mathcal{D}_{train}; \omega).$$

# 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

$$\min_{\omega} \mathcal{L}^{meta}(\omega) = \mathbb{E}_{\{\mathcal{L}, \mathcal{D}\} \sim p(\mathcal{T})} [\mathcal{L}(\mathcal{D}; \omega)].$$

- If we have  $M$  initial example tasks  $\mathcal{T}^1, \dots, \mathcal{T}^M \sim p(\mathcal{T})$ , then our target optimal  $\omega^*$  are

$$\omega^* = \arg \min_{\omega} \sum_{m=1}^M \mathcal{L}^m(\mathcal{D}^m; \omega).$$

- In the meta-test phase, we can then make predictions on new training sets  $\mathcal{D}'_{train}$  by using  $\omega^*$ , with  $\theta^* = A(\mathcal{L}', \mathcal{D}'_{train}; \omega^*)$ .

In practice, this often corresponds to solving a two-level optimization problem.

# What are different tasks?

- Image classification, but on **previously unseen classes**
- Image classification, 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.  
(Tasks aren't "random"!)

# Some meta-learning-adjacent fields

A lot of different subareas in machine learning are similar to meta-learning.

- **Multi-task** learning also considers a dataset of many different tasks at training time. However, it aims to learn a set of parameters  $\theta$  which are appropriate for **all** tasks (instead of learning meta-parameters  $\omega$ ). It's also generally assumed all tasks are simultaneously accessible.
- **Transfer learning** aims to take knowledge from previous tasks, and use it to accelerate future tasks. This typically involves fitting  $\theta$  to previous task(s), and then using those parameters as initialization on a new tasks. The main difference from meta-learning is that there is **no meta-objective** — the previous tasks are not trained in a way which is aware that  $\theta$  will be used on new data later.

# Some meta-learning-adjacent fields

A lot of different subareas in machine learning are similar to meta-learning.

- **AutoML** aims to automate large parts of the machine learning pipeline — from algorithm selection, to dataset transforms, to neural network architectures. It's 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's desirable to find architectures which are generally helpful across a wide variety of problems, and to leverage knowledge that existing architectures (and algorithms) are useful on particular datasets.

# Three things to look for when reading meta-learning papers

Hospedales et al. (2020) set out the following useful taxonomy for meta-learning:

- **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  $\omega$ ? This might be gradient descent through the inner optimization of  $\theta$ , 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.

# Amortization

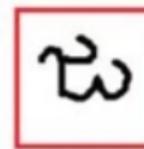
One other aspect to keep an eye out for is the amount of **amortization**.

- Idea of amortizing runtime costs: pay a large cost up front, in exchange for fast computation later
- Popular e.g. for running Bayesian inference repeatedly using the same model, on different datasets
- Different meta-learning methods have different degrees of amortization (e.g. learning an initialization less so than generating parameters directly)

# **Case study: few-shot learning**

# Learning from a single example

Which of these is like the other?



ಅ	ಇ	ಉ	ಯ	ಡ್
ಕೆ	ಲು	ಗ	ಚು	ರ್ಪು
ಇ	ರ	ಣ	ತೆ	ದ್
ನ	ಯು	ಲ	ಹ್	ಂ

# Few-shot learning as meta-learning

Suppose we want to know how to classify an image from very few examples (maybe only one!)

- We want to learn an algorithm  $A$  that outputs good parameters  $\theta$  for a model  $M$  given a very small training dataset  $\mathcal{D}$
- If we collect many such few-shot learning tasks, we can learn the algorithm  $A$  directly from data
- This will require a suitable **meta-objective**.

# Most machine learning datasets

## MNIST dataset

- 10 classes
  - 60,000 examples
- ≈ 6,000 images each



# Most real-world datasets?

## Omniglot dataset

- 1,623 classes
- ... across 50 alphabets
- ... with only 20 images each

[“transposed” MNIST]



# Example task

Here's an example task.

- Suppose we are shown 5 images, one each from five different classes:

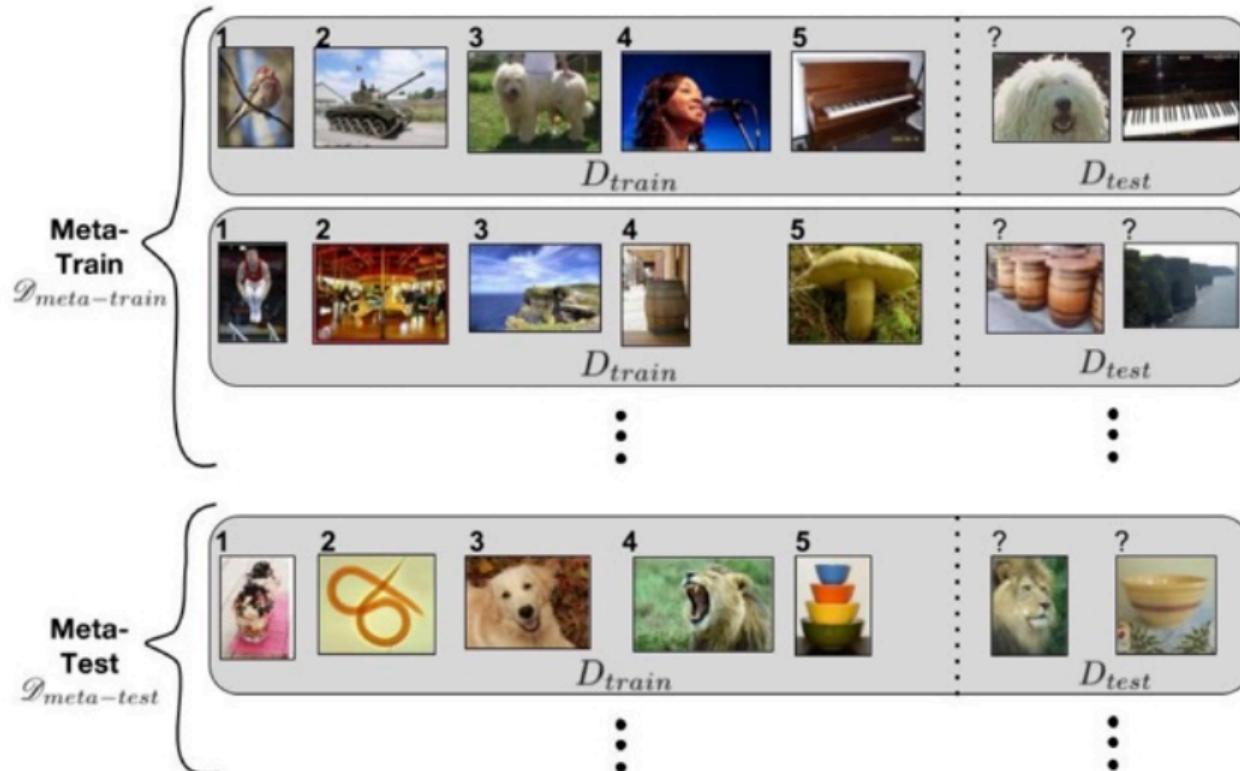


- Then, we are asked to predict the correct class of two brand-new test images:



These test images are from classes we've never seen before these five new images.

# Mini-ImageNet



# Choice of meta-representation

One approach is to start from an existing machine learning algorithm, add extra parameters, and learn those with meta-learning. We will look at three of these for classification:

- kNN or kernel classifier → Matching networks, Vinyals et al. (2016)
- Gaussian classifier → Prototypical networks, Snell et al. (2017)
- Gradient descent → MAML, Finn et al. (2017)

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)

# Matching networks

Classifier for new  $\mathbf{x}'$ :

$$\hat{y} = \sum_{i=1}^M a(\mathbf{x}', \mathbf{x}_i) y_i$$

where  $\{\mathbf{x}_i, y_i\}$  are the train set  $\mathcal{D}_{train}$  and  $\mathbf{x}' \in \mathcal{D}_{test}$ . We want to learn  $a(\dots)$ .

Advantages:

- Non-parametric method
- Can recover nearest neighbors and kernel density estimators
- Can also function as an associative memory that “points” to nearest training examples

# Matching networks

Classifier for new  $\mathbf{x}'$ :

$$\hat{y} = \sum_{i=1}^M a(\mathbf{x}', \mathbf{x}_i) y_i$$

where  $\{\mathbf{x}_i, y_i\}$  are the train set  $\mathcal{D}_{train}$  and  $\mathbf{x}' \in \mathcal{D}_{test}$ . We want to learn  $a(\dots)$ .

Idea: parameterize this in terms of an attention mechanism

$$a(\mathbf{x}', \mathbf{x}_i) = \frac{\exp\{c(f(\mathbf{x}'), g(\mathbf{x}_i))\}}{\sum_{j=1}^M \exp\{c(f(\mathbf{x}'), g(\mathbf{x}_j))\}},$$

where  $f, g$  are learnable functions, and  $c$  is the cosine distance

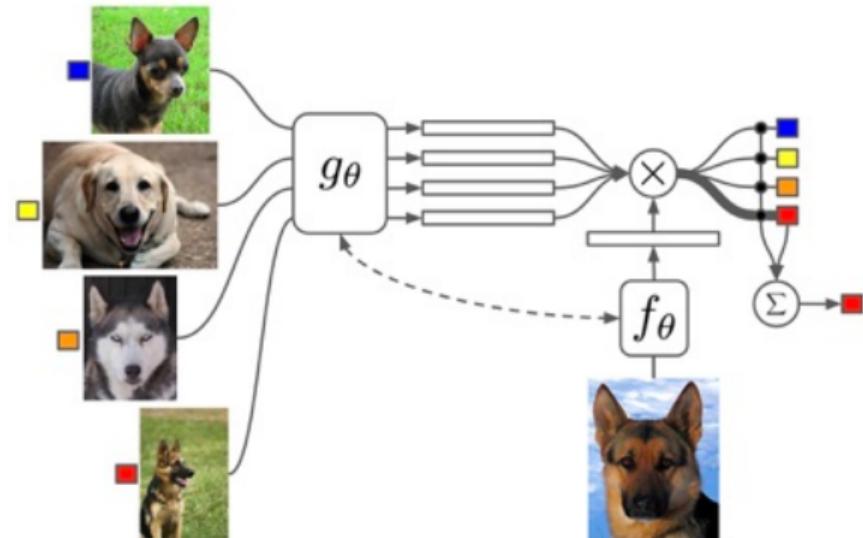
$$c(\mathbf{f}, \mathbf{g}) = 1 - \frac{\mathbf{f}^\top \mathbf{g}}{\|\mathbf{f}\| \|\mathbf{g}\|}.$$

# Matching networks

One trick: instead of just  $f_{\theta}(\mathbf{x}')$  and  $g_{\theta}(\mathbf{x}_i)$ , also provide access to an embedding of the entire training set, e.g. learning

- $f_{\theta}(\mathbf{x}') = f_{\theta}(\mathbf{x}', \mathcal{D}_{train})$
- $g_{\theta}(\mathbf{x}_i) = g_{\theta}(\mathbf{x}_i, \mathcal{D}_{train})$

where both are parameterized by sequence models (e.g. LSTMs).



# Prototypical networks

Learn an embedding  $f_{\theta}(\mathbf{x})$ , and compute class centroids  $\mathbf{c}_k$  as

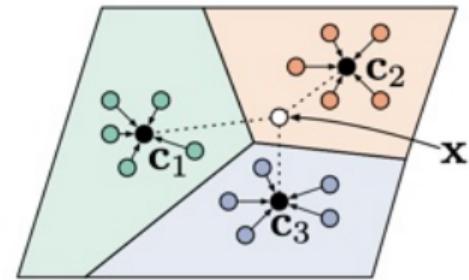
$$\mathbf{c}_k = \frac{1}{|\mathcal{S}_k|} \sum_{\{\mathbf{x}_i, y_i\} \in \mathcal{S}_k} f_{\theta}(\mathbf{x}_i)$$

where  $\mathcal{S}_k \subset \mathcal{D}_{train}$  is the set of training points with class label  $y_i = k$ .

Classifier for new  $\mathbf{x}'$ :

$$p(y = k | \mathbf{x}') = \frac{\exp\{-d(f_{\theta}(\mathbf{x}'), \mathbf{c}_k)\}}{\sum_{j=1}^K \exp\{-d(f_{\theta}(\mathbf{x}'), \mathbf{c}_j)\}}.$$

where  $d$  is a distance function.



# Prototypical networks

Since class predictions only depend on the means of the embeddings  $\mathbf{c}_k$ , this is much simpler than matching networks — there is no need for a complex embedding of the training sets.

- With a Euclidean distance  $d(\cdot, \cdot) = \|f_{\theta}(\mathbf{x}') - \mathbf{c}_k\|^2$ , this is interpretable as learning a linear model in the embedding space  $f_{\theta}(\mathbf{x})$ :

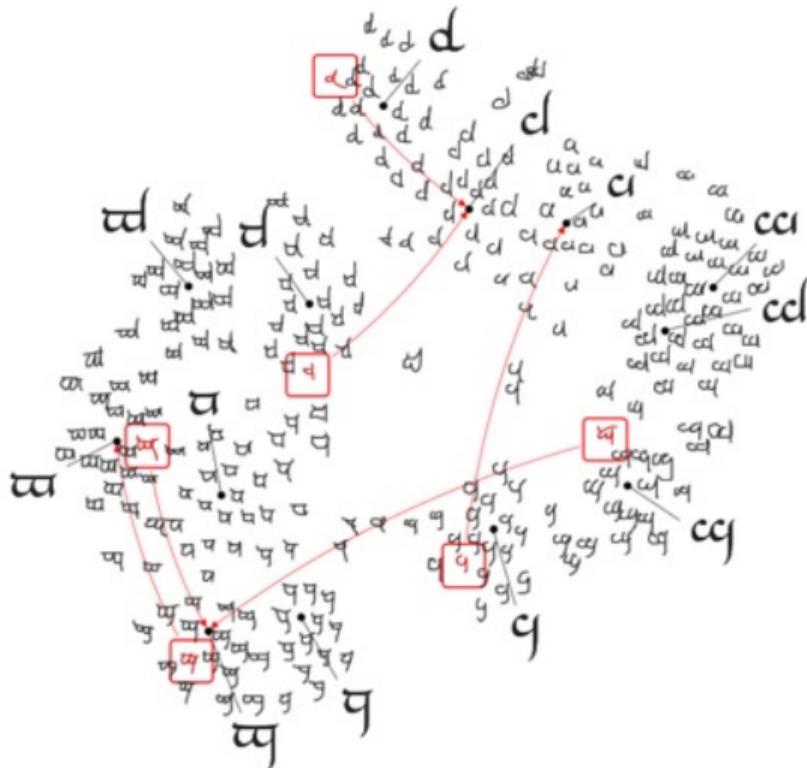
$$\begin{aligned}-\|f_{\theta}(\mathbf{x}') - \mathbf{c}_k\|^2 &= -f_{\theta}(\mathbf{x}')^\top f_{\theta}(\mathbf{x}') + 2\mathbf{c}_k^\top f_{\theta}(\mathbf{x}') - \mathbf{c}_k^\top \mathbf{c}_k \\&= \underbrace{(2\mathbf{c}_k)^\top}_{\mathbf{w}_k} f_{\theta}(\mathbf{x}') - \underbrace{\mathbf{c}_k^\top \mathbf{c}_k}_{b_k} + \text{const}\end{aligned}$$

- The learned network  $f_{\theta}$  produces, in a feedforward manner, the embedding space as well as the weights of the corresponding linear classifier.

# Prototypical networks

Example embedding for an  
Omniglot meta-test task

- T-SNE visualization
- Black highlights are class centers
- Red highlights are misclassification errors

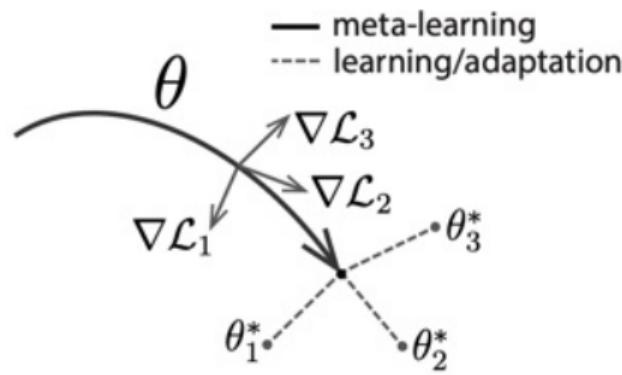


**(This one is simple enough to  
implement that I can actually  
do a live demo)**

# Model-agnostic meta learning (MAML)

A downside of the previous method was that it is not obvious how to apply outside of classification settings.

- MAML: meta-representation is a parameter vector  $\theta$  for some (any) deep learning model
- Learn a general **initialization**, from which we can quickly fine-tune to new classes
- Unlike “typical” transfer learning, where the initial  $\theta$  subject to fine-tuning come from some unrelated network, here there is an explicit meta-learning objective



# Model-agnostic meta learning (MAML)

---

**Algorithm 1** Model-Agnostic Meta-Learning

---

**Require:**  $p(\mathcal{T})$ : distribution over tasks

**Require:**  $\alpha, \beta$ : step size hyperparameters

- 1: randomly initialize  $\theta$
  - 2: **while** not done **do**
  - 3:   Sample batch of tasks  $\mathcal{T}_i \sim p(\mathcal{T})$
  - 4:   **for all**  $\mathcal{T}_i$  **do**
  - 5:     Evaluate  $\nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta})$  with respect to  $K$  examples
  - 6:     Compute adapted parameters with gradient descent:  $\theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta})$
  - 7:   **end for**
  - 8:   Update  $\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i})$
  - 9: **end while**
-

# **Black-box approaches**

# Black-box approach

Train a neural architecture to directly output predictions at the “next” task.

1. Train a network which directly outputs parameters of a model, e.g. for task  $\mathcal{T}_i$ ,

$$\omega_i = f_{\theta}(\mathcal{D}_{train}^i)$$

2. Make predictions on test data with

$$y = g_{\omega_i}(\mathbf{x}_{i,test})$$

**Clear training objective** — this is now just supervised learning. But, complex models are required, and extending to large datasets is hard.

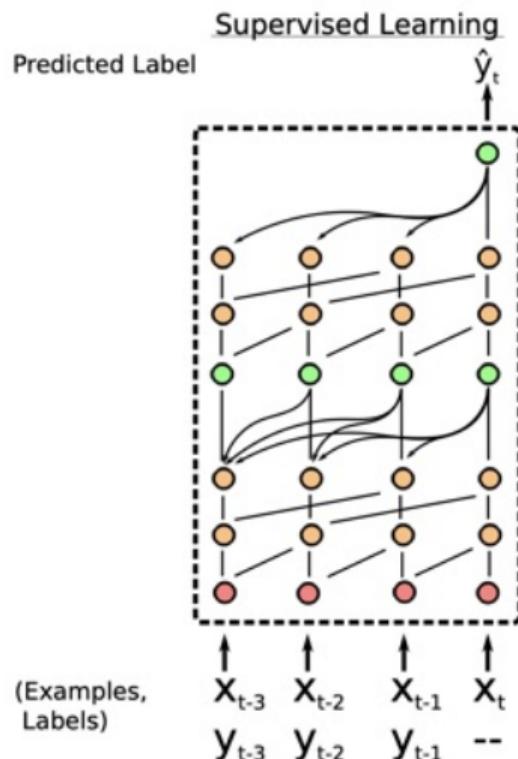


Figure: Mishra et al. (2018)

# Other things you can meta-learn

- **Neural network architectures:** This is a big one — network architectures (e.g. convnets, resnets, LSTMs, GNNs) have strong inductive biases. Can we learn architectures (or building blocks) which generalize well to many tasks?
- **Optimizers:** Crudely, an optimizer is a function that takes a current value  $\theta^{(t)}$  and a gradient g, which returns a new value  $\theta^{(t+1)}$ . Can we learn this function?
- **Losses or rewards:** what if you wanted an alternative objective function for your inner learning tasks — for example, a loss function which was robust to label noise? Can it generalize to other datasets?
- **Data augmentation:** if you have many different potential data transformations, which are likely to be label-preserving, which ones should you include when training? Can we learn a data augmentation policy?
- ...

## ... and other applications

- Domain adaptation,
- continual learning,
- defense against adversarial attacks, ...

Many more ideas (with citations) in Hospedales et al. (2020).