

11 *Figure 10.1: Illustration of variational inference. The large oval represents the set of variational distributions*
12 *$\mathcal{Q} = \{q(\mathbf{z}; \boldsymbol{\psi}) : \boldsymbol{\psi} \in \mathbb{R}^K\}$, where K is the number of variational parameters. The true distribution is the point*
13 *$p(\mathbf{z}|\mathbf{x})$, which we assume lies outside the set. Our goal is to find the best approximation to p within our*
14 *variational family; this is the point $\boldsymbol{\psi}^*$ which is closest in KL divergence. We find this point by starting an*
15 *optimization procedure from the random initial point $\boldsymbol{\psi}^{init}$. Adapted from a figure by David Blei.*

1617

18 If we define $\mathcal{E}(\mathbf{z}) = -\log p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})$ as the energy, then we can write

19

$$\mathcal{L}(\boldsymbol{\psi}|\boldsymbol{\theta}, \mathbf{x}) = \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\psi})} [\mathcal{E}(\mathbf{z})] - \mathbb{H}(q) \quad (10.6)$$

21

22 where $\mathbb{H}(q)$ is the entropy. In physics, this is known as the **variational free energy**. We can
23 interpret this as the expected energy minus the entropy:

24

$$\text{VFE} = \text{expected energy} - \text{entropy} \quad (10.7)$$

25

26 This is an upper bound on the **free energy**, $-\log p_{\boldsymbol{\theta}}(\mathbf{x})$, which follows from the fact that

27

$$D_{\text{KL}}(q||p) = \text{VFE}(\boldsymbol{\psi}|\boldsymbol{\theta}, \mathbf{x}) + \log p_{\boldsymbol{\theta}}(\mathbf{x}) \geq 0 \quad (10.8)$$

29

30 Our goal is to minimize the VFE. See Figure 10.1 for an illustration.

31

32 10.1.2 Evidence lower bound (ELBO)

33 The negative of the VFE is known as the **evidence lower bound** or **ELBO** function [BKM16]:
34

$$\mathcal{L}(\boldsymbol{\psi}|\boldsymbol{\theta}, \mathbf{x}) \triangleq \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\psi})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})] \quad (10.9)$$

36

37 The name “ELBO” arises because

38

$$\mathcal{L}(\boldsymbol{\psi}|\boldsymbol{\theta}, \mathbf{x}) \leq \log p_{\boldsymbol{\theta}}(\mathbf{x}) \quad (10.10)$$

39

40 where $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ is called the “evidence”. The inequality follows from Equation (10.8). Therefore
41 maximizing the ELBO wrt $\boldsymbol{\psi}$ will decrease the original KL, since $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ is a constant wrt $\boldsymbol{\psi}$.
42 (Note: we use the symbol \mathcal{L} for the ELBO, rather than \mathcal{L} , since the latter denotes a loss we want to
43 minimize.)

44

45 We can rewrite the ELBO as follows:

46

$$\mathcal{L}(\boldsymbol{\psi}|\boldsymbol{\theta}, \mathbf{x}) = \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\psi})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] + \mathbb{H}(q(\mathbf{z}|\boldsymbol{\psi})) \quad (10.11)$$

47

We can interpret this

$$\text{ELBO} = \text{expected log joint} + \text{entropy of posterior} \quad (10.12)$$

The second term encourages the posterior to be maximum entropy, while the first term encourages it to be a joint MAP configuration.

We can also rewrite the ELBO in the following equivalent way:

$$\mathbb{E}(\psi|\theta, x) = \mathbb{E}_{q(z|\psi)} [\log p_\theta(x|z)] - D_{\text{KL}}(q(z|\psi) \| p_\theta(z)) \quad (10.13)$$

We can interpret this as follows:

$$\text{ELBO} = \text{expected log likelihood} - \text{KL from posterior to prior} \quad (10.14)$$

The KL term acts like a regularizer, preventing the posterior from diverging too much from the prior.

10.2 Mean field VI

A common approximation in variational inference is to assume that all the latent variables are independent of each other, i.e.

$$q(\mathbf{z}|\psi) = \prod_{j=1}^J q_j(z_j) \quad (10.15)$$

where J is the number of hidden variables, and $q_j(z_j)$ is shorthand for $q_{\psi_j}(z_j)$, where ψ_j are the variational parameters for the j 'th distribution. This is called the **mean field** approximation.

From Equation (10.11), the ELBO becomes

$$\mathbb{L}(\psi) = \int q(z|\psi) \log p_{\theta}(x, z) dz + \sum_{j=1}^J \mathbb{H}(q_j) \quad (10.16)$$

since the entropy of a product distribution is the sum of entropies of each component in the product.

since the entropy of a product distribution is the sum of entropies of each component in the product. We can either directly optimize this (see e.g., [Baq+16]), or use a coordinate-wise optimization scheme as we discuss in Section 10.2.1.

10.2.1 Coordinate ascent variational inference (CAVI)

In this section, we discuss a coordinate ascent method for optimizing the mean field objective, which we call **coordinate ascent variational inference** or **CAVI**.

To derive the update equations, we initially assume there are just 3 discrete latent variables, so the ELBO is given by

$$L(q_1, q_2, q_3) = \sum_{z_1} \sum_{z_2} \sum_{z_3} q_1(z_1) q_2(z_2) q_3(z_3) \log \tilde{p}(z_1, z_2, z_3) + \sum_{i=1}^3 \mathbb{H}(q_j) \quad (10.17)$$

where $\tilde{p}(\mathbf{z}) = p_{\theta}(\mathbf{z}, \mathbf{x})$ is the unnormalized joint. (We omit θ and \mathbf{x} from the notation, since in this section, we assume both are fixed, i.e., we are just performing inference for the latent variables given

1 one example and a fixed set of parameters.) We will optimize this wrt each q_i , one at a time, keeping
2 the others fixed. Let us look at the objective for q_2 :

3

$$\mathcal{L}_2(q_2) = \sum_{z_2} q_2(z_2) \left[\sum_{z_1} \sum_{z_3} q_1(z_1) q_3(z_3) \log \tilde{p}(z_1, z_2, z_3) \right] + \mathbb{H}(q_2) + \text{const} \quad (10.18)$$

4

$$= \sum_{z_2} q_2(z_2) \left[\log \tilde{f}_2(z_2) - \log q_2(z_2) \right] + \text{const} \quad (10.19)$$

5 where

6

$$\tilde{f}_2(z_2) \triangleq \exp \left[\sum_{z_1} \sum_{z_3} q_1(z_1) q_3(z_3) \log \tilde{p}(z_1, z_2, z_3) \right] = \exp \left[\mathbb{E}_{\mathbf{z}_{-2}} [\log \tilde{p}(z_1, z_2, z_3)] \right] \quad (10.20)$$

7 where $\mathbf{z}_{-2} = (z_2, z_3)$ is all variables except z_2 . If we normalize \tilde{f}_2 to make it a distribution, $f_2(z_2)$,
8 we can rewrite Equation (10.19) as follows:

9

$$\mathcal{L}_2(q_2) = -D_{\text{KL}}(q_2 \| f_2) + \text{const} \quad (10.21)$$

10 Since $D_{\text{KL}}(q_2 \| f_2)$ achieves its minimal value of 0 when $q_2(z_2) = f_2(z_2)$, we see that $q_2^*(z_2) = f_2(z_2)$.

11 In general, when we have J groups of variables, the mean field ELBO is given by

12

$$\mathcal{L}(\mathbf{q}) = \sum_{\mathbf{z}_1} \cdots \sum_{\mathbf{z}_J} q_1(\mathbf{z}_1) \cdots q_J(\mathbf{z}_J) \log \tilde{p}(\mathbf{z}_1, \dots, \mathbf{z}_J) + \sum_{j=1}^J \mathbb{H}(q_j) \quad (10.22)$$

13 The optimal estimate of the marginal posterior for each variable is given by

14

$$q_i(\mathbf{z}_i) \propto \exp (\mathbb{E}_{\mathbf{q}_{-i}} [\log \tilde{p}(\mathbf{z})]) \quad (10.23)$$

15 where $\mathbb{E}_{\mathbf{q}_{-i}} [\log \tilde{p}(\mathbf{z})]$ takes the expectation wrt all variables except \mathbf{z}_i . The CAVI method simply
16 computes q_i for each dimension i in turn, in an iterative fashion. One can show that this coordinate
17 ascent procedure is guaranteed to converge to a local optimum.

18 10.2.2 Example: CAVI for the Ising model

19 In this section, we apply CAVI to perform mean field inference in an Ising model (Section 4.3.2.1),
20 which is a kind of Markov random field defined on binary random variables, $z_i \in \{-1, +1\}$, arranged
21 in a 2d grid.

22 Originally Ising models were developed as models of atomic spins for magnetic materials, although
23 we will apply them to an image denoising problem. Specifically, let z_i be the hidden value of pixel i ,
24 and $x_i \in \mathbb{R}$ be the observed noisy value. See Figure 10.2 for the graphical model.

25 Let $L_i(z_i) \triangleq \log p(x_i | z_i)$ be the log likelihood for the i 'th pixel (aka the **local evidence** for node i
26 in the graphical model). The overall likelihood has the form

27

$$p(\mathbf{x} | \mathbf{z}) = \prod_i p(x_i | z_i) = \exp \left(\sum_i L_i(z_i) \right) \quad (10.24)$$

28

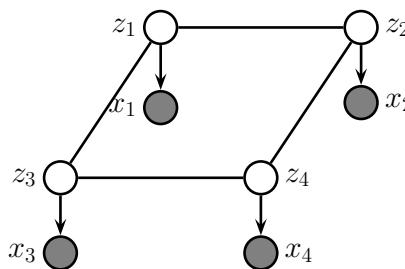


Figure 10.2: A grid-structured MRF with hidden nodes z_i and local evidence nodes x_i . The prior $p(\mathbf{z})$ is an undirected Ising model, and the likelihood $p(\mathbf{x}|\mathbf{z}) = \prod_i p(x_i|z_i)$ is a directed fully factored model.

Our goal is to approximate the posterior $p(\mathbf{z}|\mathbf{x})$. We will use an Ising model for the prior:

$$p(\mathbf{z}) = \frac{1}{Z_0} \exp(-\mathcal{E}_0(\mathbf{z})) \quad (10.25)$$

$$\mathcal{E}_0(\mathbf{z}) = - \sum_{i \sim j} W_{ij} z_i z_j \quad (10.26)$$

where we sum over each $i - j$ edge. Therefore the posterior has the form

$$p(\mathbf{z}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp(-\mathcal{E}(\mathbf{z})) \quad (10.27)$$

$$\mathcal{E}(\mathbf{z}) = \mathcal{E}_0(\mathbf{z}) - \sum_i L_i(z_i) \quad (10.28)$$

We will now make the following fully factored approximation:

$$q(\mathbf{z}) = \prod_i q_i(z_i) = \prod_i \text{Ber}(z_i|\mu_i) \quad (10.29)$$

where $\mu_i = \mathbb{E}_{q_i}[z_i]$ is the mean value of node i . To derive the update for the variational parameter μ_i , we first compute the unnormalized log joint, $\log \tilde{p}(\mathbf{z}) = -\mathcal{E}(\mathbf{z})$, dropping terms that do not involve z_i :

$$\log \tilde{p}(\mathbf{z}) = z_i \sum_{j \in \text{nbr}_i} W_{ij} z_j + L_i(z_i) + \text{const} \quad (10.30)$$

This only depends on the states of the neighboring nodes. Hence

$$q_i(z_i) \propto \exp(\mathbb{E}_{q_{-i}(\mathbf{z})} [\log \tilde{p}(\mathbf{z})]) = \exp \left(z_i \sum_{j \in \text{nbr}_i} W_{ij} \mu_j + L_i(z_i) \right) \quad (10.31)$$

where $q_{-i}(\mathbf{z}) = \prod_{j \neq i} q_j(z_j)$. Thus we replace the states of the neighbors by their average values.

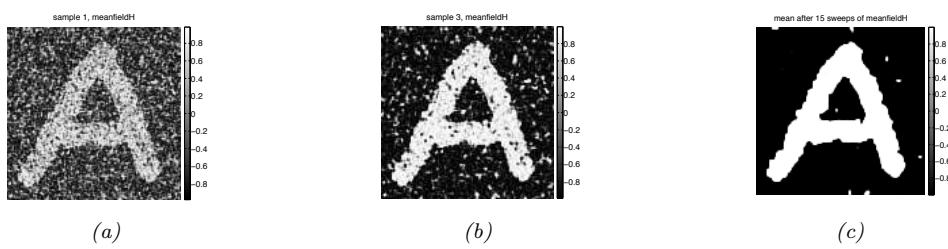


Figure 10.3: Example of image denoising using mean field (with parallel updates and a damping factor of 0.5). We use an Ising prior with $W_{ij} = 1$ and a Gaussian noise model with $\sigma = 2$. We show the results after 1, 3 and 15 iterations across the image. Compare to Figure 12.6, which shows the results of using Gibbs sampling.

Generated by `ising_image_denoise_demo.py`.

We now simplify this expression. Let $m_i = \sum_{j \in \text{nbr}_i} W_{ij} \mu_j$ be the mean field influence on node i . Also, let $L_i^+ \triangleq L_i(+1)$ and $L_i^- \triangleq L_i(-1)$. The approximate marginal posterior is given by

$$q_i(z_i = 1) = \frac{e^{m_i + L_i^+}}{e^{m_i + L_i^+} + e^{-m_i + L_i^-}} = \frac{1}{1 + e^{-2m_i + L_i^- - L_i^+}} = \sigma(2a_i) \quad (10.32)$$

$$a_i \triangleq m_i + 0.5(L_i^+ - L_i^-) \quad (10.33)$$

Similarly, we have $q_i(z_i = -1) = \sigma(-2a_i)$. From this we can compute the new mean for site i :

$$\mu_i = \mathbb{E}_{q_i}[z_i] = q_i(z_i = +1) \cdot (+1) + q_i(z_i = -1) \cdot (-1) \quad (10.34)$$

$$= \frac{1}{1 + e^{-2a_i}} - \frac{1}{1 + e^{2a_i}} = \frac{e^{a_i}}{e^{a_i} + e^{-a_i}} - \frac{e^{-a_i}}{e^{-a_i} + e^{a_i}} = \tanh(a_i) \quad (10.35)$$

We can turn the above equations into a fixed point algorithm by writing

$$\mu_i^t = \tanh \left(\sum_{j \in \text{nbr}_i} W_{ij} \mu_j^{t-1} + 0.5(L_i^+ - L_i^-) \right) \quad (10.36)$$

It is usually better to use **damped updates** of the form

$$\mu_i^t = (1 - \lambda) \mu_i^{t-1} + \lambda \tanh \left(\sum_{j \in \text{nbr}_i} W_{ij} \mu_j^{t-1} + 0.5(L_i^+ - L_i^-) \right) \quad (10.37)$$

for $0 < \lambda < 1$. We can update all the nodes in parallel, or update them asynchronously.

Figure 10.3 shows the method in action, applied to a 2d Ising model with homogeneous attractive potentials, $W_{ij} = 1$. We use parallel updates with a damping factor of $\lambda = 0.5$. (If we don't use damping, we tend to get "checkerboard" artefacts.)

41

10.2.3 Variational Bayes

In Bayesian modeling, we treat the parameters θ as latent variables. Thus our goal is to approximate the parameter posterior $p(\theta|\mathcal{D}) \propto p(\theta)p(\mathcal{D}|\theta)$. Applying VI to this problem is called **variational Bayes** [Att00].

47

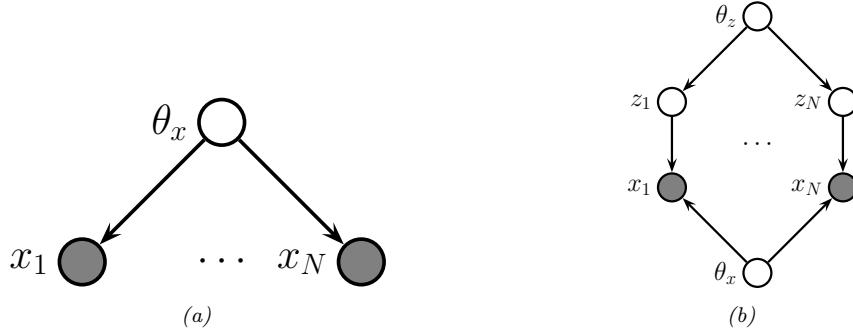


Figure 10.4: Graphical models with (a) Global hidden variable θ_x and observed variables $\mathbf{x}_{1:N}$. (b) Local hidden variables $\mathbf{z}_{1:N}$, global hidden variables θ_x, θ_z , and observed variables $\mathbf{x}_{1:N}$.

In this section, we assume there are no latent variables except for the shared global parameters, so the model has the form

$$p(\boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathcal{D}_n | \boldsymbol{\theta}) \quad (10.38)$$

These conditional independencies are illustrated in Figure 10.4a.

We will fit the variational posterior by maximizing the ELBO

$$\mathcal{L}(\boldsymbol{\psi}_{\boldsymbol{\theta}} | \mathcal{D}) = \mathbb{E}_{q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}})} [\log p(\boldsymbol{\theta}, \mathcal{D})] + \mathbb{H}(q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}})) \quad (10.39)$$

We will assume the variational posterior factorizes over the parameters:

$$q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}}) = \prod_j q(\boldsymbol{\theta}_j | \boldsymbol{\psi}_{\boldsymbol{\theta}_j}) \quad (10.40)$$

We can then update each $\boldsymbol{\psi}_{\boldsymbol{\theta}_j}$ using CAVI (Section 10.2.1).

10.2.4 Example: VB for a univariate Gaussian

Consider inferring the parameters of a 1d Gaussian. The likelihood is given by $p(\mathcal{D} | \boldsymbol{\theta}) = \prod_{n=1}^N \mathcal{N}(x_n | \mu, \lambda^{-1})$, where μ is the mean and λ is the precision. Suppose we use a conjugate prior of the form

$$p(\mu, \lambda) = \mathcal{N}(\mu | \mu_0, (\kappa_0 \lambda)^{-1}) \text{Ga}(\lambda | a_0, b_0) \quad (10.41)$$

It is possible to derive the posterior $p(\mu, \lambda | \mathcal{D})$ for this model exactly, as shown in Section 3.2.3.3. However, here we use the VB method with the following factored approximate posterior:

$$q(\mu, \lambda) = q(\mu | \boldsymbol{\psi}_{\mu}) q(\lambda | \boldsymbol{\psi}_{\lambda}) \quad (10.42)$$

We do not need to specify the forms for the distributions $q(\mu | \boldsymbol{\psi}_{\mu})$ and $q(\lambda | \boldsymbol{\psi}_{\lambda})$; the optimal forms will “fall out” automatically during the derivation (and conveniently, they turn out to be Gaussian and Gamma respectively). Our presentation follows [Mac03, p429].

1
2 **10.2.4.1 Target distribution**

3 The unnormalized log posterior has the form
4

$$\log \tilde{p}(\mu, \lambda) = \log p(\mu, \lambda, \mathcal{D}) = \log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda) + \log p(\lambda) \quad (10.43)$$

$$= \frac{N}{2} \log \lambda - \frac{\lambda}{2} \sum_{n=1}^N (x_n - \mu)^2 - \frac{\kappa_0 \lambda}{2} (\mu - \mu_0)^2$$

$$+ \frac{1}{2} \log(\kappa_0 \lambda) + (a_0 - 1) \log \lambda - b_0 \lambda + \text{const} \quad (10.44)$$

12
13 **10.2.4.2 Updating $q(\mu|\psi_\mu)$**

14 The optimal form for $q(\mu|\psi_\mu)$ is obtained by averaging over λ :
15

$$\log q(\mu|\psi_\mu) = \mathbb{E}_{q(\lambda|\psi_\lambda)} [\log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda)] + \text{const} \quad (10.45)$$

$$= -\frac{\mathbb{E}_{q(\lambda|\psi_\lambda)} [\lambda]}{2} \left\{ \kappa_0(\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right\} + \text{const} \quad (10.46)$$

21 By completing the square one can show that $q(\mu|\psi_\mu) = \mathcal{N}(\mu|\mu_N, \kappa_N^{-1})$, where
22

$$\mu_N = \frac{\kappa_0 \mu_0 + N \bar{x}}{\kappa_0 + N}, \quad \kappa_N = (\kappa_0 + N) \mathbb{E}_{q(\lambda|\psi_\lambda)} [\lambda] \quad (10.47)$$

26 At this stage we don't know what $q(\lambda|\psi_\lambda)$ is, and hence we cannot compute $\mathbb{E}[\lambda]$, but we will derive
27 this below.
28

29
30 **10.2.4.3 Updating $q(\lambda|\psi_\lambda)$**

31 The optimal form for $q(\lambda|\psi_\lambda)$ is given by
32

$$\log q(\lambda|\psi_\lambda) = \mathbb{E}_{q(\mu|\psi_\mu)} [\log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda) + \log p(\lambda)] + \text{const} \quad (10.48)$$

$$= (a_0 - 1) \log \lambda - b_0 \lambda + \frac{1}{2} \log \lambda + \frac{N}{2} \log \lambda \\ - \frac{\lambda}{2} \mathbb{E}_{q(\mu|\psi_\mu)} \left[\kappa_0(\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right] + \text{const} \quad (10.49)$$

40 We recognize this as the log of a Gamma distribution, hence $q(\lambda|\psi_\lambda) = \text{Ga}(\lambda|a_N, b_N)$, where
41

$$a_N = a_0 + \frac{N + 1}{2} \quad (10.50)$$

$$b_N = b_0 + \frac{1}{2} \mathbb{E}_{q(\mu|\psi_\mu)} \left[\kappa_0(\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right] \quad (10.51)$$

1 **10.2.4.4 Computing the expectations**

3 To implement the updates, we have to specify how to compute the various expectations. Since
4 $q(\mu) = \mathcal{N}(\mu|\mu_N, \kappa_N^{-1})$, we have
5

6 $\mathbb{E}_{q(\mu)} [\mu] = \mu_N \tag{10.52}$

7 $\mathbb{E}_{q(\mu)} [\mu^2] = \frac{1}{\kappa_N} + \mu_N^2 \tag{10.53}$

10 Since $q(\lambda) = \text{Ga}(\lambda|a_N, b_N)$, we have

11 $\mathbb{E}_{q(\lambda)} [\lambda] = \frac{a_N}{b_N} \tag{10.54}$

14 We can now give explicit forms for the update equations. For $q(\mu)$ we have

16 $\mu_N = \frac{\kappa_0 \mu_0 + N \bar{x}}{\kappa_0 + N} \tag{10.55}$

18 $\kappa_N = (\kappa_0 + N) \frac{a_N}{b_N} \tag{10.56}$

20 and for $q(\lambda)$ we have

22 $a_N = a_0 + \frac{N+1}{2} \tag{10.57}$

24 $b_N = b_0 + \frac{1}{2} \kappa_0 (\mathbb{E} [\mu^2] + \mu_0^2 - 2\mathbb{E} [\mu] \mu_0) + \frac{1}{2} \sum_{n=1}^N (x_n^2 + \mathbb{E} [\mu^2] - 2\mathbb{E} [\mu] x_n) \tag{10.58}$

27 We see that μ_N and a_N are in fact fixed constants, and only κ_N and b_N need to be updated
28 iteratively. (In fact, one can solve for the fixed points of κ_N and b_N analytically, but we don't do
29 this here in order to illustrate the iterative updating scheme.)
30

31 **10.2.4.5 Illustration**

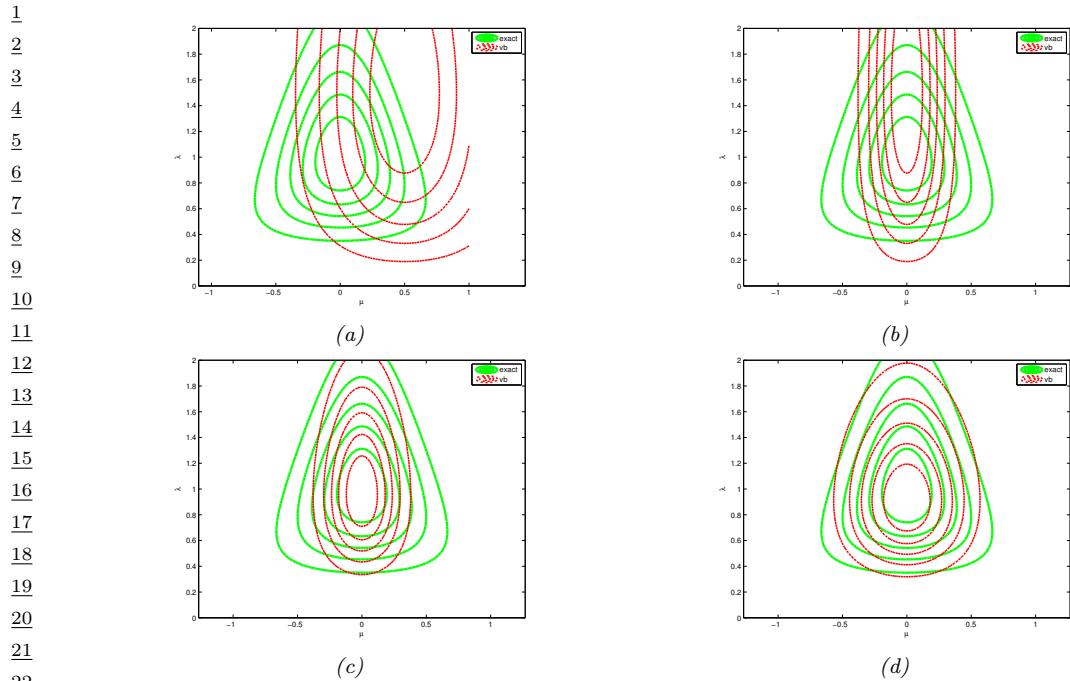
33 Figure 10.5 gives an example of this method in action. The green contours represent the exact
34 posterior, which is Gaussian-Gamma. The dotted red contours represent the variational approximation
35 over several iterations. We see that the final approximation is reasonably close to the exact solution.
36 However, it is more "compact" than the true distribution. It is often the case that mean field inference
37 underestimates the posterior uncertainty, for reasons explained in Section 5.1.3.2.

39 **10.2.4.6 Lower bound**

41 In VB, we maximize a lower bound on the log marginal likelihood:

42 $\mathcal{L}(\psi_\theta | \mathcal{D}) \leq \log p(\mathcal{D}) = \log \int \int p(\mathcal{D} | \mu, \lambda) p(\mu, \lambda) d\mu d\lambda \tag{10.59}$

45 It is very useful to compute the lower bound itself, for three reasons. First, it can be used to assess
46 convergence of the algorithm. Second, it can be used to assess the correctness of one's code: as with
47



²³ Figure 10.5: Factored variational approximation (red) to the Gaussian-Gamma distribution (green). (a)
²⁴ Initial guess. (b) After updating $q(\mu|\psi_\mu)$. (c) After updating $q(\lambda|\psi_\lambda)$. (d) At convergence (after 5 iterations).
²⁵ Adapted from Fig. 10.4 of [Bis06]. Generated by `unigauss vb demo.py`.

²⁸ EM, if we use CAVI to optimize the objective, the bound should increase monotonically at each
²⁹ iteration, otherwise there must be a bug. Third, the bound can be used as an approximation to the
³⁰ marginal likelihood, which can be used for Bayesian model selection. One can show that the lower
³¹ bound has the following form:

$$\frac{33}{34} \quad \mathcal{L} = \text{const} + \frac{1}{2} \ln \frac{1}{\kappa_N} + \ln \Gamma(a_N) - a_N \ln b_N \quad (10.60)$$

36 10.2.5 Variational Bayes EM

In Bayesian latent variable models, we have two forms of hidden variables: local (or per example) hidden variables z_n , and global (shared) hidden variables θ , which represent the parameters of the model. See Figure 10.4b for an illustration. (Note that the parameters, which are fixed in number, are sometimes called **intrinsic variables**, whereas the local hidden variables are called **extrinsic variables**.) If $h = (\theta, z_{1:N})$ represents all the hidden variables, then the joint distribution is given by

$$\begin{aligned} & \frac{44}{45} \quad p(\mathbf{h}, \mathcal{D}) = p(\boldsymbol{\theta}, \mathbf{z}_{1:N}, \mathcal{D}) = p(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathbf{z}_n | \boldsymbol{\theta}) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}) \\ & \frac{46}{47} \end{aligned} \quad (10.61)$$

1 We will make the following mean field assumption:

2

$$\underline{q}(\boldsymbol{\theta}, \mathbf{z}_{1:N}) = q(\boldsymbol{\theta}|\psi_{\boldsymbol{\theta}}) \prod_{n=1}^N q(\mathbf{z}_n|\psi_n) \quad (10.62)$$

3

4 We will use VI to maximize the ELBO:

5

$$\underline{L}(\psi_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}} | \mathcal{D}) = \mathbb{E}_{q(\boldsymbol{\theta}, \mathbf{z}_{1:N} | \psi_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}})} [\log p(\mathbf{z}_{1:N}, \boldsymbol{\theta}, \mathcal{D}) - \log q(\boldsymbol{\theta}, \mathbf{z}_{1:N} | \psi_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}})] \quad (10.63)$$

6

7 If we use the mean field assumption, then we can apply the CAVI approach to optimize each set of
8 variational parameters. In particular, we can alternate between optimizing the $q_n(\mathbf{z}_n)$ in parallel,
9 independently of each other, with $q(\boldsymbol{\theta})$ held fixed, and then optimizing $q(\boldsymbol{\theta})$ with the q_n held fixed.
10 This is known as **variational Bayes EM** [BG06]. It is similar to regular EM, except in the E step,
11 we infer an approximate posterior for \mathbf{z}_n averaging out the parameters (instead of plugging in a point
12 estimate), and in the M step, we update the parameter posterior parameters using the expected
13 sufficient statistics.

14 Now suppose we approximate $q(\boldsymbol{\theta})$ by a delta function, $q(\boldsymbol{\theta}) = \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})$. The Bayesian LVM ELBO
15 objective from Equation (10.63) simplifies to the the “LVM ELBO”:

16

$$\underline{L}(\psi_{1:N}, \boldsymbol{\theta} | \mathcal{D}) = \mathbb{E}_{q(\mathbf{z}_{1:N} | \psi_{1:N})} [\log p(\boldsymbol{\theta}, \mathcal{D}, \mathbf{z}_{1:N}) - \log q(\mathbf{z}_{1:N} | \psi_{1:N})] \quad (10.64)$$

17

18 We can optimize this using the **variational EM** algorithm, which is a CAVI algorithm which updates
19 the ψ_n in parallel in the variational E step, and then updates $\boldsymbol{\theta}$ in the M step.

20 VEM is simpler than VBEM since in the the variational E step, we compute $q(\mathbf{z}_n | \mathbf{x}_n, \hat{\boldsymbol{\theta}})$, instead
21 of $\mathbb{E}_{\boldsymbol{\theta}}[q(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta})]$; that is, we plugin a point estimate of the model parameters, rather than averaging
22 over the parameters. For more details on VEM, see Section 6.7.6.1.

23 10.2.6 Example: VBEM for a GMM

24 Consider a standard Gaussian mixture model (GMM):

25

$$p(\mathbf{z}, \mathbf{x} | \boldsymbol{\theta}) = \prod_n \prod_k \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1})^{z_{nk}} \quad (10.65)$$

26

27 where $z_{nk} = 1$ if data point n belongs to cluster k , and $z_{nk} = 0$ otherwise. Our goal is to approximate
28 the posterior $p(\mathbf{z}, \boldsymbol{\theta} | \mathbf{x})$ under the following conjugate prior

29

$$p(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \check{\boldsymbol{\alpha}}) \prod_k \mathcal{N}(\boldsymbol{\mu}_k | \check{\boldsymbol{m}}, (\check{\boldsymbol{\Lambda}}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \check{\boldsymbol{L}}, \check{\nu}) \quad (10.66)$$

30

31 where $\boldsymbol{\Lambda}_k$ is the precision matrix for cluster k . For the mixing weights, we usually use a symmetric
32 prior, $\check{\boldsymbol{\alpha}} = \alpha_0 \mathbf{1}$.

33 The exact posterior $p(\mathbf{z}, \boldsymbol{\theta} | \mathcal{D})$ is a mixture of K^N distributions, corresponding to all possible
34 labelings \mathbf{z} , which is intractable to compute. In this section, we derive a VBEM algorithm, which
35 will approximate the posterior around a local mode. We follow the presentation of [Bis06, Sec 10.2].
36 (See also Section 10.3.5, where we discuss a different variational approximation for this model.)

1
2 **10.2.6.1 The variational posterior**

3 We will use the standard mean field approximation to the posterior: $q(\mathbf{h}) = q(\boldsymbol{\theta}) \prod_n q_n(z_n)$. At this
4 stage we have not specified the forms of the q functions; these will be determined by the form of the
5 likelihood and prior. Below we will show that the optimal forms are as follows:
6

7

$$q_n(z_n) = \text{Cat}(z_n | \mathbf{r}_n) \quad (10.67)$$

8

$$q(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \hat{\boldsymbol{\alpha}}) \prod_k \mathcal{N}(\boldsymbol{\mu}_k | \hat{\mathbf{m}}_k, (\hat{\kappa}_k \boldsymbol{\Lambda}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \hat{\mathbf{L}}_k, \hat{\nu}_k) \quad (10.68)$$

9
10 where \mathbf{r}_n are the posterior responsibilities, and the parameters with hats on them are the hyperpa-
11 rameters from the prior updated with data.
12

13
14 **10.2.6.2 Derivation of $q(\boldsymbol{\theta})$ (variational M step)**

15 Using the mean field recipe, we write down the log joint, and take expectations over \mathbf{z} :
16

17

$$\begin{aligned} \log q(\boldsymbol{\theta}) &= \log p(\boldsymbol{\pi}) + \sum_k \log p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) + \sum_n \mathbb{E}_{q(z_n)} [\log p(\mathbf{z}_n | \boldsymbol{\pi})] \\ &\quad + \sum_k \sum_n \mathbb{E}_{q(z_n)} [z_{nk}] \log \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}) + \text{const} \end{aligned} \quad (10.69)$$

18 where $q(\mathbf{z}_n) = \text{Cat}(z_n | \mathbf{r}_n)$, where \mathbf{r}_n is the variational posterior distribution over states for data case
19 n .

20 We see that the variational posterior factorizes into the form
21

22

$$q(\boldsymbol{\theta}) = q(\boldsymbol{\pi}) \prod_k q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) \quad (10.70)$$

23 For the $\boldsymbol{\pi}$ term, we have
24

25

$$\log q(\boldsymbol{\pi}) = (\alpha_0 - 1) \sum_k \log \pi_k + \sum_k \sum_n r_{nk} \log \pi_k + \text{const} \quad (10.71)$$

26 Exponentiating, we recognize this as a Dirichlet distribution:
27

28

$$q(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi} | \hat{\boldsymbol{\alpha}}) \quad (10.72)$$

29

$$\hat{\alpha}_k = \alpha_0 + N_k \quad (10.73)$$

30

$$N_k = \sum_n r_{nk} \quad (10.74)$$

31

For the $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ terms, we have

$$q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = \mathcal{N}(\boldsymbol{\mu}_k | \widehat{\boldsymbol{m}}_k, (\widehat{\kappa}_k \boldsymbol{\Lambda}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \widehat{\mathbf{L}}_k, \widehat{\nu}_k) \quad (10.75)$$

$$\widehat{\kappa}_k = \check{\kappa} + N_k \quad (10.76)$$

$$\widehat{\boldsymbol{m}}_k = (\check{\kappa} \check{\boldsymbol{m}} + N_k \bar{\boldsymbol{x}}_k) / \widehat{\kappa}_k \quad (10.77)$$

$$\widehat{\mathbf{L}}_k^{-1} = \check{\mathbf{L}}^{-1} + N_k \mathbf{S}_k + \frac{\check{\kappa} N_k}{\check{\kappa} + N_k} (\bar{\boldsymbol{x}}_k - \check{\boldsymbol{m}})(\bar{\boldsymbol{x}}_k - \check{\boldsymbol{m}})^\top \quad (10.78)$$

$$\widehat{\nu}_k = \check{\nu} + N_k \quad (10.79)$$

$$\bar{\boldsymbol{x}}_k = \frac{1}{N_k} \sum_n r_{nk} \boldsymbol{x}_n \quad (10.80)$$

$$\mathbf{S}_k = \frac{1}{N_k} \sum_n r_{nk} (\boldsymbol{x}_n - \bar{\boldsymbol{x}}_k)(\boldsymbol{x}_n - \bar{\boldsymbol{x}}_k)^\top \quad (10.81)$$

This is very similar to the M step for MAP estimation for GMMs, except here we are computing the parameters of the posterior for $\boldsymbol{\theta}$ rather than a point estimate $\hat{\boldsymbol{\theta}}$.

10.2.6.3 Derivation of $q(z)$ (variational E step)

The variational E step is more interesting, since it is quite different from the E step in regular EM, because we need to average over the parameters, rather than condition on them. In particular, we have

$$\begin{aligned} \log q(\boldsymbol{z}) &= \sum_n \sum_k z_{nk} \left(\mathbb{E}_{q(\boldsymbol{\pi})} [\log \pi_k] + \frac{1}{2} \mathbb{E}_{q(\boldsymbol{\Lambda}_k)} [\log |\boldsymbol{\Lambda}_k|] - \frac{D}{2} \log(2\pi) \right. \\ &\quad \left. - \frac{1}{2} \mathbb{E}_{q(\boldsymbol{\theta})} [(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Lambda}_k (\boldsymbol{x}_n - \boldsymbol{\mu}_k)] \right) + \text{const} \end{aligned} \quad (10.82)$$

Using the fact that $q(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi} | \widehat{\boldsymbol{\alpha}})$, one can show that

$$\exp(\mathbb{E}_{q(\boldsymbol{\pi})} [\log \pi_k]) = \frac{\exp(\psi(\widehat{\alpha}_k))}{\exp(\psi(\sum_{k'} \widehat{\alpha}_{k'}))} \triangleq \tilde{\pi}_k \quad (10.83)$$

where ψ is the **digamma function**

$$\psi(x) = \frac{d}{dx} \log \Gamma(x) \quad (10.84)$$

This takes care of the first term.

For the second term, one can show

$$\mathbb{E}_{q(\boldsymbol{\Lambda}_k)} [\log |\boldsymbol{\Lambda}_k|] = \sum_{j=1}^D \psi\left(\frac{\widehat{\nu}_k + 1 - j}{2}\right) + D \log 2 + \log |\widehat{\mathbf{L}}_k| \quad (10.85)$$

Finally, for the expected value of the quadratic form, one can show

$$\mathbb{E}_{q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)} [(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Lambda}_k (\boldsymbol{x}_n - \boldsymbol{\mu}_k)] = D \widehat{\kappa}_k^{-1} + \widehat{\nu}_k (\boldsymbol{x}_n - \widehat{\boldsymbol{m}}_k)^\top \widehat{\mathbf{L}}_k (\boldsymbol{x}_n - \widehat{\boldsymbol{m}}_k) \triangleq \tilde{\Lambda}_k \quad (10.86)$$

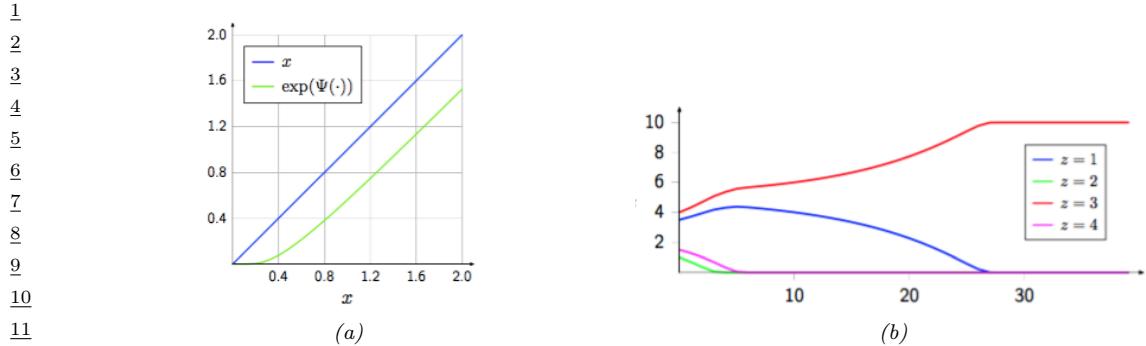


Figure 10.6: (a) We plot $\exp(\psi(x))$ vs x (green line). We see that it performs a form of shrinkage, so that small values get set to zero. *(b)* We plot N_k vs time for 4 different states (z values), starting from random initial values. We perform a series of VBEM updates, ignoring the likelihood term. We see that states that initially had higher counts get reinforced, and sparsely populated states get killed off. From [LK07]. Used with kind permission of Percy Liang.

Thus we get that the posterior responsibility of cluster k for datapoint n is

$$r_{nk} \propto \tilde{\pi}_k \tilde{\Lambda}_k^{\frac{1}{2}} \exp\left(-\frac{D}{2 \tilde{\kappa}_k} - \frac{\tilde{\nu}_k}{2} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^T \hat{\boldsymbol{\Lambda}}_k (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)\right) \quad (10.87)$$

Compare this to the expression used in regular EM:

$$r_{nk}^{EM} \propto \hat{\pi}_k |\hat{\boldsymbol{\Lambda}}_k|^{\frac{1}{2}} \exp\left(-\frac{1}{2} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^T \hat{\boldsymbol{\Lambda}}_k (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)\right) \quad (10.88)$$

where $\hat{\pi}_k$ is the MAP estimate for π_k . The significance of this difference is discussed in Section 10.2.6.4.

10.2.6.4 Automatic sparsity inducing effects of VBEM

In regular EM, the E step has the form given in Equation (10.88), whereas in VBEM, the E step has the form given in Equation (10.87). Although they look similar, they differ in an important way. To understand this, let us ignore the likelihood term, and just focus on the prior. Then we have

$$r_{nk}^{VB} = \tilde{\pi}_k = \frac{\exp(\psi(\hat{\alpha}_k))}{\exp(\psi(\sum_{k'} \hat{\alpha}_{k'}))} \quad (10.89)$$

$$r_{nk}^{EM} = \hat{\pi}_k = \frac{\hat{\alpha}_k - 1}{\sum_{k'} (\hat{\alpha}_{k'} - 1)} \quad (10.90)$$

where $\hat{\alpha}_k = \alpha_0 + N_k$, and $N_k = \sum_n r_{nk}$ is the expected number of assignments to cluster k .

We know from Figure 2.11 that using $\alpha_0 \ll 1$ causes π to be sparse, which will encourage \mathbf{r}_n to be sparse, which will “kill off” unnecessary mixture components (i.e., ones for which $N_k \ll N$, meaning very few data points are assigned to cluster k). To encourage this sparsity promoting effect, let us

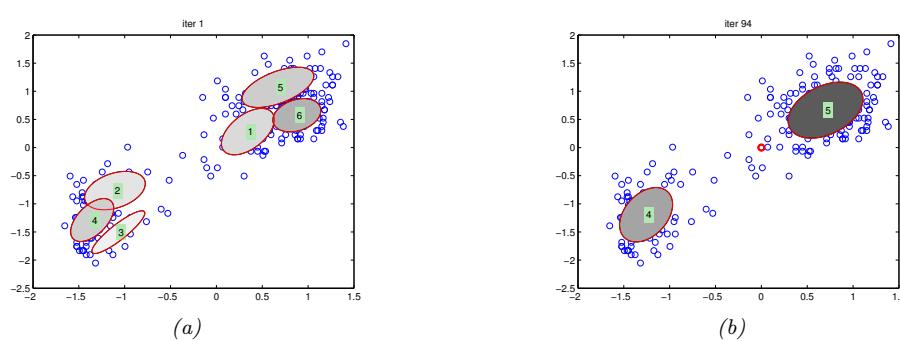


Figure 10.7: We visualize the posterior mean parameters at various stages of the VBEM algorithm applied to a mixture of Gaussians model on the Old Faithful data. Shading intensity is proportional to the mixing weight. We initialize with K-means and use $\alpha_0 = 0.001$ as the Dirichlet hyper-parameter. (The red dot on the right panel represents all the unused mixture components, which collapse to the prior at 0.) Adapted from Figure 10.6 of [Bis06]. Generated by `variational_mixture_gaussians_demo.py`.

set $\alpha_0 = 0$. In this case, the updated parameters for the mixture weights are given by the following:

$$\tilde{\pi}_k = \frac{\exp(\psi(N_k))}{\exp(\psi(\sum_{k'} N_{k'}))} \quad (10.91)$$

$$\hat{\pi}_k = \frac{N_k - 1}{\sum_{k'}(N_{k'} - 1)} \quad (10.92)$$

Now consider a cluster which has no assigned data, so $N_k = 0$. In regular EM, $\hat{\pi}_k$ might end up negative, as pointed out in [FJ02]. (This will not occur if we use maximum likelihood training, which corresponds to $\alpha_0 = 1$, but this will not induce any sparsity, either.) This problem does not arise in VBEM, since we use the digamma function, which is always positive, as shown in Figure 10.6(a).

More interestingly, let us consider the effect of these updates on clusters that have unequal, but non-zero, number of assignments. Suppose we start with a random assignment of counts to 4 clusters, and iterate the VBEM algorithm, ignoring the contribution from the likelihood for simplicity. Figure 10.6(b) shows how the counts N_k evolve over time. We notice that clusters that started out with small counts end up with zero counts, and clusters that started out with large counts end up with even larger counts. In other words, the initially popular clusters get more and more members. This is called the **rich get richer** phenomenon; we will encounter it again in Section 33.2.4, when we discuss Dirichlet process mixture models.

The reason for this effect is shown in Figure 10.6(a): we see that $\exp(\psi(N_k)) < N_k$, and is zero if N_k is sufficiently small, similar to the soft-thresholding behavior induced by ℓ_1 -regularization (see Section 15.2.5). Importantly, this effect of reducing N_k is greater on clusters with small counts, so it acts like a regressive tax, punishing the poor more.

We now demonstrate this automatic pruning method on a real example. We fit a mixture of 6 Gaussians to the Old Faithful dataset, using $\alpha_0 = 0.001$. Since the data only really “needs” 2 clusters, the remaining 4 get “killed off”, as shown in Figure 10.7. In Figure 10.8, we plot the initial and final values of α_k : we see that $\hat{\alpha}_k = 0$ for all but two of the components k .

Thus we see that VBEM for GMMs with a sparse Dirichlet prior provides an efficient way to choose

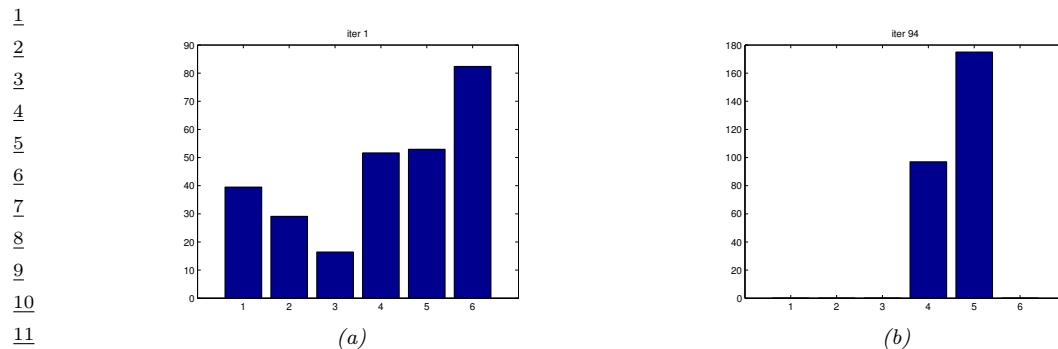


Figure 10.8: We visualize the posterior values of α_k for the model in Figure 10.7 after the first and last iteration of the algorithm. We see that unnecessary components get “killed off”. (Interestingly, the initially large cluster 6 gets “replaced” by cluster 5.) Generated by `variational_mixture_gaussians_demo.py`.

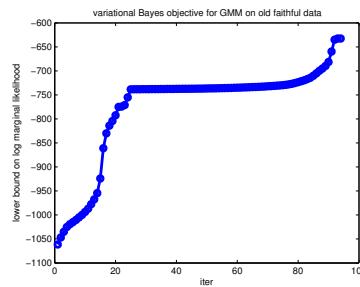


Figure 10.9: Lower bound vs iterations for the VB algorithm in Figure 10.7. The steep parts of the curve correspond to places where the algorithm figures out that it can increase the bound by “killing off” unnecessary mixture components, as described in Section 10.2.6.6. The plateaus correspond to slowly moving the clusters around. Generated by `variational_mixture_gaussians_demo.py`.

the number of clusters. Similar techniques can be used to choose the number of states in an HMM and other latent variable models. However, this **variational pruning effect** (also called **posterior collapse**), is not always desirable, since it can cause the model to “ignore” the latent variables \mathbf{z} if the likelihood function $p(\mathbf{x}|\mathbf{z})$ is sufficiently powerful. We discuss this more in Section 22.4.

10.2.6.5 Lower bound on the marginal likelihood

The VBEM algorithm is maximizing the following lower bound

$$\mathcal{L} = \sum_{\mathbf{z}} \int d\boldsymbol{\theta} q(\mathbf{z}, \boldsymbol{\theta}) \log \frac{p(\mathbf{x}, \mathbf{z}, \boldsymbol{\theta})}{q(\mathbf{z}, \boldsymbol{\theta})} \leq \log p(\mathbf{x}) \quad (10.93)$$

This quantity increases monotonically with each iteration, as shown in Figure 10.9.

1 **10.2.6.6 Model selection using VBEM**

3 Section 10.2.6.4 discusses a way to choose K automatically, during model fitting, by “killing off”
4 unneeded clusters. An alternative approach is to fit several models, and then to use the variational
5 lower bound to the log marginal likelihood, $\mathcal{L}(K) \leq \log p(\mathcal{D}|K)$, to approximate $p(K|\mathcal{D})$:
6

7
$$p(K|\mathcal{D}) = \frac{e^{\mathcal{L}(K)}}{\sum_{K'} e^{\mathcal{L}(K')}} \quad (10.94)$$

8

9 It is shown in [BG06] that the VB approximation to the marginal likelihood is more accurate than
10 BIC [BG06]. However, the lower bound needs to be modified somewhat to take into account the lack
11 of identifiability of the parameters. In particular, although VB will approximate the volume occupied
12 by the parameter posterior, it will only do so around one of the local modes. With K components,
13 there are $K!$ equivalent modes, which differ merely by permuting the labels. Therefore we should use
14 $\log p(\mathcal{D}|K) \approx \mathcal{L}(K) + \log(K!)$.
15

16 **10.2.7 Variational message passing (VMP)**

18 In this section, we describe the CAVI algorithm for a generic model in which each complete conditional,
19 $p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x})$, is in the exponential family, i.e.,
20

21
$$p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x}) = h(\mathbf{z}_j) \exp[\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})^\top \mathcal{T}(\mathbf{z}_j) - A_j(\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x}))] \quad (10.95)$$

22 where $\mathcal{T}(\mathbf{z}_j)$ is the vector of sufficient statistics, $\boldsymbol{\eta}_j$ are the natural parameters, A_j is the log partition
23 function, and $h(\mathbf{z}_j)$ is the base distribution. This assumption holds if the prior $p(\mathbf{z}_j)$ is conjugate to
24 the likelihood, $p(\mathbf{z}_{-j}, \mathbf{x}|\mathbf{z}_j)$.
25

If Equation (10.95) holds, the mean field update node j becomes

27
$$q_j(\mathbf{z}_j) \propto \exp[\mathbb{E}[\log p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x})]] \quad (10.96)$$

28
$$= \exp\left[\log h(\mathbf{z}_j) + \mathbb{E}\left[\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})\right]^\top \mathcal{T}(\mathbf{z}_j) - \mathbb{E}[A_j(\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x}))]\right] \quad (10.97)$$

30
$$\propto h(\mathbf{z}_j) \exp\left[\mathbb{E}\left[\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})\right]^\top \mathcal{T}(\mathbf{z}_j)\right] \quad (10.98)$$

32 Thus we update the local natural parameters using the expected values of the other nodes. These
33 become the new variational parameters:

34
$$\boldsymbol{\psi}_j = \mathbb{E}[\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})] \quad (10.99)$$

36 We can generalize the above approach to work with any model where each full conditional is
37 conjugate. The resulting algorithm is known as **variational message passing** or **VMP** [WB05]
38 that works for any directed graphical model. VMP is similar to belief propagation (Section 9.2): at
39 each iteration, each node collects all the messages from its parents, and all the messages from its
40 children (which might require the children to get messages from their co-parents), and combines them
41 to compute the expected value of the node’s sufficient statistics. The messages that are sent are the
42 expected sufficient statistics of a node, rather than just a discrete or Gaussian distribution (as in BP).
43 Several software libraries have implemented this framework (see e.g., [Win; Min+18; Lut16; Wan17]).
44

44 VMP can be extended to the case where each full conditional is conditionally conjugate using the
45 CVI framework in Section 10.3.8. See also [ABV21], where they use local Laplace approximations to
46 intractable factors inside of a message passing framework.
47

1 **10.2.8 Autoconj**

2 The VMP method requires the user to manually specify a graphical model; the corresponding node
3 update equations are then computed for each node using a lookup table, for each possible combination
4 of node types. It is possible to automatically derive these update equations for any conditionally
5 conjugate directed graphical model using a technique called **autoconj** [HJT18]. This is analogous to
6 the use of automatic differentiation (autodiff) to derive the gradient for any differentiable function.
7 (Note that autoconj uses autodiff internally.) The resulting full conditionals can be used for CAVI,
8 and also for Gibbs sampling (Section 12.3).

9

10

11

12

13

14 **10.3 Fixed-form VI**

15

16 In the mean field method of Section 10.2, all that we assumed about the variational posterior was
17 that it factorized across (groups of) variables, $q(\mathbf{z}|\boldsymbol{\psi}) = \prod_{j=1}^J q_j(z_j)$. The functional form of each
18 q_j follows automatically from the form of the model. In this section, we take a different approach,
19 in which we assume the a specific functional form for q , such as a Gaussian. That is, our goal is to
20 optimize the following lower bound:

21

22

23

$$\underline{24} \quad L(\boldsymbol{\psi}|\mathcal{D}, \boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{z}) p_{\boldsymbol{\theta}}(\mathcal{D}|\mathbf{z})}{q_{\boldsymbol{\psi}}(\mathbf{z})} \right] = \mathbb{E}_{q_{\boldsymbol{\psi}}} [\ell_{\boldsymbol{\psi}}(\mathbf{z})] \quad (10.100)$$

25

26

27

28

29

30

$$\underline{31} \quad \ell_{\boldsymbol{\psi}}(\mathbf{z}) = \log p_{\boldsymbol{\theta}}(\mathbf{z}, \mathcal{D}) - \log q_{\boldsymbol{\psi}}(\mathbf{z}) \quad (10.101)$$

32

33

We will use gradient based methods to optimize this.

34

35

36

37

38 **10.3.1 Black-box variational inference**

39

40 In this section, we assume that we can evaluate $\ell_{\boldsymbol{\psi}}(\mathbf{z})$ pointwise, but we do not assume we can take
41 gradients of this function. (For example, \mathbf{z} may contain discrete variables.) We are thus treating the
42 model as a “blackbox”. Hence this approach is called **black box variational inference** or **BBVI**

43 [RGB14; ASD20].

44 To estimate the gradient of the ELBO, we will use the **score function estimator**, also called
45 the **REINFORCE** estimator (Section 6.6.3). To derive this, we the fact that $\nabla \log q = \frac{\nabla q}{q}$ to
46 conclude that $\nabla q = q \nabla \log q$. (This is called the **log derivative trick**.) We also exploit the fact

47

that $\int q(\mathbf{z})d\mathbf{z} = 1$. With this, the gradient of the ELBO can be derived as follows:

$$\nabla_{\psi} \bar{L}(\psi) = \nabla_{\psi} \int q_{\psi}(\mathbf{z}) \log \frac{p_{\theta}(\mathbf{z}, \mathcal{D})}{q_{\psi}(\mathbf{z})} d\mathbf{z} \quad (10.102)$$

$$= \int [\nabla_{\psi} q_{\psi}(\mathbf{z})] \left[\log \frac{p_{\theta}(\mathbf{z}, \mathcal{D})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} + \int [q_{\psi}(\mathbf{z})] \left[\nabla_{\psi} \log \frac{p_{\theta}(\mathbf{z}, \mathcal{D})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} \quad (10.103)$$

$$= \int [\nabla_{\psi} q_{\psi}(\mathbf{z})] \left[\log \frac{p_{\theta}(\mathbf{z}) p_{\theta}(\mathcal{D}|\mathbf{z})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} - \int q_{\psi}(\mathbf{z}) \nabla_{\psi} \log q_{\psi}(\mathbf{z}) d\mathbf{z} \quad (10.104)$$

$$= \int [\nabla_{\psi} q_{\psi}(\mathbf{z})] \left[\log \frac{p_{\theta}(\mathbf{z}) p_{\theta}(\mathcal{D}|\mathbf{z})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} - \int \nabla_{\psi} q_{\psi}(\mathbf{z}) d\mathbf{z} \quad (10.105)$$

$$= \int [\nabla_{\psi} q_{\psi}(\mathbf{z})] \left[\log \frac{p_{\theta}(\mathbf{z}) p_{\theta}(\mathcal{D}|\mathbf{z})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} - \nabla_{\psi} \int q_{\psi}(\mathbf{z}) d\mathbf{z} \quad (10.106)$$

$$= \int [\nabla_{\psi} q_{\psi}(\mathbf{z})] \left[\log \frac{p_{\theta}(\mathbf{z}) p_{\theta}(\mathcal{D}|\mathbf{z})}{q_{\psi}(\mathbf{z})} \right] d\mathbf{z} \quad (10.107)$$

$$= \int q_{\psi}(\mathbf{z}) \nabla_{\psi} \log q_{\psi}(\mathbf{z}) \times \ell_{\psi}(\mathbf{z}) d\mathbf{z} \quad (10.108)$$

$$= \mathbb{E}_{q_{\psi}(\mathbf{z})} [\nabla_{\psi} \log q_{\psi}(\mathbf{z}) \times \ell_{\psi}(\mathbf{z})] \quad (10.109)$$

We can compute a stochastic approximation to this gradient by sampling $\mathbf{z}_s \sim q_{\psi}(\mathbf{z})$ and then computing

$$\widehat{\nabla_{\psi} \bar{L}(\psi_t)} = \frac{1}{S} \sum_{s=1}^S \nabla_{\psi} \log q_{\psi}(\mathbf{z}_s) \times \ell_{\psi}(\mathbf{z}_s) |_{\psi=\psi_t} \quad (10.110)$$

We can pass this to any kind of gradient optimizer, such as SGD or Adam.

In practice, the variance of this estimator is quite large, so it is important to use methods such as **control variates** (Section 6.6.3.1). To see how this works, consider the naive gradient estimator in Equation (10.110), which for the i 'th component we can write as

$$\widehat{\nabla_{\psi_i} \bar{L}(\psi_t)}^{\text{naive}} = \frac{1}{S} \sum_{s=1}^S \tilde{g}_i(\mathbf{z}_s) \quad (10.111)$$

$$\tilde{g}_i(\mathbf{z}_s) = g_i(\mathbf{z}_s) \times \ell_{\psi}(\mathbf{z}_s) \quad (10.112)$$

$$g_i(\mathbf{z}_s) = \nabla_{\psi_i} \log q_{\psi}(\mathbf{z}_s) \quad (10.113)$$

The control variate version of this can be obtained by replacing $\tilde{g}_i(\mathbf{z}_s)$ with

$$\tilde{g}_i^{CV}(\mathbf{z}) = \tilde{g}_i(\mathbf{z}) + c_i (\mathbb{E}[b_i(\mathbf{z})] - b_i(\mathbf{z})) \quad (10.114)$$

where $b_i(\mathbf{z})$ is a baseline function and c_i is some constant, to be specified below. A convenient baseline is the score function, $b_i(\mathbf{z}) = \nabla_{\psi_i} \log q_{\psi_i}(\mathbf{z})$, since this is correlated with $\tilde{g}_i(\mathbf{z})$, and has the property that $\mathbb{E}[b_i(\mathbf{z})] = \mathbf{0}$, since the expected value of the score function is zero, as we showed in Equation (2.223). Hence

$$\tilde{g}_i^{CV}(\mathbf{z}) = \tilde{g}_i(\mathbf{z}) - c_i g_i(\mathbf{z}) = g_i(\mathbf{z})(\ell_{\psi}(\mathbf{z}) - c_i) \quad (10.115)$$

1 so the CV estimator is given by
2

$$\nabla_{\psi_i} \widehat{\mathbb{L}}(\psi_t)^{\text{cv}} = \frac{1}{S} \sum_{s=1}^S g_i(z_s) \times (\ell_{\psi}(z_s) - c_i) \quad (10.116)$$

6 One can show that the optimal c_i that minimizes the variance of the CV estimator is
7

$$c_i = \frac{\text{Cov}[g_i(z)\ell_{\psi}(z), g_i(z)]}{\mathbb{V}[g_i(z)]} \quad (10.117)$$

10 which can be estimated by sampling $z \sim q_{\psi}(z)$. Thus the overall algorithm is as shown in Algorithm 6.
11

12 **Algorithm 6:** Blackbox VI with control variates

13 1 Initialize ψ_0
14 2 $z_s \sim q_{\psi_0}(z)$, $s = 1 : S$
15 3 $h_{\psi_0}(z_s) = \log p_{\theta}(z_s, \mathcal{D}) - \log q_{\psi_0}(z_s)$, $s = 1 : S$
16 4 $\mathbf{g}_0 = \frac{1}{S} \sum_{s=1}^S [\nabla_{\psi} \log q_{\psi_0}(z_s)] h_{\psi_0}(z_s)$
17 5 Compute \mathbf{c}_1 using Equation (10.117) applied to z_s
18 6 **for** $t = 1 : T$ **do**
19 7 $z_s \sim q_{\psi_t}(z)$, $s = 1 : S$
20 8 $h_{\psi_t}(z_s) = \log p_{\theta}(z_s, \mathcal{D}) - \log q_{\psi_t}(z_s)$, $s = 1 : S$
21 9 $\mathbf{g}_t = \frac{1}{S} \sum_{s=1}^S [\nabla_{\psi} \log q_{\psi_t}(z_s)] \odot (h_{\psi_t}(z_s) - \mathbf{c}_t)$
22 10 Compute \mathbf{c}_{t+1} using Equation (10.117) applied to z_s
23 11 $\psi_t = \text{gradient-update}(\psi_{t-1}, \mathbf{g}_t)$
24

26 We can stop the algorithm when the lower bound stops increasing. We can compute a stochastic
27 approximation to the lower bound using
28

$$\hat{\mathbb{L}}(\psi) = \frac{1}{S} \sum_{s=1}^S \ell_{\psi}(z_s) \quad (10.118)$$

32 where $z_s \sim q_{\psi}(z)$. To smooth out the noise, we can use a running average over the last w observations
33 to get

$$\bar{\mathbb{L}}(\psi_t) = \frac{1}{w} \sum_{k=1}^w \hat{\mathbb{L}}(\psi_{t-k+1}) \quad (10.119)$$

37 If the moving average does not improve after P consecutive iterations, we declare convergence, where
38 P is the **patience** parameter. Typical values are $P = 20$ and $w = 20$ [TND21].
39

40 **10.3.2 Stochastic variational inference**
41

42 Suppose z are the latent variables, and the observed variables are a set of iid observations, $\mathcal{D} = \{x_n : n = 1 : N\}$. In this case, we have
43

$$\ell_{\psi}(z_s) = \sum_{n=1}^N \log p_{\theta}(x_n | z_s) + \log p_{\theta}(z_s) - \log q_{\psi}(z_s) \quad (10.120)$$

If N is large, we can compute an unbiased minibatch approximation to this expression as follows:

$$\hat{\ell}_{\psi}(\mathbf{z}_s) = \left[\frac{N}{B} \sum_{b=1}^B \log p_{\theta}(\mathbf{x}_b | \mathbf{z}_s) \right] + \log p_{\theta}(\mathbf{z}_s) - \log q_{\psi}(\mathbf{z}_s) \quad (10.121)$$

This is called **stochastic variational inference** or **SVI** [Hof+13].

We can combine this with the above stochastic gradient approximation of the ELBO to get the following **doubly stochastic** approximation [TLG14]:

$$\widehat{\nabla_{\psi} \mathbb{L}(\psi_t)} = \frac{1}{S} \sum_{s=1}^S \nabla_{\psi} \log q_{\psi}(\mathbf{z}_s) \times \hat{\ell}_{\psi}(\mathbf{z}_s) |_{\psi=\psi_t} \quad (10.122)$$

Of course, this can be combined with control variates.

10.3.3 Reparameterization VI

In this section, we exploit the **reparameterization trick** from Section 6.6.4 to get a lower variance estimator for the gradient. This assumes that $\ell_{\psi}(\mathbf{z})$ is a differentiable function of \mathbf{z} . It also assumes that we can sample $\mathbf{z} \sim q_{\psi}(\mathbf{z})$ by first sampling a noise term $\epsilon \sim q_0(\epsilon)$, and then transforming it to compute the latent random variables $\mathbf{z} = r(\psi, \epsilon)$. In this case, the ELBO becomes

$$\mathbb{L}(\psi) = \mathbb{E}_{q_0(\epsilon)} [\ell_{\psi}(r(\psi, \epsilon))] = \mathbb{E}_{q_0(\epsilon)} [\log p_{\theta}(r(\psi, \epsilon), \mathcal{D}) - \log q_{\psi}(r(\psi, \epsilon))] \quad (10.123)$$

Since the sampling distribution $q_0(\epsilon)$ is independent of the variational parameters ψ , we can push the gradient operator inside the expectation, and thus we can estimate the gradient using standard automatic differentiation methods, as shown in Algorithm 7. This is called **reparameterized VI** or **RVI**, and has provably lower variance than BBVI in certain cases [Xu+19].

Algorithm 7: Estimate of ELBO gradient

```

1 def elbo(\psi):
2     \epsilon \sim q_0(\epsilon)
3     z = r(\psi, \epsilon)
4     return log p_{\theta}(z, \mathcal{D}) - q_{\psi}(z | \mathcal{D})
5 def elbo-grad(\psi):
6     return grad(elbo(\psi))

```

10.3.3.1 “Sticking the landing” estimator

Applying the results from Section 6.6.4.2, we can derive the gradient estimate of the reparameterized ELBO, for a single Monte Carlo sample, as follows:

$$\nabla_{\psi} \mathbb{L}(\psi, \epsilon) = \nabla_{\psi} [\log p(\mathbf{z}, \mathcal{D}) - \log q_{\psi}(\mathbf{z} | \mathcal{D})] \quad (10.124)$$

$$= \underbrace{\nabla_{\mathbf{z}} [\log p(\mathbf{z} | \mathcal{D}) - \log q_{\psi}(\mathbf{z} | \mathcal{D})]}_{\text{path derivative}} \mathbf{J} - \underbrace{\nabla_{\psi} \log q_{\psi}(\mathbf{z} | \mathcal{D})}_{\text{score function}} \quad (10.125)$$

where $z = r(\psi, \epsilon)$ and $\mathbf{J} = \nabla_{\psi}r(\psi, \epsilon)$ is the Jacobian matrix of the noise transformation.

The first term is the indirect effect of ψ on the objective via the generated samples \mathbf{z} . The second term is the direct effect of ψ on the objective. The second term is zero in expectation since it is the score function (see Equation (2.223)), but it may be non-zero for a finite number of samples, even if $q_\psi(\mathbf{z}|\mathcal{D}) = p(\mathbf{z}|\mathcal{D})$ is the true posterior. In a paper called “**sticking the landing**”, [RWD17] propose to drop the second term to create a lower variance estimator.² In practice, this means we compute the gradient using Algorithm 8 instead of Algorithm 7.³

Algorithm 8: “Sticking the landing” estimator of ELBO gradient

```

11 1 def elbo-pd( $\psi$ ):
12 2      $\epsilon \sim q_0(\epsilon)$ 
13 3      $z = r(\psi, \epsilon)$ 
14 4      $\psi' = \text{stop-gradient}(\psi)$ 
15 5     return  $\log p_{\theta}(z, \mathcal{D}) - q_{\psi'}(z|\mathcal{D})$ 
16 6 def elbo-grad-pd( $\psi$ ):
17 7     return grad(elbo-pd( $\psi$ ))

```

20 Note that the STL estimator is not always better than the “standard” estimator. In [GD20], they
 21 propose to use a weighted combination of estimators, where the weights are optimized so as to reduce
 22 variance for a fixed amount of compute.

24 10.3.4 Gaussian VI

²⁶The most widely used RVI approximation is when $q_{\psi}(z)$ is a Gaussian, where $\psi = (\mu, \Sigma)$. Following
²⁷[TND21], we call this **Gaussian VI**. We give some examples of this below.

10.3.4.1 Gaussian posterior: Cholesky decomposition

³⁰ In this section, we represent the covariance using its Cholesky decomposition, $\Sigma = \mathbf{L}\mathbf{L}^\top$, where
³¹ $\mathbf{L} = \text{tril}(\mathbf{l})$ is a lower triangular matrix, as in [TLG14; TN18]. The variational parameters are
³² $\psi = (\boldsymbol{\mu}, \mathbf{l})$. The noise transformation has the form

$$^{34} \quad z \sim N(\mu, \Sigma) \iff z = \mu + L\epsilon \quad (10.126)$$

where $\epsilon \sim \mathcal{N}(0, I)$, so $r(\psi, \epsilon) \equiv \mu + L\epsilon$.

We have $\nabla_{\mu} r(\psi, \epsilon) = \mathbf{I}$, so the gradient of the ELBO wrt μ has the form

$$\nabla_{\psi} \mathbb{E}_{\psi}[\ell(\psi)] = \mathbb{E}_{\pi^*(z)}[\nabla_z \ell_{\psi}(z)] \quad (10.127)$$

To compute the gradient wrt \mathbf{L} , we need some notation. For a $d \times d$ matrix \mathbf{A} , let $\text{vec}(\mathbf{A})$ be the d^2 -vector obtained by stacking the columns of \mathbf{A} , let $\text{vech}(\mathbf{A})$ be the $\frac{1}{2}d(d+1)$ -vector obtained by

⁴³ 2. The expression “to stick a landing” means to land firmly on one’s feet after performing a gymnastics move. In the current context, the analogy is this: if the variational posterior is optimal, so $q_\psi(z|\mathcal{D}) = p(z|\mathcal{D})$, then we want our objective to be 0, and not to “wobble” with Monte Carlo noise.

45 3. The difference is that the path derivative version ignores the score function. This can be achieved by using
 46 $\log q_{\psi'}(\mathbf{z}|\mathcal{D})$, where ψ' is a “disconnected” copy of ψ that does not affect the gradient.

stacking the columns of the lower triangular part of \mathbf{A} , and let $\mathbf{A} \otimes \mathbf{B}$ be the Kronecker product of \mathbf{A} and \mathbf{B} . For any matrices \mathbf{A} , \mathbf{B} and \mathbf{X} of suitable sizes, we have that

$$\text{vec}(\mathbf{AXB}) = (\mathbf{B}^T \otimes \mathbf{A})\text{vec}(\mathbf{X}) \quad (10.128)$$

Hence $\mathbf{L}\boldsymbol{\epsilon} = \text{vec}(\mathbf{I} \mathbf{L} \boldsymbol{\epsilon}) = (\boldsymbol{\epsilon}^T \otimes \mathbf{I})\text{vec}(\mathbf{L})$ and $\nabla_{\text{vec}(\mathbf{L})} r(\boldsymbol{\psi}, \boldsymbol{\epsilon}) = \boldsymbol{\epsilon}^T \otimes \mathbf{I}$ so

$$\nabla_{\text{vec}(\mathbf{L})} \bar{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [\nabla_{\text{vec}(\mathbf{L})} r(\boldsymbol{\psi}, \boldsymbol{\epsilon})^T \nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z})] \quad (10.129)$$

$$= \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [(\boldsymbol{\epsilon} \otimes \mathbf{I}) \nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z})] \quad (10.130)$$

$$= \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [\text{vec}(\nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z}) \boldsymbol{\epsilon}^T)] \quad (10.131)$$

where $\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}$. Hence

$$\nabla_{\text{vech}(\mathbf{L})} \bar{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [\text{vech}(\nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z}) \boldsymbol{\epsilon}^T)] \quad (10.132)$$

Thus the overall algorithm is as shown in Algorithm 9. The main requirement is that we have a way to compute

$$\nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z}) = \nabla_{\mathbf{z}} \log p(\mathbf{z}, \mathcal{D}) - \nabla_{\mathbf{z}} \log q_{\boldsymbol{\psi}}(\mathbf{z}) \quad (10.133)$$

where $\nabla_{\mathbf{z}} \log p(\mathbf{z}, \mathcal{D})$ is the model-specific gradient of the log joint, and

$$\nabla_{\mathbf{z}} \log q_{\boldsymbol{\psi}}(\mathbf{z}) = -\boldsymbol{\Sigma}^{-1}(\mathbf{z} - \boldsymbol{\mu}) \quad (10.134)$$

Algorithm 9: Reparameterized VI with posterior $q(\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T)$

```

1 Initialize  $\boldsymbol{\psi}_0 = (\boldsymbol{\mu}_0, \mathbf{L}_0)$ 
2 for  $t = 1 : T$  do
3    $\boldsymbol{\epsilon}_s \sim q_0(\boldsymbol{\epsilon}), s = 1 : S$ 
4    $\mathbf{z}_s = \boldsymbol{\mu}_t + \mathbf{L}_t \boldsymbol{\epsilon}_s, s = 1 : S$ 
5    $\nabla_{\boldsymbol{\mu}} \bar{L}(\boldsymbol{\psi}_t) = \frac{1}{S} \sum_{s=1}^S \nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z}_s) |_{\boldsymbol{\psi}=\boldsymbol{\psi}_t}$ 
6    $\nabla_{\text{vec}(\mathbf{L})} \bar{L}(\boldsymbol{\psi}_t) = \frac{1}{S} \sum_{s=1}^S \text{vech}(\nabla_{\mathbf{z}} \ell_{\boldsymbol{\psi}}(\mathbf{z}_s) \boldsymbol{\epsilon}_s^T) |_{\boldsymbol{\psi}=\boldsymbol{\psi}_t}$ 
7    $\mathbf{g}_t = [\nabla_{\boldsymbol{\mu}} \bar{L}(\boldsymbol{\psi}_t), \nabla_{\text{vec}(\mathbf{L})} \bar{L}(\boldsymbol{\psi}_t)]$ 
8    $\boldsymbol{\psi}_t = \text{gradient-update}(\boldsymbol{\psi}_{t-1}, \mathbf{g}_t)$ 

```

We give an example of this applied to a simple 2d binary logistic regression problem in Figure 10.10b. We see that the predictive distribution from the VI posterior is similar to that produced by MCMC.

10.3.4.2 Gaussian posterior: Low-rank plus diagonal

In high dimensions, an efficient alternative to using a Cholesky decomposition is the factor decomposition

$$\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^T + \mathbf{C}^2 \quad (10.135)$$

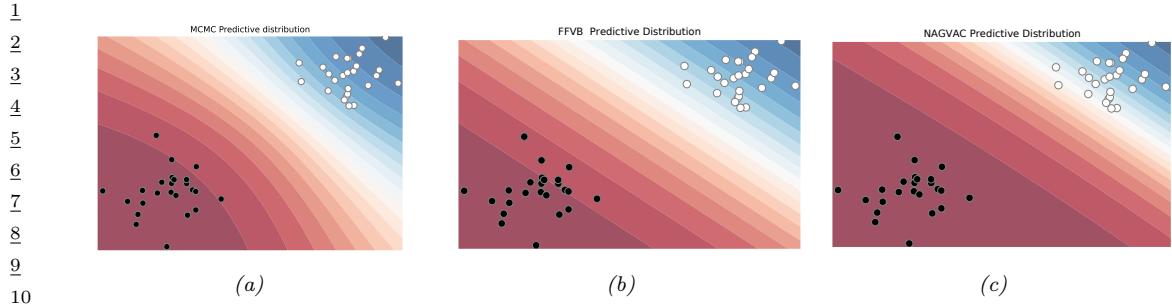


Figure 10.10: Bayesian inference applied to a 2d binary logistic regression problem, $p(y = 1|\mathbf{x}) = \sigma(w_0 + w_1x_1 + w_2x_2)$. We show the training data and the posterior predictive produced by different methods. (a) MCMC approximation. (b) VB approximation using full covariance matrix (Cholesky decomposition). (c) VB using rank 1 approximation. Generated by `vb_gauss_biclusters_demo.py`.

15

16

17 where \mathbf{B} is the factor loading matrix of size $d \times f$, where $f \ll d$ is the number of factors, d is
18 the dimensionality of \mathbf{z} , and $\mathbf{C} = \text{diag}(c_1, \dots, c_d)$. This reduces the total number of variational
19 parameters from $d + d(d+1)/2$ to $(f+2)d$. In [ONS18], they called this approach **VAFC** for
20 Variational Approximation with Factor Covariance.

21 In the special case where $f = 1$, the covariance matrix becomes $\Sigma = \mathbf{b}\mathbf{b}^\top + \mathbf{C}^2$. In this case, it is
22 possible to compute the natural gradient (Section 6.4) of the ELBO in closed form in $O(d)$ time, as
23 shown in [Tra+20b], [Tra+20b], who call the approach **NAGVAC-1** (Natural Gradient Gaussian
24 variational approximation). This can result in much faster convergence than following the normal
25 gradient.

26 We give an example of this applied to a simple 2d binary logistic regression problem in Figure 10.10c.
27 We see that the predictive distribution from the VI posterior is similar to that produced by MCMC.

28

29 10.3.4.3 Comparing Gaussian VI and HMC on (non-conjugate) 1d linear regression

30

31 In this section, we give a comparison of HMC (Section 12.5) and stochastic Gaussian VI. We use a
32 simple example from [McE20, Sec 8.1].⁴ Here the goal is to predict (log) GDP G of various countries
33 (in the year 2000) as a function of two input variables: the ruggedness R of the country's terrain, and
34 whether the country is in Africa or not (A). Specifically, we use the following 1d regression model:

$$35 \quad y_i \sim \mathcal{N}(\mu_i, \sigma^2) \tag{10.136}$$

$$36 \quad \mu_i = \alpha + \beta^\top \mathbf{x}_i \tag{10.137}$$

$$37 \quad \alpha \sim \mathcal{N}(0, 10) \tag{10.138}$$

$$38 \quad \beta_j \sim \mathcal{N}(0, 1) \tag{10.139}$$

$$39 \quad \sigma \sim \text{Unif}(0, 10) \tag{10.140}$$

42 where $\mathbf{x}_i = (R_i, A_i, A_i \times R_i)$ are the features, and $y_i = G_i$ is the response.

43 We first use HMC, which is often considered the “gold standard” of posterior inference. The
44 resulting model fit is shown in Figure 10.11. This shows that GDP increases as a function of

45
46 4. We choose this example since it is used as the introductory example in the [Pyro tutorial](#).

47

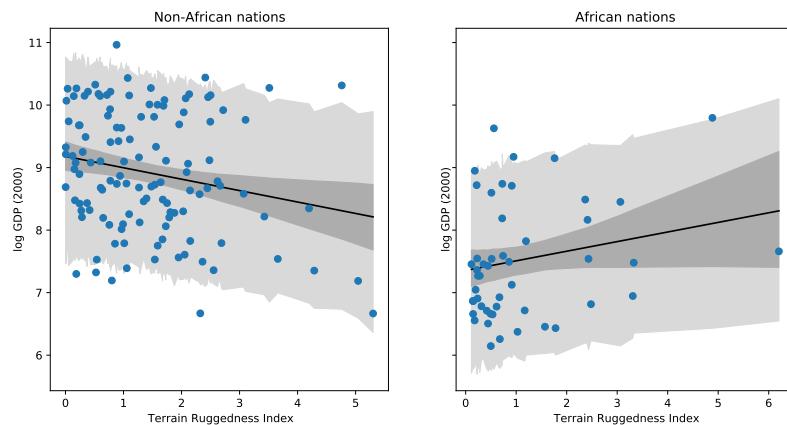


Figure 10.11: Posterior predictive distribution for the linear model applied to the Africa data. Dark shaded region is the 95% credible interval for μ_i . The light shaded region is the 95% credible interval for y_i . Adapted from Figure 8.5 of [McE20]. Generated by `linreg_bayes_svi_hmc_pyro.ipynb`.

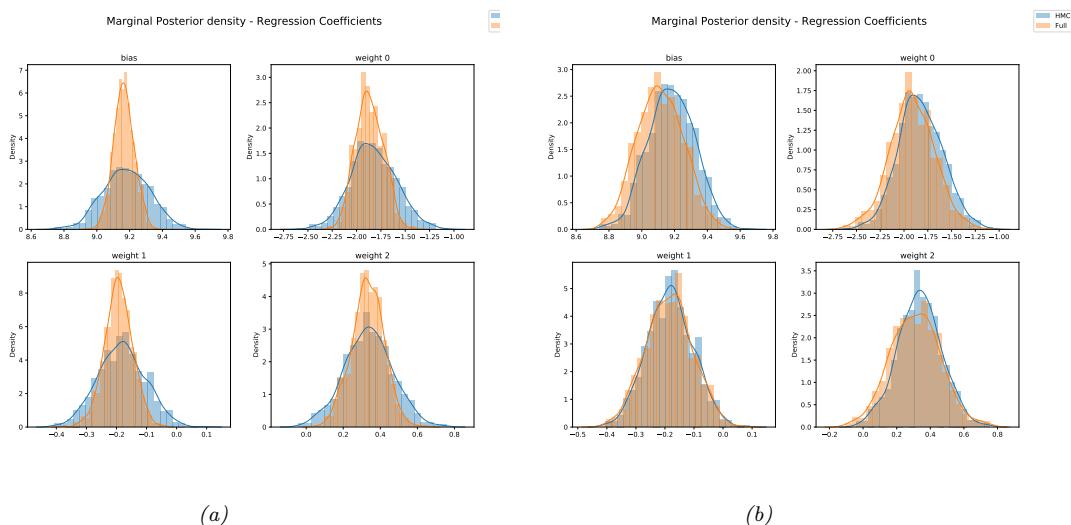


Figure 10.12: Posterior marginals for the linear model applied to the Africa data. (a) Blue is HMC, orange is Gaussian approximation with diagonal covariance. (b) Blue is HMC, orange is Gaussian approximation with full covariance. Generated by `linreg_bayes_svi_hmc_pyro.ipynb`.

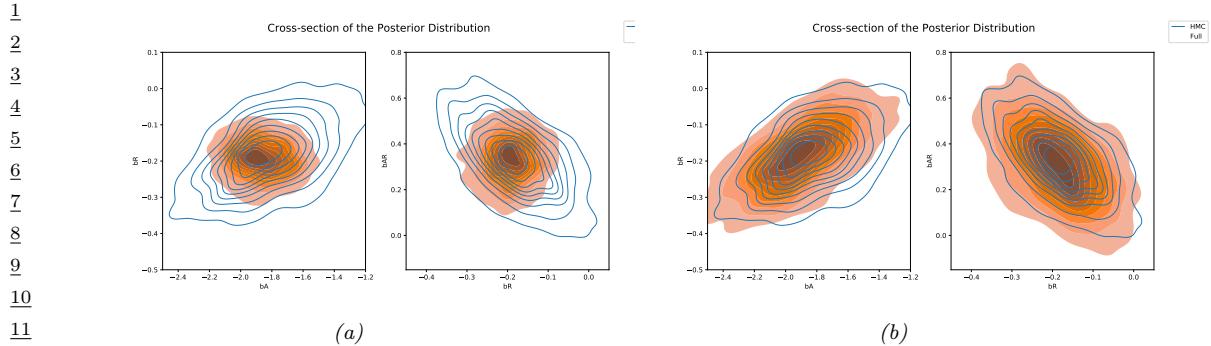


Figure 10.13: Joint posterior of pairs of variables for the linear model applied to the Africa data. (a) Blue is HMC, orange is Gaussian approximation with diagonal covariance. (b) Blue is HMC, orange is Gaussian approximation with full covariance. Generated by [linreg_bayes_svi_hmc_pyro.ipynb](#).

ruggedness for African countries, but decreases for non-African countries. (The reasons for this are unclear, but [NP12] suggest that it is because more rugged Africa countries were less exploited by the slave trade, and hence are now wealthier.)

Now we consider a variational approximation to the posterior, of the form $p(\boldsymbol{\theta}|\mathcal{D}) \approx q(\boldsymbol{\theta}) = q(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. (Since the standard deviation σ must lie in the interval $[0, 10]$ due to the uniform prior, first transform it to the unconstrained value $\tau = \text{logit}(\sigma/10)$ before applying the Gaussian approximation, as explained in Section 10.3.5.)

Suppose we initially choose a diagonal Gaussian approximation. In Figure 10.12a, we compare the marginals of this posterior approximation (for the bias term and the 3 regression coefficients) with the “exact” posterior from HMC. We see that the variational marginals have roughly the same mean, but their variances are too small, meaning they are overconfident. Furthermore, the variational approximation neglects any posterior correlations, as shown in Figure 10.13a.

We can improve the quality of the approximation by using a full covariance Gaussian. The resulting posterior marginals are shown in Figure 10.12b, and some bivariate posteriors are shown in Figure 10.13b. We see that the posterior approximation is now much more accurate.

Interestingly, both variational approximations give a similar predictive distribution to the HMC one in Figure 10.11. However, in some statistical problems we care about interpreting the parameters themselves (e.g., to assess the strength of the dependence on ruggedness), so a more accurate approximation is necessary to avoid reaching invalid conclusions.

37

38 10.3.5 Automatic differentiation VI

To apply Gaussian VI, we need to transform constrained parameters (such as variance terms) to unconstrained form, so they live in \mathbb{R}^D . For example, suppose the original variable has distribution $h \sim \Gamma(a, b)$, so $h \in \mathbb{R}_+$. We can define $z = \log(h)$, so $z \in \mathbb{R}$, where $p(z) = p(h)|dh/dz|$. We can compute the Jacobian term $|dh/dz|$ using automatic differentiation, and then apply SVI. This technique can be generalized to any distribution for which we can define a bijection to \mathbb{R}^D . This approach is called **automatic differentiation variational inference** or **ADVI** [Kuc+16].

As an example, let us revisit the GMM model from Section 10.2.6. We marginalize out the

discrete local latents \mathbf{z}_n analytically, so the only unknowns are the global parameters $\boldsymbol{\theta}$. Following Section 3.3.2.2, we use an LKJ prior for the covariance. Thus we get the following non-conjugate prior:

$$p(\boldsymbol{\theta}) = p(\boldsymbol{\pi}) \prod_{k=1}^K p(\boldsymbol{\mu}_k) p(\mathbf{R}_k) p(\boldsymbol{\sigma}_k) \quad (10.141)$$

$$p(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi} | \frac{1}{K} \mathbf{1}) \quad (10.142)$$

$$p(\boldsymbol{\mu}_k) = \prod_d \mathcal{N}(\mu_{k,d} | 0, 1) \quad (10.143)$$

$$p(\mathbf{R}_k) = \text{LKJ}(\mathbf{R}_k | 1) \quad (10.144)$$

$$p(\boldsymbol{\sigma}_k) = \prod_d \mathcal{N}_+(\sigma_{k,d} | 0, 1) \quad (10.145)$$

where \mathbf{R}_k is a lower triangular correlation matrix, $\boldsymbol{\sigma}_k$ is the scale vector, and $\boldsymbol{\Sigma}_k = \text{diag}(\boldsymbol{\sigma}_k) \mathbf{R}_k \text{diag}(\boldsymbol{\sigma}_k)$ is the induced (lower triangular) covariance matrix. The posterior approximation for the unconstrained parameters in \mathbb{R}^N is

$$q(\tilde{\boldsymbol{\theta}}) = \mathcal{N}(\tilde{\boldsymbol{\theta}} | \boldsymbol{\psi}_{\boldsymbol{\mu}}, \boldsymbol{\psi}_{\boldsymbol{\Sigma}}) \quad (10.146)$$

where

$$\tilde{\boldsymbol{\theta}} = f(\boldsymbol{\theta}) = [f_{\boldsymbol{\pi}}(\boldsymbol{\pi}), \{f_{\boldsymbol{\mu}}(\boldsymbol{\mu}_k), f_{\mathbf{R}}(\mathbf{R}_k), f_{\boldsymbol{\sigma}}(\boldsymbol{\sigma}_k)\}_{k=1}^K] \quad (10.147)$$

are the unconstrained parameters, derived from the original model parameters $\boldsymbol{\theta}$ via a stack of suitable bijections. The variational parameters $\boldsymbol{\psi}_{\boldsymbol{\mu}}$ and $\boldsymbol{\psi}_{\boldsymbol{\Sigma}}$ are optimized using ADVI.

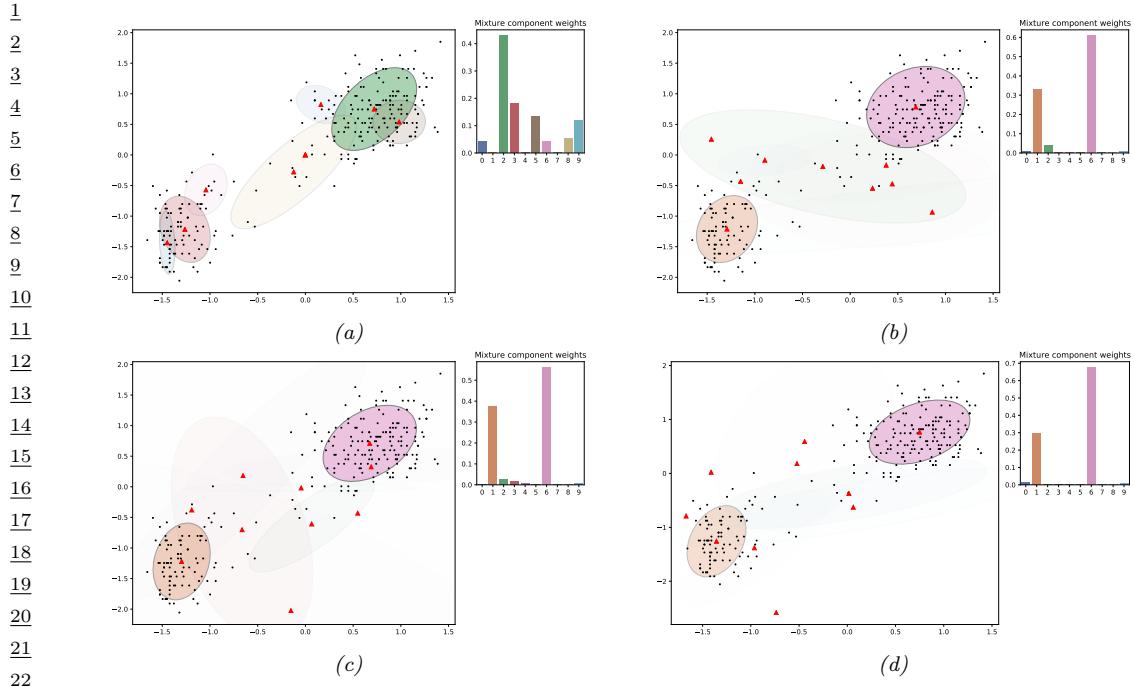
We apply this method to the Old Faithful dataset from Figure 10.7, using $K = 10$ mixture components. The results are shown in Figure 10.14. In the top left, we show the special case where we constrain the posterior to be a MAP estimate, by setting $\boldsymbol{\psi}_{\boldsymbol{\Sigma}} = \mathbf{0}$. We see that there is no sparsity in the posterior, since there is no Bayesian “Occam factor” from marginalizing out the parameters. In panels (c–d), we show 3 samples from the posterior. We see that the Bayesian method strongly prefers just 2 mixture components, although there is a small amount of support for some other Gaussian components (shown by the faint ellipses).

10.3.6 Beyond Gaussian posteriors

In this section, we show how to perform parameter inference for a GMM using reparameterized VI. As in Section 10.3.5, we marginalize out the discrete latent variables, so just need to approximate $p(\boldsymbol{\theta} | \mathcal{D})$. However, rather than transforming the variance parameters and mixing weights, and approximating them all with a Gaussian, we use “domain appropriate” conjugate priors and posteriors. Nevertheless, the form of our variational posterior will be chosen to be a reparameterizable distribution.

For simplicity, we assume diagonal covariance matrices. Thus the likelihood for one data point, $\mathbf{x} \in \mathbb{R}^D$, is

$$p(\mathbf{x} | \boldsymbol{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \text{diag}(\boldsymbol{\lambda}_k)^{-1}) \quad (10.148)$$



23 *Figure 10.14: Posterior over the mixing weights (histogram) and the means and covariances of each Gaussian*
 24 *mixture component, using $K = 10$, when fitting the model to the Old Faithful dataset from Figure 10.7.* (a)
 25 *MAP approximation. (b-d) 3 samples from the Gaussian approximation. The intensity of the shading is*
 26 *proportional to the mixture weight. Generated by `vb_gmm_tfp.ipynb`.*

27

28

29 where $\boldsymbol{\mu}_k = (\mu_{k1}, \dots, \mu_{kD})$ are the means, $\boldsymbol{\lambda}_k = (\lambda_{k1}, \dots, \lambda_{kD})$ are the precisions, and $\boldsymbol{\pi} =$
 30 (π_1, \dots, π_K) are the mixing weights. We use the following prior for these parameters:

$$31 \quad p(\boldsymbol{\theta}) = \left[\prod_{k=1}^K \prod_{d=1}^D \mathcal{N}(\mu_{kd}|0, 1) \text{Ga}(\lambda_{kd}|5, 5) \right] \text{Dir}(\boldsymbol{\pi}|\mathbf{1}) \quad (10.149)$$

34 We assume the following mean field posterior:
 35

$$36 \quad q(\boldsymbol{\theta}|\boldsymbol{\psi}_{\boldsymbol{\theta}}) = \left[\prod_{k=1}^K \prod_{d=1}^D \mathcal{N}(\mu_{kd}|m_{kd}, s_{kd}) \text{Ga}(\lambda_{kd}|\alpha_{kd}, \beta_{kd}) \right] \text{Dir}(\boldsymbol{\pi}|\mathbf{c}) \quad (10.150)$$

39 where $\boldsymbol{\psi}_{\boldsymbol{\theta}} = (\mathbf{m}_{1:K, 1:D}, \mathbf{s}_{1:K, 1:D}, \boldsymbol{\alpha}_{1:K, 1:D}, \boldsymbol{\beta}_{1:K, 1:D}, \mathbf{c})$ are the variational parameters for $\boldsymbol{\theta}$.
 40

We can compute the ELBO using
 41

$$42 \quad \mathcal{L}(\boldsymbol{\psi}_{\boldsymbol{\theta}}|\mathcal{D}) = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi}_{\boldsymbol{\theta}})} \left[\sum_{n=1}^N \log p(\mathbf{x}_n|\boldsymbol{\theta}) \right] - D_{\text{KL}}(q(\boldsymbol{\theta}|\boldsymbol{\psi}_{\boldsymbol{\theta}})\|p(\boldsymbol{\theta})) \quad (10.151)$$

45 We can approximate the first term using minibatching. Since $q(\boldsymbol{\theta}|\boldsymbol{\psi}_{\boldsymbol{\theta}})$ is reparameterizable (see
 46 Section 10.3.3), we can sample from it and push gradients inside. If we use a single posterior sample
 47

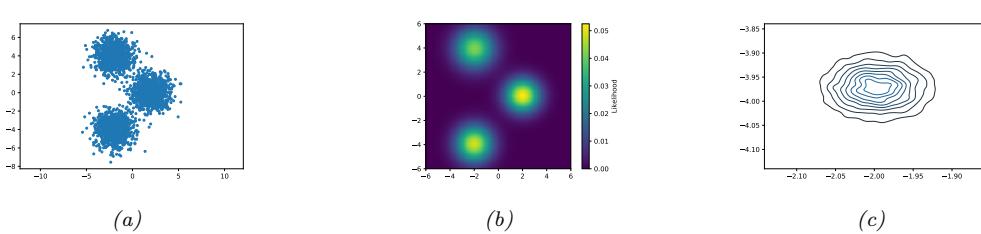


Figure 10.15: SVI for fitting a mixture of 3 Gaussians in 2d. (a) 3000 training points. (b) Fitted density, plugging in the posterior mean parameters. (c) Kernel density estimate fit to 10,000 samples from $q(\mu_1|\psi_\theta)$. Generated by `svi_gmm_demo_2d_tfp.py`.

per minibatch, $\theta^s \sim q(\theta|\psi_\theta)$, we get

$$\nabla_{\psi_\theta} \mathbb{L}(\psi_\theta | \mathcal{D}) \approx \frac{N}{B} \sum_{b=1}^B \nabla_{\psi_\theta} \log p(\mathbf{x}_b | \theta^s) - \nabla_{\psi_\theta} D_{\text{KL}}(q(\theta|\psi_\theta) \| p(\theta)) \quad (10.152)$$

We can now optimize this with SGD.

Figure 10.15 gives an example of this in practice. We generate a dataset from a mixture of 3 Gaussians in 2d, using $\mu_1^* = [2, 0]$, $\mu_2^* = [-2, -4]$, $\mu_3^* = [-2, 4]$, precisions $\lambda_{dk}^* = 1$, and uniform mixing weights, $\pi^* = [1/3, 1/3, 1/3]$. Figure 10.15a shows the training set of 3000 points. We fit this using SVI, with a batch size of 500, for 1000 epochs, using the Adam optimizer. Figure 10.15b shows the predictions of the fitted model. More precisely, it shows $p(\mathbf{x}|\bar{\theta})$, where $\bar{\theta} = \mathbb{E}_{q(\theta|\psi_\theta)}[\theta]$. Figure 10.15c shows a kernel density estimate fit to 10,000 samples from $q(\mu_1|\psi_\theta)$. We see that the posterior mean is $\mathbb{E}[\mu_1] \approx [-2, -4]$. Due to label switching unidentifiability, we see this matches μ_2^* rather than μ_1^* .

10.3.7 Amortized inference

Suppose we want to perform parameter estimation in a model with local latent variables:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{n=1}^N \log \sum_{z_n} p(\mathbf{x}_n, z_n | \theta) \quad (10.153)$$

To compute the marginal likelihood, we need to compute the posteriors $q_\psi(z_n)$ for each example n . When using BBVI or RVI for this, we have to solve an optimization problem for each $q(z_n|\psi_n)$, which can be slow.

An alternative approach is to train a model, known as an **inference network** or **recognition network**, to predict ψ_n from the observed data, \mathbf{x}_n , using $\psi_n = f_\phi^{\text{inf}}(\mathbf{x}_n)$. This technique is known as **amortized inference**, since we are reducing the cost of per-example time inference by training a model that is shared across all examples (see e.g., [Amo22] for a general discussion of amortized optimization). For brevity, we will write

$$q(z_n|\psi_n) = q(z_n|f_\phi^{\text{inf}}(\mathbf{x}_n)) = q_\phi(z_n|\mathbf{x}_n) \quad (10.154)$$

The “**amortized ELBO**”, for a model with local latents and fixed global parameters, becomes

$$\hat{L}(\phi, \theta | \mathcal{D}) = \frac{1}{N} \sum_{n=1}^N [\mathbb{E}_{q_\phi(z_n | \mathbf{x}_n)} [\log p_\theta(\mathbf{x}_n, z_n) - \log q_\phi(z | \mathbf{x}_n)]] \quad (10.155)$$

We can approximate this by sampling a single data point $\mathbf{x}_n \sim p_{\mathcal{D}}$, and then sampling a single latent $z_n \sim q_\phi(z_n | \mathbf{x}_n)$, as in DSVI, to get

$$L(\phi, \theta | \mathbf{x}_n, z_n) = \log p_\theta(\mathbf{x}_n, z_n^s) - \log q_\phi(z_n) \quad (10.156)$$

(We call this the “**per-sample ELBO**”, although [Ble17] call it the **instantaneous ELBO**.) If the posteriors are reparameterizable, we can push gradients inside and then apply SGD. See Algorithm 10 for the resulting pseudocode.

Algorithm 10: Amortized SVI

```

1 Initialize  $\theta, \phi$ 
2 repeat
3   Sample  $\mathbf{x}_n \sim p_{\mathcal{D}}$ 
4   Sample  $z_n \sim q_\phi(z | \mathbf{x}_n)$ 
5    $\theta := \theta + \eta \nabla_\theta \hat{L}(\phi, \theta | \mathbf{x}_n, z_n)$ 
6    $\phi := \phi + \eta \nabla_\phi \hat{L}(\phi, \theta | \mathbf{x}_n, z_n)$ 
7   Update learning rate  $\eta$ 
8 until converged;

```

This method is very widely used for fitting LVMs, e.g., for VAEs (see Section 22.2), for topic models [SS17a], for probabilistic programming [RHG16a], for CRFs [TG18], etc. However, the use of an inference network can result in a suboptimal setting of the local variational parameters ψ_n . This is called the **amortization gap** [CLD18]. We can close this gap by using the inference network to warm-start an optimizer for ψ_n ; this is known as **semi-amortized VI** [Kim+18c]. The key insight is that the local SVI procedure is itself differentiable, so the inference network and generative model can be trained end-to-end. (See also [MYM18], who propose a closely related method called **iterative amortized inference**.)

An alternative approach is to use the inference network as a proposal distribution. If we combine this with importance sampling, we get the IWAE bound of Section 10.5.1. If we use this with Metropolis Hastings, we get a VI-MCMC hybrid (see Section 10.4.5).

10.3.8 Exploiting partial conjugacy

If the full conditionals of the joint model are conjugate distributions, we can use the VMP approach of Section 10.2.7 to approximate the posterior one term at a time, similar to Gibbs sampling (Section 12.3). However, in many models, some parts of the joint distribution are conjugate, and some are non-conjugate. In [KL17a] they proposed the **conjugate-computation variational inference** or **CVI** method to tackle models of this form. They exploit the partial conjugacy to perform some updates in closed form, and perform the remaining updates using stochastic approximations.

To explain the method in more detail, let us assume the joint distribution has the form

$$p(\mathbf{y}, \mathbf{z}) \propto \tilde{p}_{nc}(\mathbf{y}, \mathbf{z}) \tilde{p}_c(\mathbf{y}, \mathbf{z}) \quad (10.157)$$

where \mathbf{z} are all the latents (global or local), \mathbf{y} are all the observabels (data)⁵, p_c is the conjugate part, p_{nc} is the non-conjugate part, and the tilde symbols indicate that these distributions may not be normalized wrt \mathbf{z} . More precisely, we assume the conjugate part is an exponential family model of the following form:

$$\tilde{p}_c(\mathbf{y}, \mathbf{z}) = h(\mathbf{z}) \exp[\mathcal{T}(\mathbf{z})^\top \boldsymbol{\eta} - A_c(\boldsymbol{\eta})] \quad (10.158)$$

where $\boldsymbol{\eta}$ is a *known* vector of natural parameters. (Any unknown model parameters should be included in the latent state \mathbf{z} , as we illustrate below.) We also assume that the variational posterior is an exponential family model with the same sufficient statistics, but different parameters:

$$q(\mathbf{z}|\boldsymbol{\lambda}) = h(\mathbf{z}) \exp[\mathcal{T}(\mathbf{z})^\top \boldsymbol{\eta} - A(\boldsymbol{\lambda})] \quad (10.159)$$

The mean parameters are given by $\boldsymbol{\mu} = \mathbb{E}_q [\mathcal{T}(\mathbf{z})]$. We assume the sufficient statistics are minimal, so that there is a unique 1:1 mapping between $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$: using

$$\boldsymbol{\mu} = \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda}) \quad (10.160)$$

$$\boldsymbol{\lambda} = \nabla_{\boldsymbol{\mu}} A^*(\boldsymbol{\mu}) \quad (10.161)$$

where A^* is the conjugate of A (see Section 2.5.4). The ELBO is given by

$$\mathcal{L}(\boldsymbol{\lambda}) = \mathbb{E}_q [\log p(\mathbf{y}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\lambda})] \quad (10.162)$$

$$\mathcal{L}(\boldsymbol{\mu}) = \mathcal{L}(\boldsymbol{\lambda}(\boldsymbol{\mu})) \quad (10.163)$$

The simplest way to fit this variational posterior is to perform SGD on the ELBO wrt the natural parameters:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \eta_t \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}_t) \quad (10.164)$$

(Note the + sign in front of the gradient, since we are maximizing the ELBO.) The above gradient update is equivalent to solving the following optimization problem:

$$\boldsymbol{\lambda}_{t+1} = \underset{\boldsymbol{\lambda} \in \Omega}{\operatorname{argmin}} \underbrace{(\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}_t))^\top \boldsymbol{\lambda} - \frac{1}{2\eta_t} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}_t\|_2^2}_{J(\boldsymbol{\lambda})} \quad (10.165)$$

where Ω is the space of valid natural parameters, and $\|\cdot\|_2$ is the Euclidean norm. To see this, note that the first order optimality conditions satisfy

$$\nabla_{\boldsymbol{\lambda}} J(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}_t) - \frac{1}{2\eta_t} (2\boldsymbol{\lambda} - 2\boldsymbol{\lambda}_t) = \mathbf{0} \quad (10.166)$$

⁵ 5. We denote observables by \mathbf{y} since the examples we consider later on are conditional models, where \mathbf{x} denote the inputs.

1 from which we get Equation (10.164).

2 We can replace the Euclidean distance with a more general proximity function, such as the Bregman
3 divergence between the distributions (see Section 6.5.1). This gives rise to the **mirror descent**
4 algorithm (Section 6.5). We can also perform updates in the mean parameter space. In [RM15a],
5 they show that this is equivalent to performing natural gradient updates in the natural parameter
6 space. Thus this method is sometimes called **natural gradient VI** or **NGVI**. Combining these two
7 steps gives the following update equation:

8

$$\underline{10} \quad \boldsymbol{\mu}_{t+1} = \operatorname{argmin}_{\boldsymbol{\mu} \in \mathcal{M}} (\nabla_{\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}_t))^T \boldsymbol{\mu} - \frac{1}{\eta_t} B_{A^*}(\boldsymbol{\mu} || \boldsymbol{\mu}_t) \quad (10.167)$$

11

12 where \mathcal{M} is the space of valid mean parameters and $\eta_t > 0$ is a stepsize.

13 In [KL17a], they show that the above update is equivalent to performing exact Bayesian inference
14 in the following conjugate model:

15

$$\underline{16} \quad q(\mathbf{z} | \boldsymbol{\lambda}_{t+1}) \propto e^{\mathcal{T}(\mathbf{z})^T \tilde{\boldsymbol{\lambda}}_t} \tilde{p}_c(\mathbf{y}, \mathbf{z}) \quad (10.168)$$

17

18 We can think of the first term as an exponential family approximation to the non-conjugate part of
19 the model, using local variational natural parameters $\tilde{\boldsymbol{\lambda}}_t$. (These are similar to the site parameters
20 used in expectation propagation Section 10.7.) These can be computed using the following recursive
21 update:

22

$$\underline{23} \quad \tilde{\boldsymbol{\lambda}}_t = (1 - \eta_t) \tilde{\boldsymbol{\lambda}}_{t-1} + \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q(\mathbf{z} | \boldsymbol{\mu}_t)} [\log \tilde{p}_{nc}(\mathbf{y}, \mathbf{z})] \quad (10.169)$$

24

25 where $\tilde{\boldsymbol{\lambda}}_0 = \mathbf{0}$ and $\tilde{\boldsymbol{\lambda}}_1 = \boldsymbol{\eta}$. (Details on how to compute this derivative are given in Section 6.4.5.)
26 Once we can have “conjugated” the non-conjugate part, the natural parameter of the new variational
27 posterior is obtained by

28

$$\underline{29} \quad \boldsymbol{\lambda}_{t+1} = \tilde{\boldsymbol{\lambda}}_t + \boldsymbol{\eta} \quad (10.170)$$

30

31 This corresponds to a multiplicative update of the form

32

$$\underline{33} \quad q_{t+1}(\mathbf{z}) \propto q_t(\mathbf{z})^{1-\eta_t} \left[\exp(\tilde{\boldsymbol{\lambda}}_t^T \mathcal{T}(\mathbf{z})) \right]^{\eta_t} \quad (10.171)$$

34

35 We give some examples of this below.

36 10.3.8.1 Example: Gaussian process with non-Gaussian likelihoods

37 In Chapter 18, we discuss Gaussian processes, which are a popular model for non-parametric
38 regression. Given a set of N inputs $\mathbf{x}_n \in \mathcal{X}$ and outputs $y_n \in \mathbb{R}$, we define the following joint
39 Gaussian distribution:

40

$$\underline{41} \quad p(\mathbf{y}_{1:N}, \mathbf{z}_{1:N} | \mathbf{X}) = \left[\prod_{n=1}^N \mathcal{N}(y_n | z_n, \sigma^2) \right] \mathcal{N}(\mathbf{z} | \mathbf{0}, \mathbf{K}) \quad (10.172)$$

42

43 where \mathbf{K} is the kernel matrix computed using $K_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$, and $z_n = f(\mathbf{x}_n)$ is the unknown
44 function value for input n . Since this model is jointly Gaussian, we can easily compute the exact
45 posterior $p(\mathbf{z} | \mathbf{y})$ in $O(N^3)$ time. (Faster approximations are also possible, see Section 18.5.)

46

One challenge with GPs arises when the likelihood function $p(y_n|z_n)$ is non-Gaussian, as occurs with classification problems. To tackle this, we will use CVI. Since the conjugate part of the model is a Gaussian, we require that the variational approximation also be Gaussian, so we use $q(\mathbf{z}|\boldsymbol{\lambda}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\lambda}^{(1)}, \boldsymbol{\lambda}^{(2)})$.

Since the likelihood term factorizes across data points $n = 1 : N$, we will only need to compute marginals of this variational posterior. From Section 2.5.2.3 we know that the sufficient statistics and natural parameters of a univariate Gaussian are given by

$$\mathcal{T}(z_n) = [z_n, z_n^2] \quad (10.173)$$

$$\boldsymbol{\lambda}_n = \left[\frac{m_n}{v_n}, -\frac{1}{2v_n} \right] \quad (10.174)$$

The corresponding moment parameters are

$$\boldsymbol{\mu}_n = [m_n, m_n^2 + v_n] \quad (10.175)$$

$$m_n = v_n \lambda_n^{(1)} \quad (10.176)$$

$$v_n = \frac{1}{2\lambda_n^{(2)}} \quad (10.177)$$

We need to compute the gradient terms $\nabla_{\boldsymbol{\mu}_n} \mathbb{E}_{\mathcal{N}(z_n|\boldsymbol{\mu}_n)} [\log p(y_n|z_n)]$. We can do this by sampling z_n from the local Gaussian posterior, and then pushing gradients inside, using the results from Section 6.4.5.1. Let the resulting stochastic gradients at step t be $\hat{g}_{n,t}^{(1)}$ and $\hat{g}_{n,t}^{(2)}$. We can then update the likelihood approximation as follows:

$$\tilde{\lambda}_{n,t}^{(i)} = (1 - \eta_t) \tilde{\lambda}_{n,t-1}^{(i)} + \eta_t \hat{g}_{n,t}^{(i)} \quad (10.178)$$

We can also perform a “doubly stochastic” approximation (as in Section 10.3.2) by just updating a random subset of these terms. Once we have updated the likelihood, we can update the posterior using

$$q(\mathbf{z}|\boldsymbol{\lambda}_{t+1}) \propto \left[\prod_{n=1}^N e^{z_n \tilde{\lambda}_{n,t}^{(1)} + z_n^2 \tilde{\lambda}_{n,t}^{(2)}} \right] \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{K}) \quad (10.179)$$

$$\propto \left[\prod_{n=1}^N \mathcal{N}_c(z_n|\tilde{\lambda}_{n,t}^{(1)}, \tilde{\lambda}_{n,t}^{(2)}) \right] \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{K}) \quad (10.180)$$

$$= \left[\prod_{n=1}^N \mathcal{N}(z_n|\tilde{m}_{n,t}, \tilde{v}_{n,t}) \right] \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{K}) \quad (10.181)$$

$$= \left[\prod_{n=1}^N \mathcal{N}(\tilde{m}_{n,t}|z_n, \tilde{v}_{n,t}) \right] \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{K}) \quad (10.182)$$

where $\tilde{m}_{n,t}$ and $\tilde{v}_{n,t}$ are derived from $\tilde{\lambda}_{n,t}$. We can think of this as **Gaussianizing the likelihood** at each step, where we replace the observations y_n by **pseudo-observations** $\tilde{m}_{n,t}$ and use a variational variance $\tilde{v}_{n,t}$. This lets us use exact GP regression updates in the inner loop. See [SKZ19; Cha+20] for details.

1 **10.3.8.2 Example: Bayesian logistic regression**

3 In this section, we discuss how to compute a Gaussian approximation to $p(\mathbf{w}|\mathcal{D})$ for a binary logistic
4 regression model with a Gaussian prior on the weights. We will use CVI in which we “Gaussianize”
5 the likelihoods, and then perform closed form Bayesian linear regression in the inner loop. This is
6 similar to the approach used in Section 15.3.7, where we derive a quadratic lower bound to the log
7 likelihood. However, such “local VI” methods are not guaranteed to converge to a local maximum of
8 the ELBO [Kha12], unlike the CVI method.

9 The joint distribution has the form

11

$$\begin{aligned} \underline{12} \quad p(\mathbf{y}_{1:N}, \mathbf{w} | \mathbf{X}) &= \left[\prod_{n=1}^N p(y_n | z_n) \right] \mathcal{N}(\mathbf{w} | \mathbf{0}, \delta \mathbf{I}) \\ \underline{13} \end{aligned} \tag{10.183}$$

15 where $z_n = \mathbf{w}^\top \mathbf{x}_n$ is the local latent, and $\delta > 0$ is the prior variance (analogous to an ℓ_2 regularizer).
16 We compute the local Gaussian likelihood terms $\tilde{\lambda}_n$ as in in Section 10.3.8.1. We then have the
17 following variational joint:

19

$$\begin{aligned} \underline{20} \quad q(\mathbf{w} | \boldsymbol{\lambda}_{t+1}) &\propto \left[\prod_{n=1}^N \mathcal{N}(\tilde{m}_{n,t} | \mathbf{w}^\top \mathbf{x}_n, \tilde{v}_{n,t}) \right] \mathcal{N}(\mathbf{w} | \mathbf{0}, \delta \mathbf{I}) \\ \underline{21} \end{aligned} \tag{10.184}$$

22 This corresponds to a Bayesian linear regression problem with pseudo-observations $\tilde{m}_{n,t}$ and variational
23 variance $\tilde{v}_{n,t}$.

26 **10.3.8.3 Example: Kalman smoothing with GLM likelihoods**

28 We can extend the above examples to perform posterior inference in a linear-Gaussian state-space
29 model (Section 31.2) with generalized linear model (GLM) likelihoods: we alternate between Gaus-
30 sianizing the likelihoods and running the Kalman smoother (Section 8.4.4).

32 **10.3.9 Online variational inference**

34 In this section, we discuss how to perform **online variational inference**. In particular, we discuss
35 the **streaming variational Bayes (SVB)** approach of [Bro+13] in which we, at step t , we compute
36 the new posterior using the previous posterior as the prior:

38

$$\begin{aligned} \underline{39} \quad \boldsymbol{\psi}_t &= \operatorname{argmin}_{\boldsymbol{\psi}} \underbrace{\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})} [\ell_t(\boldsymbol{\theta})] + D_{\text{KL}}(q(\boldsymbol{\theta}|\boldsymbol{\psi}) \| q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1}))}_{-\mathbb{L}_t(\boldsymbol{\psi})} \\ \underline{40} \end{aligned} \tag{10.185}$$

41

$$\begin{aligned} \underline{42} \quad &= \operatorname{argmin}_{\boldsymbol{\psi}} \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})} [\ell_t(\boldsymbol{\theta}) + \log q(\boldsymbol{\theta}|\boldsymbol{\psi}) - \log q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})] \\ \underline{43} \end{aligned} \tag{10.186}$$

44 where $\ell_t(\boldsymbol{\theta}) = -\log p(\mathcal{D}_t|\boldsymbol{\theta})$ is the negative log likelihood (or, more generally, some loss function)
45 of the data batch at step t . This approach is also called **variational continual learning** or **VCL**
46 [Ngu+18a]. (We discuss continual learning in Section 20.7.)

47

1 **10.3.9.1 FOO-VB**

3 In this section, we discuss a particular implementation of sequential VI called **FOO-VB**, which
4 stands for ‘‘Fixed-point Operator for Online Variational Bayes’’ [Zen+21]. This assumes Gaussian
5 priors and posteriors. In particular, let

7 $q(\boldsymbol{\theta}|\psi_t) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad q(\boldsymbol{\theta}|\psi_{t-1}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}, \mathbf{V}) \quad (10.187)$

8 In this case, we can write the ELBO as follows:

10 $\mathbb{L}_t(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2} \left[\log \frac{\det(\mathbf{V})}{\det(\boldsymbol{\Sigma})} - D + \text{tr}(\mathbf{V}^{-1}\boldsymbol{\Sigma}) + (\mathbf{m} - \boldsymbol{\mu})^\top \mathbf{V}^{-1}(\mathbf{m} - \boldsymbol{\mu}) \right] + \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})} [\ell_t(\boldsymbol{\theta})] \quad (10.188)$

12 where D is the dimensionality of $\boldsymbol{\theta}$.

13 Let $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^\top$. We can compute the new variational parameters by solving the joint first order
14 stationary conditions, $\nabla_{\boldsymbol{\mu}} \mathbb{L}_t(\boldsymbol{\mu}, \mathbf{L}) = \mathbf{0}$ and $\nabla_{\mathbf{L}} \mathbb{L}_t(\boldsymbol{\mu}, \mathbf{L}) = \mathbf{0}$. For the derivatives of the KL term, we
15 use the identities

17 $\frac{\partial \text{tr}(\mathbf{V}^{-1}\boldsymbol{\Sigma})}{\partial L_{ij}} = 2 \sum_n V_{in}^{-1} L_{nj} \quad (10.189)$

19 $\frac{\partial \log |\det(\mathbf{L})|}{\partial L_{ij}} = L_{ij}^{-T} \quad (10.190)$

22 For the derivatives of the expected loss, we use the the reparameterization trick, $\boldsymbol{\theta} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}$, and
23 following identities:

24 $\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\mu}, \mathbf{L})} [\ell_t(\boldsymbol{\theta})] = \mathbb{E}_{\boldsymbol{\epsilon}} [\ell_t(\boldsymbol{\theta}(\boldsymbol{\epsilon}))] \quad (10.191)$

26 $\frac{\partial \mathbb{E}_{\boldsymbol{\epsilon}} [\ell_t(\boldsymbol{\theta})]}{\partial L_{ij}} = \mathbb{E}_{\boldsymbol{\epsilon}} \left[\frac{\partial \ell_t(\boldsymbol{\theta})}{\partial \theta_i} \epsilon_j \right] \quad (10.192)$

28 Note that the expectation depends on the unknown variational parameters for q_t , so we get a fixed
29 point equation which we need to iterate. As a faster alternative, [Zen+18; Zen+21] propose to
30 evaluate the expectations using the variational parameters from the previous step, which then gives
31 the new parameters in a single step, similar to EM.

32 We now derive the update equations. From $\nabla_{\boldsymbol{\mu}} \mathbb{L}_t(\boldsymbol{\mu}, \mathbf{L}) = \mathbf{0}$ we get

34 $\mathbf{0} = -\mathbf{V}^{-1}(\mathbf{m} - \boldsymbol{\mu}) + \mathbb{E}_{\boldsymbol{\epsilon}} [\nabla \ell_t(\boldsymbol{\theta})] \quad (10.193)$

35 $\boldsymbol{\mu} = \mathbf{m} - \mathbf{V} \mathbb{E}_{\boldsymbol{\epsilon}} [\nabla \ell_t(\boldsymbol{\theta})] \quad (10.194)$

37 From $\nabla_{\mathbf{L}} \mathbb{L}_t(\boldsymbol{\mu}, \mathbf{L}) = \mathbf{0}$. we get

38 $0 = -(L^{-T})_{ij} + \sum_n V_{in}^{-1} L_{nj} + \mathbb{E}_{\boldsymbol{\epsilon}} \left[\frac{\partial \ell_t(\boldsymbol{\theta})}{\partial \theta_i} \epsilon_j \right] \quad (10.195)$

41 In matrix form, we have

42 $\mathbf{0} = -\mathbf{L}^{-T} + \mathbf{V}^{-1}\mathbf{L} + \mathbb{E}_{\boldsymbol{\epsilon}} [\nabla \ell_t(\boldsymbol{\theta}) \boldsymbol{\epsilon}^T] \quad (10.196)$

44 Explicitly solving for \mathbf{L} in the case of a general (or low rank) matrix $\boldsymbol{\Sigma}$ is somewhat complicated;
45 for the details, see [Zen+21]. Fortunately, in the case of a diagonal appproximation, things simplify
46 significantly, as we discuss in Section 10.3.9.2.

1 **10.3.9.2 Bayesian gradient descent**

3 Let $\mathbf{V} = \text{diag}(v_i^2)$, $\boldsymbol{\Sigma} = \text{diag}(\sigma_i^2)$, so $\mathbf{L} = \text{diag}(\sigma_i)$. Also, let $g_i = \frac{\partial \ell_t(\boldsymbol{\theta})}{\partial \theta_i}$, which depends on ϵ_i . Then
4 Equation (10.194) becomes
5

$$\mu_i = m_i - \eta v_i^2 \mathbb{E}_{\epsilon_i} [g_i(\epsilon_i)] \quad (10.197)$$

8 where we have included an explicit learning rate η to compensate for the fact that the fixed point
9 equation update is approximate. For the variance terms, Equation (10.196) becomes
10

$$0 = -\frac{1}{\text{diag}(\sigma_i)} + \frac{\text{diag}(\sigma_i)}{\text{diag}(v_i^2)} + \mathbb{E}_{\epsilon_i} [g_i \epsilon_i] \quad (10.198)$$

13 This is a quadratic equation for each σ_i :
14

$$\frac{1}{v_i^2} \sigma_i^2 + \mathbb{E}_{\epsilon_i} [g_i \epsilon_i] \sigma_i - 1 = 0 \quad (10.199)$$

18 the solution of which is given by the following (since $\sigma_i > 0$):
19

$$\sigma_i = \frac{-\mathbb{E}_{\epsilon_i} [g_i \epsilon_i] + \sqrt{(\mathbb{E}_{\epsilon_i} [g_i \epsilon_i])^2 + 4/v_i^2}}{2/v_i^2} = -\frac{1}{2} v_i^2 \mathbb{E}_{\epsilon_i} [g_i \epsilon_i] + \frac{1}{2} v_i^2 \sqrt{(\mathbb{E}_{\epsilon_i} [g_i \epsilon_i])^2 + 4/v_i^2} \quad (10.200)$$

$$= -\frac{1}{2} v_i^2 \mathbb{E}_{\epsilon_i} [g_i \epsilon_i] + \sqrt{\frac{v_i^4}{4} ((\mathbb{E}_{\epsilon_i} [g_i \epsilon_i])^2 + 4/v_i^2)} = -\frac{1}{2} v_i^2 \mathbb{E}_{\epsilon_i} [g_i \epsilon_i] + v_i \sqrt{1 + (\frac{1}{2} v_i \mathbb{E}_{\epsilon_i} [g_i \epsilon_i])^2} \quad (10.201)$$

26 We can approximate the above expectations using K Monte Carlo samples. Thus the overall algorithm
27 is very similar to standard SGD, except we compute the gradient K times, and we update $\boldsymbol{\mu} \in \mathbb{R}^D$
28 and $\boldsymbol{\sigma} \in \mathbb{R}^D$ rather than $\boldsymbol{\theta} \in \mathbb{R}^D$. In [Zen+18], they call the resulting algorithm “**Bayesian gradient**
29 **descent**”, and they show it works well on some continual learning problems (see Section 20.7). See
30 Algorithm 11 for the pseudocode, and see [Kur+20] for a related algorithm.
31

32 **10.3.9.3 Generalized variational continual learning**
33

34 One problem with the VCL objective in Equation (10.185) is that the KL term can cause the model
35 to become too sparse, which can prevent the model from adapting or learning new tasks. This
36 problem is called **variational overpruning** [TT17]. More precisely, the reason this happens as
37 is as follows: some weights might not be needed to fit a given dataset, so their posterior will be
38 equal to the prior; but sampling from these high-variance weights will add noise to the likelihood; to
39 reduce this, the optimization method will prefer to set the bias term to a large negative value, so
40 the corresponding unit is “turned off”, and thus has no effect on the likelihood. Unfortunately, these
41 “dead units” become stuck, so there is not enough network capacity to learn the next task, as shown
42 in Figure 20.19(b).

43 In [LST21], they propose a solution to this, known as **generalized variational continual**
44 **learning**. The first step is to downweight the KL term by a factor $\beta < 1$ to get
45

$$\mathcal{L}_t = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})} [\ell_t(\boldsymbol{\theta})] + \beta D_{\text{KL}} (q(\boldsymbol{\theta}|\boldsymbol{\psi}) \| q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})) \quad (10.202)$$

47

Algorithm 11: One step of Bayesian gradient descent

```

1 Function  $(\mu_t, \sigma_t, \mathcal{L}_t) = \text{BGD-update}(\mu_{t-1}, \sigma_{t-1}, \mathcal{D}_t; \eta, K)$ ;
2 for  $k = 1 : K$  do
3   Sample  $\epsilon^k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
4    $\theta^k = \mu_{t-1} + \sigma_{t-1} \odot \epsilon^k$ 
5    $\mathbf{g}^k = \nabla_{\theta} - \log p(\mathcal{D}_t | \theta)|_{\theta^k}$ 
6 for  $i = 1 : D$  do
7    $E_{1i} = \frac{1}{K} \sum_{k=1}^K g_i^k$ 
8    $E_{2i} = \frac{1}{K} \sum_{k=1}^K g_i^k \epsilon_i^k$ 
9    $\mu_{t,i} = \mu_{t-1,i} - \sigma_{t-1,i}^2 E_{1i}$ 
0    $\sigma_{t,i} = \sigma_{t-1,i} \sqrt{1 + (\frac{1}{2} \sigma_{t-1,i} E_{2i})^2} - \frac{1}{2} \sigma_{t-1,i}^2 E_{2i}$ 
1 for  $k = 1 : K$  do
2    $\theta^k = \mu_t + \sigma_t \odot \epsilon^k$ 
3    $\ell_t^k = -\log p(\mathcal{D}_t | \theta^k)$ 
4  $\mathcal{L}_t = - \left[ \frac{1}{K} \sum_{k=1}^K \ell_t^k + D_{\text{KL}}(\mathcal{N}(\mu_t, \sigma_t) \| \mathcal{N}(\mu_{t-1}, \sigma_{t-1})) \right]$ 

```

Interestingly, one can show that in the limit of $\beta \rightarrow 0$, this recovers several standard methods that use a Laplace approximation, based on the Hessian rather than the posterior covariance. In particular if Q is diagonal, this reduces to **Online elastic weight consolidation** [Sch+18]; if Q is block-diagonal and Kronecker factored, this reduces to **Online structured Lapalce** [RBB18b]; and if Q is a low-rank precision matrix, this reduces to the **SOLA** method [Yin+20].

The second step is to replace the prior and posterior by using **tempering**, which is useful when the model is misspecified [KJD21]. In the case of Gaussians, raising the distribution to the power λ is equivalent to tempering with a temperature of $\tau = 1/\lambda$, which is the same as scaling the covariance by λ^{-1} . Thus the GVCL objective becomes

$$\mathbb{E}_t = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})} [\ell_t(\boldsymbol{\theta})] + \beta D_{\mathbb{KL}}(q(\boldsymbol{\theta}|\boldsymbol{\psi})^\lambda \| q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})^\lambda) \quad (10.203)$$

Using a value of $\lambda \gg 1$ is equivalent to using a “**cold posterior**”, which we discuss in Section 17.3.

10.4 More accurate variational posteriors

In general, we can improve the tightness of the ELBO lower bound, and hence reduce the KL divergence of our posterior approximation, if we use more flexible posterior families (although optimizing within more flexible families may be slower, and can incur statistical error if the sample size is low [Bha+21]). In this section, we give several examples of more accurate variational posteriors, going beyond fully factored mean field approximations.

1 **10.4.1 Structured mean field**

3 The mean field assumption is quite strong, and can sometimes give poor results. Fortunately,
4 sometimes we can exploit **tractable substructure** in our problem, so that we can efficiently handle
5 some kinds of dependencies between the variables in the posterior in an analytic way, rather than
6 assuming they are all independent. This is called the **structured mean field** approach [SJ95].
7

8 A common example arises when applying VI to time series models, such as HMMs, where the
9 latent variables within each sequence are usually highly correlated across time. Rather than as-
10 suming a fully factorized posterior, we can treat each sequence $\mathbf{z}_{n,1:T}$ as a block, and just assume
11 independence between blocks and the parameters: $q(\mathbf{z}_{1:N,1:T}, \boldsymbol{\theta}) = q(\boldsymbol{\theta}) \prod_{n=1}^N q(\mathbf{z}_{n,1:T})$, where
12 $q(\mathbf{z}_{n,1:T}) = \prod_t q(\mathbf{z}_{n,t} | \mathbf{z}_{n,t-1})$. We can compute the joint distribution $q(\mathbf{z}_{n,1:T})$, taking into account
13 the dependence between time steps, using the forwards-backwards algorithm. For details, see
14 [JW14; Fot+14]. A similar approach was applied to the factorial HMM model, as we discuss in
15 Section 30.5.4.1.

16 An automatic way to derive a structured variational approximation to a probabilistic model,
17 specified by a probabilistic programming language, is discussed in [AHG20].

18

19 **10.4.2 Hierarchical (auxiliary variable) posteriors**

20 Suppose $q_\phi(\mathbf{z}|\mathbf{x}) = \prod_k q_\phi(z_k|\mathbf{x})$ is a factorized distribution, such as a diagonal Gaussian. This does
21 not capture dependencies between the latent variables (components of \mathbf{z}). We could of course use a
22 full covariance matrix, but this might be too expensive.
23

24 An alternative approach is to use a hierarchical model, in which we add **auxiliary latent variables**
25 \mathbf{a} , which are used to increase the flexibility of the variational posterior. In particular, we can still
26 assume $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a})$ is conditionally factorized, but when we marginalize out \mathbf{a} , we induce dependencies
27 between the elements of \mathbf{z} , i.e.,

28

$$q_\phi(\mathbf{z}|\mathbf{x}) = \int q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a}) q_\phi(\mathbf{a}|\mathbf{x}) d\mathbf{a} \neq \prod_k q_\phi(z_k|\mathbf{x}) \quad (10.204)$$

29

30

31 This is called a **hierarchical variational model** [Ran16], or an **auxiliary variable deep gener-
32 ative model** [Maa+16].

33 In [TRB16], they model $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a})$ as a Gaussian process, which is a flexible nonparametric
34 distribution (see Chapter 18), where \mathbf{a} are the inducing points. This combination is called a
35 **variational GP**.

36

37 **10.4.3 Normalizing flow posteriors**

38 Normalizing flows are a class of probability models which work by passing a simple source distribution,
39 such as a diagonal Gaussian, through a series of nonlinear, but invertible, mappings f to create
40 a more complex distribution final distribution. This can be used to get more accurate posterior
41 approximations, as we discuss in Section 24.1.3.2.

42 For example, an autoregressive flow for a sequence model $p(\mathbf{y})$ has the form

43

$$y_i = \mu_i(\mathbf{y}_{1:i-1}) + \sigma_i(\mathbf{y}_{1:i-1}) \epsilon_i \quad (10.205)$$

44

where μ_i and σ_i are functions of the previously generated outputs $\mathbf{y}_{1:i-1}$. Unfortunately sampling from such a model is inherently sequential, which makes it too slow for use in variational inference. However, we can invert this process to get

$$\epsilon_i = \frac{y_i - \mu_i(\mathbf{y}_{1:i-1})}{\sigma_i(\mathbf{y}_{1:i-1})} \quad (10.206)$$

This is called an **inverse autoregressive flow** (see Section 24.2.4.3 for details). In vector form, this can be written as

$$\boldsymbol{\epsilon} = (\mathbf{y} - \boldsymbol{\mu}(\mathbf{y})) / \boldsymbol{\sigma}(\mathbf{y}) \quad (10.207)$$

Alternatively, we can use

$$\boldsymbol{\epsilon} = \boldsymbol{\mu}(\mathbf{y}) + \boldsymbol{\sigma}(\mathbf{y}) \cdot \mathbf{y} \quad (10.208)$$

Due to the autoregressive structure of this transformation, we have $\partial\epsilon_i/\partial y_j = 0$ for $j > i$, so the Jacobian is lower triangular, and the determinant is a product of the diagonals. Hence

$$\log \det \left| \frac{\partial \boldsymbol{\epsilon}}{\partial \mathbf{y}} \right| = \sum_{k=1}^K \log \sigma_k(\mathbf{y}) \quad (10.209)$$

Hence we can easily compute

$$\log p(\mathbf{y}) = \log p(\boldsymbol{\epsilon}) - \log \det \left| \frac{\partial \boldsymbol{\epsilon}}{\partial \mathbf{y}} \right| \quad (10.210)$$

To apply this to approximate the posterior, we will chain together T such IAF models. First we sample $\boldsymbol{\epsilon}_0 \sim p(\boldsymbol{\epsilon})$ from a simple based distribution, then we compute $\boldsymbol{\epsilon}_t = f_t(\boldsymbol{\epsilon}_{t-1}, \mathbf{x})$ for $t = 1 : T$, and finally we set $\mathbf{z} = \boldsymbol{\epsilon}_T$. By the change of variables formula, the new distribution is given by

$$q(\mathbf{z}) = q(\boldsymbol{\epsilon}_0) \left| \det \frac{\partial \boldsymbol{\epsilon}_T}{\partial \boldsymbol{\epsilon}_0} \right|^{-1} \quad (10.211)$$

where the determinant of the Jacobian of the forward mapping is given by

$$\left| \det \frac{\partial \boldsymbol{\epsilon}_T}{\partial \boldsymbol{\epsilon}_0} \right| = \prod_{t=1}^T \left| \det \frac{\partial \boldsymbol{\epsilon}_t}{\partial \boldsymbol{\epsilon}_{t-1}} \right| \quad (10.212)$$

Hence

$$\log q_\phi(\mathbf{z} | \mathbf{x}) = \log p(\boldsymbol{\epsilon}_0) - \sum_{t=1}^T \log \left| \det \frac{\partial \boldsymbol{\epsilon}_t}{\partial \boldsymbol{\epsilon}_{t-1}} \right| \quad (10.213)$$

In [Kin+16], the base distribution is computed using a factorized Gaussian, as in a vanilla VAE: $\boldsymbol{\epsilon}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $(\boldsymbol{\mu}_0, \log \boldsymbol{\sigma}_0, \mathbf{h}) = e_\phi(\mathbf{x})$, and $\mathbf{z}_0 = \boldsymbol{\mu}_0 + \boldsymbol{\sigma}_0 \odot \boldsymbol{\epsilon}_0$, where \mathbf{h} is an extra output returned by the initial encoder. Then we gradually transform the initial sample using LSTM-like updates:

$$(\mathbf{m}_t, \mathbf{s}_t) = \text{AutoRegressiveNN}_t(\boldsymbol{\epsilon}_{t-1}, \mathbf{h}; \phi) \quad (10.214)$$

$$\boldsymbol{\sigma}_t = \boldsymbol{\sigma}(\mathbf{s}_t) \quad (10.215)$$

$$\boldsymbol{\epsilon}_t = \boldsymbol{\sigma}_t \odot \boldsymbol{\epsilon}_{t-1} + (1 - \boldsymbol{\sigma}_t) \odot \mathbf{m}_t \quad (10.216)$$

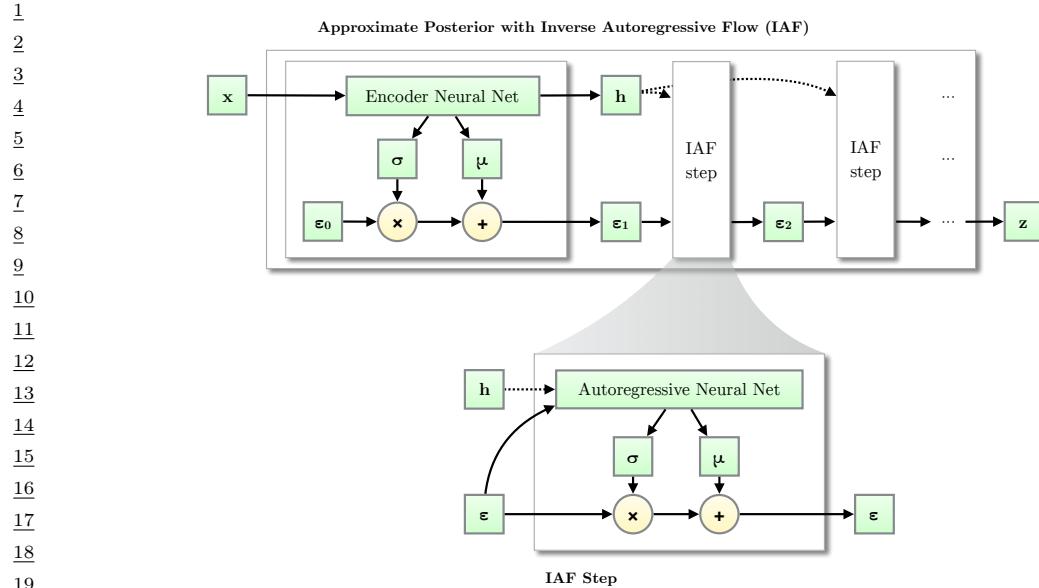


Figure 10.16: Using an inverse autoregressive flow network to approximate the posterior of a VAE. From Figure 3.1 of [KW19a]. Used with kind permission of Durk Kingma.

where $\sigma(s_t) \in [0, 1]^K$ is a vector of gating terms. At the end we set $\mathbf{z} = \epsilon_T$. Since each transformation has the form given in Equation (10.208), the log det Jacobian can be computed as before. Thus we have

$$\log q_\phi(\mathbf{z}|\mathbf{x}) = - \sum_{k=1}^K \left[\frac{1}{2} \epsilon_{0,k}^2 + \frac{1}{2} \log(2\pi) + \sum_{t=0}^T \log \sigma_{t,k} \right] \quad (10.217)$$

See Figure 10.16 for an illustration.

10.4.4 Implicit posteriors

In Chapter 27, we discuss implicit probability distributions, which are models which we can sample from, but which we cannot evaluate pointwise. For example, consider passing a Gaussian noise term, $\mathbf{z}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, through a nonlinear, *non-invertible* mapping f to create $\mathbf{z} = f(\mathbf{z}_0)$; it is easy to sample from $q(\mathbf{z})$, but it is intractable to evaluate the density $q(\mathbf{z})$ (unlike with flows). This makes it hard to evaluate the log density ratio $\log p_\theta(\mathbf{z})/q_\psi(\mathbf{z}|\mathbf{x})$, which is needed to compute the ELBO. However, we can use the same method as is used in GANs (generative adversarial networks, Chapter 27), in which we train a classifier that discriminates prior samples from samples from the variational posterior by evaluating $T(\mathbf{x}, \mathbf{z}) = \log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{z})$. See e.g., [TR19] for details.

10.4.5 Combining VI with MCMC inference

There are various ways to combine variational inference with MCMC to get an improved approximate posterior. In [SKW15], they propose **Hamiltonian Variational Inference**, in which they train an inference network to initialize an HMC sampler (Section 12.5). The gradient of the log posterior (wrt the latents), which is needed by HMC, is given by

$$\nabla_{\mathbf{z}} \log p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \nabla_{\mathbf{z}} \log [p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log p_{\boldsymbol{\theta}}(\mathbf{x})] = \nabla_{\mathbf{z}} \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) \quad (10.218)$$

This is easy to compute. They use the final sample to approximate the posterior $q_{\phi}(\mathbf{z}|\mathbf{x})$. To compute the entropy of this distribution, they also learn an auxiliary inverse inference network to reverse the HMC Markov chain.

A simpler approach is proposed in [Hof17]. Here they train an inference network to initialize an HMC sampler, using the standard ELBO for ϕ , but they optimize the generative parameters $\boldsymbol{\theta}$ using a stochastic approximation to the log marginal likelihood, given by $\log p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})$ where \mathbf{z} is a sample from the HMC chain. This does not require learning a reverse inference network, and avoids problems with variational pruning, since it does not use the ELBO for training the generative model.

10.5 Lower bounds

An alternative way to improve the quality of the posterior approximation is to optimize q wrt a bound that is a tighter approximation to the log marginal likelihood compared to the standard ELBO.

10.5.1 Multi-sample ELBO (IWAE bound)

In this section, we discuss a method known as the **importance weighted autoencoder** or **IWAE** [BGS16], which is a way to tighten the variational lower bound by using self-normalized importance sampling (Section 11.5.2). (It can also be interpreted as standard ELBO maximization in an expanded model, where we add extra auxiliary variables [CMD17; DS18; Tuc+19].)

Let the inference network $q_{\phi}(\mathbf{z}|\mathbf{x})$ be viewed as a proposal distribution for the target posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. Define $w_s^* = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s)}{q_{\phi}(\mathbf{z}_s|\mathbf{x})}$ as the unnormalized importance weight for a sample, and $w_s = w_s^*/(\sum_{s'=1}^S w_{s'}^*)$ as the normalized importance weights. From Equation (11.43) we can compute an estimate of the marginal likelihood $p(\mathbf{x})$ using

$$\hat{p}_S(\mathbf{x}|\mathbf{z}_{1:S}) \triangleq \frac{1}{S} \sum_{k=1}^S \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s)}{q_{\phi}(\mathbf{z}_s|\mathbf{x})} = \frac{1}{S} \sum_{k=1}^S w_s \quad (10.219)$$

This is unbiased, i.e. $\mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\hat{p}_S(\mathbf{x}|\mathbf{z}_{1:S})] = p(\mathbf{x})$, where $q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x}) = \prod_{k=1}^S q_{\phi}(\mathbf{z}_s|\mathbf{x})$. In addition, since the estimator is always positive, we can take logarithms, and thus obtain a stochastic lower bound on the log likelihood:

$$L_S(\phi, \boldsymbol{\theta}|\mathbf{x}) \triangleq \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} \left[\log \left(\frac{1}{S} \sum_{s=1}^S w_s \right) \right] = \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\log \hat{p}_S(\mathbf{z}_{1:S})] \quad (10.220)$$

$$\leq \log \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\hat{p}_S(\mathbf{z}_{1:S})] = \log p(\mathbf{x}) \quad (10.221)$$

1 where we used Jensen’s inequality in the penultimate line, and the unbiased property in the last line.
 2 This is called the **multi-sample ELBO** or **IWAE bound** [BGS16]. If $S = 1$, \mathcal{L}_S reduces to the
 3 standard ELBO:
 4

$$\mathcal{L}_1(\phi, \theta | \mathbf{x}) = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log w] = \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} \quad (10.222)$$

5 One can show [BGS16] that increasing the number of samples S is guaranteed to make the bound
 6 tighter, thus making it a better proxy for the log likelihood. Intuitively, averaging the S samples
 7 inside the log removes the need for every sample \mathbf{z}_s to explain the data \mathbf{x} . This encourages the
 8 proposal distribution q to be less concentrated than the single-sample variational posterior.
 9

10.5.1.1 Pathologies of optimizing the IWAE bound

10 Unfortunately, increasing the number of samples in the IWAE bound can decrease the signal to
 11 noise ratio, resulting in learning a worse model [Rai+18]. Intuitively, the reason this happens is that
 12 increasing S reduces the dependence of the bound on the quality of the inference network, which
 13 makes the gradient of the ELBO wrt ϕ less informative (higher variance).
 14

15 One solution to this is to use the **doubly reparameterized gradient estimator** [TL18b].
 16 Another approach is to use alternative estimation methods that avoid ELBO maximization, such
 17 as using the thermodynamic variational objective (see Section 10.5.2) or the reweighted wake sleep
 18 algorithm (see Section 22.7).
 19

10.5.2 The thermodynamic variational objective (TVO)

20 In [MLW19; Bre+20b], they present the **thermodynamic variational objective** or **TVO**. This
 21 is an alternative to IWAE for creating tighter variational bounds, which has certain advantages,
 22 particularly for posteriors that are not reparameterizable (e.g., discrete latent variables). The
 23 framework also has close connections with the reweighted wake sleep algorithm from Section 22.7, as
 24 we will see in Section 10.6.2.
 25

26 The TVO technique uses **thermodynamic integration**, also called **path sampling**, which is
 27 a technique used in physics and phylogenetics to approximate intractable normalization constants
 28 of high dimensional distributions (see e.g., [GM98; LP06; FP08]). This is based on the insight
 29 that it is easier to calculate the ratio of two unknown constants than to calculate the constants
 30 themselves. This is similar to the idea behind annealed importance sampling (Section 11.5.4), but TI
 31 is deterministic. For details, see [MLW19; Bre+20b].
 32

10.6 Upper bounds

33 Classical variational inference minimizes $D_{\text{KL}}(q \| p)$, which maximizes a lower bound on $\log Z =$
 34 $\log p(\mathbf{x})$. The reverse KL is zero-forcing (mode-seeking), and can cause problems with overconfidence
 35 in the variational posterior. One possible solution to this is to minimize the forwards KL, $D_{\text{KL}}(p \| q)$,
 36 which is zero avoiding (mode-covering). However, it is intractable to compute this objective, since
 37 it requires taking expectations wrt the true posterior p . The expectation propagation algorithm
 38 (Section 10.7) tackles this by optimizing the KL locally, one term at a time. Unfortunately EP is not
 39 optimizing an overall bound, and may not converge.
 40

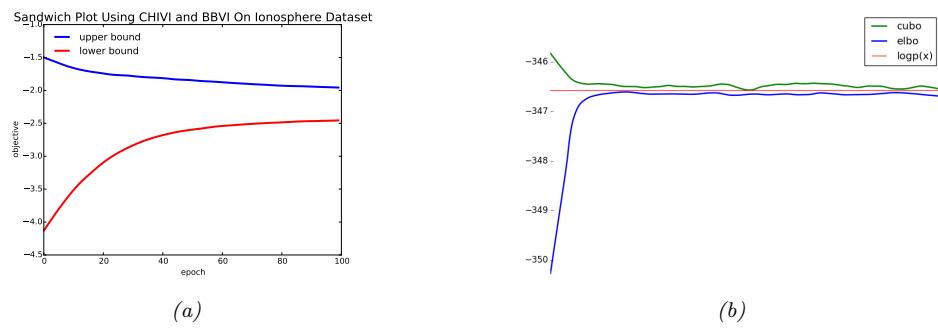


Figure 10.17: Illustration of the CUBO upper bound being minimized by CHIVI (Section 10.6.1) and the ELBO lower bound being maximized by BBVI (Section 10.3.1) on UCI Ionosphere dataset and a synthetic dataset where we know the true value of $\log p(\mathbf{x})$. From Figure 1 of [Die+17]. Used with kind permission of Adji Dieng.

In this section, we discuss some methods that provably minimize an upper bound on $\log Z$. This can help avoid some of the overconfidence issues with classic VI. In addition, it is useful to have lower and upper bounds on $\log Z$, since then we can compute a **sandwich estimate** of the form $\log \hat{Z}_- \leq \log Z \leq \hat{Z}_+$, where \hat{Z}_- is the lower bound and \hat{Z}_+ is the upper bound. This can be useful for model selection and evaluation, and potentially for parameter estimation. See Figure 10.17 for an example. (See also [GGA15] for an approach to sandwich estimation using bidirectional MC, which uses AIS (Section 11.5.4) starting at \mathbf{x} and starting at \mathbf{z} .)

10.6.1 Minimizing the χ -divergence upper bound

In this section, we discuss an algorithm for minimizing the χ -divergence between the true posterior and the variational posterior. This is defined by

$$\chi^2(p\|q) \triangleq \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\psi})} \left[\left(\frac{p(\mathbf{z}|\mathbf{x})}{q(\mathbf{z}|\boldsymbol{\psi})} \right)^2 - 1 \right] \quad (10.223)$$

Minimizing this is known as γ -divergence VI or CHIVI [Die+17].

It turns out that this will also minimize an upper bound on $\log Z$. To see this, note that

$$\mathbb{E}_{q(\mathbf{z}|\psi)} \left[\frac{p(\mathbf{x}, \mathbf{z})^2}{q(\mathbf{z}|\psi)^2} \right] = \mathbb{E}_{q(\mathbf{z}|\psi)} \left[\frac{p(\mathbf{z}|\mathbf{x})^2 p(\mathbf{x})^2}{q(\mathbf{z}|\psi)^2} \right] = p(\mathbf{x})^2 [1 + \chi^2(p(\mathbf{z}|\mathbf{x}) \| q(\mathbf{z}|\psi))] \quad (10.224)$$

80

$$\frac{1}{2} \log [1 + \chi^2(p(\mathbf{z}|\mathbf{x})||q(\mathbf{z}|\psi))] = -\log p(\mathbf{x}) + \frac{1}{2} \log \mathbb{E}_{q(\mathbf{z}|\psi)} \left[\left(\frac{p(\mathbf{z}, \mathbf{x})}{q(\mathbf{z}|\psi)} \right)^2 \right] \quad (10.225)$$

Since $\log p(x)$ is constant, minimizing this is equivalent to minimizing the χ -divergence upper

1 **bound or CUBO:**

3

$$\text{CUBO}(\psi) \triangleq \frac{1}{2} \log \mathbb{E}_{q(z|\psi)} \left[\left(\frac{p(z, x)}{q(z|\psi)} \right)^2 \right] \quad (10.226)$$

4

6 One can show that $\mathbb{L} \leq \log p(x) \leq \text{CUBO}$.

7 We can compute a Monte Carlo approximation to this upper bound using

9

$$U(\psi) = \frac{1}{2} \log \frac{1}{S} \sum_{s=1}^S \left(\frac{p(z^s, x)}{q(z^s|\psi)} \right)^2 \quad (10.227)$$

10

12 where $z^s \sim q(z|\psi)$. However, this is biased, i.e., $\mathbb{E}_q [U] \neq \text{CUBO}$, as is its gradient (this follows from

13 Jensen's inequality). So let us consider a different upper bound, namely $V(\psi) = \exp(2 \text{CUBO}(\psi))$.

14 We can compute an unbiased MC approximation of this using

15

$$V(\psi) = \frac{1}{S} \sum_{s=1}^S \left(\frac{p(z^s, x)}{q(z^s|\psi)} \right)^2 \quad (10.228)$$

16

18 We will assume $q(z)$ is reparameterizable, so we can sample ϵ^s and then compute $z^s = g(\epsilon^s, \psi)$,

19 where the distribution of ϵ^s is independent of ψ . Hence we can approximate the gradient at the

20 current parameter values using

22

$$\nabla_\psi V(\psi) = \frac{1}{S} \sum_{s=1}^S \nabla \left(\frac{p(z^s, x)}{q(z^s|\psi)} \right)^2 = -\frac{2}{S} \sum_{s=1}^S (w^{(s)})^2 \nabla_\psi \log q(z^s|\psi) \quad (10.229)$$

23

25 where $w^{(s)} = \frac{p(z^s)p(x|z^s)}{q(z^s|\psi_t)}$.

27 Unfortunately, the use of exponentiation in the definition of $V(\psi)$ makes this a very high variance

28 estimator [PHDV19].

30 10.6.2 Minimizing the evidence upper bound

31 Recall that the ELBO is given by

33

$$\mathbb{L}(\theta, \phi|x) = \log p_\theta(x) - D_{\text{KL}}(q_\phi(z|x) \| p_\theta(z|x)) \leq \log p_\theta(x) \quad (10.230)$$

34

35 By analogy, we can define the **evidence upper bound** or **EUBO** as follows:

36

$$\text{EUBO}(\theta, \phi|x) = \log p_\theta(x) + D_{\text{KL}}(p_\theta(z|x) \| q_\phi(z|x)) \geq \log p_\theta(x) \quad (10.231)$$

37

38 If we sample x from the generative model, and minimize $\mathbb{E}_{p_\theta(x)} [\text{EUBO}(\theta, \phi|x)]$ wrt ϕ , we recover

39 the sleep phase of the wake-sleep algorithm (see Section 22.7.2), which in turn is equivalent to

40 inference compilation.

41 Now suppose we sample x from the empirical distribution, and minimize $\mathbb{E}_{p_D(x)} [\text{EUBO}(\theta, \phi|x)]$

42 wrt ϕ . To approximate the expectation, we can use self-normalized importance sampling, as in

43 Equation (22.153), to get

44

$$\nabla_\phi \text{EUBO}(\theta, \phi|x) = \sum_{s=1}^S \bar{w}_s \nabla_\phi \log q_\phi(z^s|x) \quad (10.232)$$

45

46

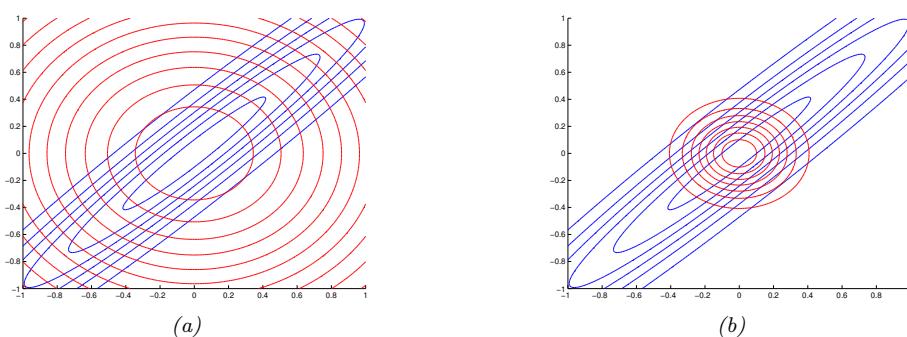


Figure 10.18: Illustrating forwards vs reverse KL on a symmetric Gaussian. The blue curves are the contours of the true distribution p . The red curves are the contours of a factorized approximation q . (a) Minimizing $D_{\text{KL}}(p\|q)$. (b) Minimizing $D_{\text{KL}}(q\|p)$. Adapted from Figure 10.2 of [Bis06]. Generated by `kl_pq_gauss.py`.

where $\bar{w}_s = w^{(s)} / (\sum_{s'} w^{(s')})$, and $w^{(s)} = \frac{p(\mathbf{x}, \mathbf{z}^s)}{q(\mathbf{z}^s | \phi_t)}$. This is equivalent to the “daydream” update (aka “wake-phase ϕ update”) of the wake-sleep algorithm (see Section 22.7.3). This is similar to Equation (10.229), except here we use normalized importance weights, rather than squared unnormalized weights. In [JS19], they show empirically that this **daydream estimator** (our name) works better than CUBO, in the sense that it is much lower variance. Furthermore, it can converge to the true $\log p_{\theta}(\mathbf{x})$ faster than the ELBO in some cases. See [MLW19] for further discussion.

10.7 Expectation propagation (EP)

One problem with lower bound maximization (i.e., standard VI) is that we are minimizing $D_{\text{KL}}(q\|p)$, which induces **zero-forcing** behavior, as we discussed in Section 5.1.3.2. This means that $q(\mathbf{z}|\mathbf{x})$ tends to be too compact (over-confident), to avoid the situation in which $q(\mathbf{z}|\mathbf{x}) > 0$ but $p(\mathbf{z}|\mathbf{x}) = 0$, which would incur infinite KL penalty.

Although zero-forcing can be desirable behavior for some multi-modal posteriors (e.g., mixture models), it is not so reasonable for many unimodal posteriors (e.g., Bayesian logistic regression). One way to avoid this problem is to minimize $D_{\text{KL}}(p\|q)$, which is zero-avoiding, as we discussed in Section 5.1.3.2. This tends to result in broad posteriors, which avoids overconfidence.

We illustrate the difference between these two objective functions below, and then discuss algorithms for minimizing $D_{\text{KL}}(p\|q)$.

10.7.1 Minimizing forwards vs reverse KL

Suppose the true target distribution p is a correlated 2d Gaussian, $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1})$, where

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \boldsymbol{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{pmatrix} \quad (10.233)$$

We will approximate this with a distribution q which is a product of two 1d Gaussians, i.e., a Gaussian with a diagonal covariance matrix.

1 **10.7.1.1 Moment projection**

2 Suppose we compute q as follows:

3
$$q = \underset{q}{\operatorname{argmin}} D_{\text{KL}}(p\|q) \quad (10.234)$$

4 This is called **M-projection**, or **moment projection**, since the optimal q matches the moments of
5 p (this is called **moment matching**). To see this, we assume q is an exponential family distribution,
6 and hence

7
$$q(\mathbf{x}) = h(\mathbf{x}) \exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})] \quad (10.235)$$

8 where $\mathcal{T}(\mathbf{x})$ is the vector of sufficient statistics, and $\boldsymbol{\eta}$ are the natural parameters. The first order
9 optimality conditions are as follows:

10
$$\partial_{\eta_i} D_{\text{KL}}(p\|q) = -\partial_{\eta_i} \sum_{\mathbf{x}} p(\mathbf{x}) \log q(\mathbf{x}) \quad (10.236)$$

11
$$= -\partial_{\eta_i} \sum_{\mathbf{x}} p(\mathbf{x}) \log (h(\mathbf{x}) \exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})]) \quad (10.237)$$

12
$$= -\partial_{\eta_i} \sum_{\mathbf{x}} p(\mathbf{x}) (\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})) \quad (10.238)$$

13
$$= -\sum_{\mathbf{x}} p(\mathbf{x}) \mathcal{T}_i(\mathbf{x}) + \mathbb{E}_{q(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] \quad (10.239)$$

14
$$= -\mathbb{E}_{p(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] + \mathbb{E}_{q(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] = 0 \quad (10.240)$$

15 where in the penultimate line we used the fact that the derivative of the log partition function yields
16 the expected sufficient statistics, as shown in Equation (2.177).

17 In this example, the optimal q must therefore have the following form:

18
$$q(\mathbf{x}) = \mathcal{N}(x_1|\mu_1, \Sigma_{11}) \mathcal{N}(x_2|\mu_2, \Sigma_{22}) \quad (10.241)$$

19 where $\boldsymbol{\Sigma} = \boldsymbol{\Lambda}^{-1}$. In Figure 10.18(a), we show the resulting distribution. We see that q covers
20 (includes) p , but its support is too broad.

21

22 **10.7.1.2 Information projection**

23 Now suppose we compute q as follows:

24
$$q = \underset{q}{\operatorname{argmin}} D_{\text{KL}}(q\|p) \quad (10.242)$$

25 This is called **I-projection**, or **information projection**.

26 In this example, one can show that the solution has the form

27
$$q(\mathbf{x}) = \mathcal{N}(x_1|m_1, \Lambda_{11}^{-1}) \mathcal{N}(x_2|m_2, \Lambda_{22}^{-1}) \quad (10.243)$$

28
$$m_1 = \mu_1 - \Lambda_{11}^{-1} \Lambda_{12} (m_2 - \mu_2) \quad (10.244)$$

29
$$m_2 = \mu_2 - \Lambda_{22}^{-1} \Lambda_{21} (m_1 - \mu_1) \quad (10.245)$$

30 Unless $\boldsymbol{\Lambda}$ is singular, the solution must satisfy $m_i = \mu_i$, so we have again captured the mean exactly.

31 However, the posterior variance is in general too narrow as shown in Figure 10.18(b). (See [Tur+08]
32 for a discussion of this point.)

33

10.7.2 EP as generalized ADF

In Section 8.8, we discussed assumed density filtering (ADF). This is a form of sequential Bayesian inference. At each step, it maintains a tractable approximation to the posterior, $q_t(\mathbf{z}) \in \mathcal{Q}$, updates it with the likelihood from the next observation, $\hat{p}_{t+1}(\mathbf{z}) \propto q_t(\mathbf{z})p(\mathbf{x}_t|\mathbf{z})$, and then projects the resulting updated posterior back to the tractable family using moment matching: $q_{t+1} = \operatorname{argmin}_{q \in \mathcal{Q}} D_{\text{KL}}(\hat{p}_{t+1} \| q)$.

ADF minimizes KL in the desired direction. However, it is a sequential algorithm, designed for the online setting. In the batch setting, the method can give different results depending on the order in which the updates are performed. In addition, if we perform multiple passes over the data, we will include the same likelihood terms multiple times, resulting in an overconfident posterior. In this section, we discuss **expectation propagation** or **EP** [Min01b], which can be seen as a generalization of ADF to the batch setting.

10.7.3 Algorithm

We assume the exact posterior can be written as follows:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z_p} \hat{p}(\boldsymbol{\theta}), \quad \hat{p}(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta}) \prod_{k=1}^K f_k(\boldsymbol{\theta}) \quad (10.246)$$

where $\hat{p}(\boldsymbol{\theta})$ is the unnormalized posterior, p_0 is the prior, f_k corresponds to the k 'th likelihood term or **local factor** (also called a **site potential**). Here $Z_p = p(\mathcal{D})Z_0$ is the normalization constant for the posterior, where Z_0 is the normalization constant for the prior. To simplify notation, we let $f_0(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta})$ be the prior.

We will approximate the posterior as follows:

$$q(\boldsymbol{\theta}) = \frac{1}{Z_q} \hat{q}(\boldsymbol{\theta}), \quad \hat{q}(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta}) \prod_{k=1}^K \tilde{f}_k(\boldsymbol{\theta}) \quad (10.247)$$

where $\tilde{f}_k \in \mathcal{Q}$ is the approximate local factor, and \mathcal{Q} is some tractable family in the exponential family, usually a Gaussian [Gel+14b].

We will optimize each \tilde{f}_i in turn, keeping the others fixed. We initialize each \tilde{f}_i using a uniform distribution, so $q(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta})$.

To compute the new local factor \tilde{f}_i^{new} , we proceed as follows. First we compute the **cavity distribution** by deleting the \tilde{f}_i from the approximate posterior by dividing it out:

$$q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) = \frac{q(\boldsymbol{\theta})}{\tilde{f}_i(\boldsymbol{\theta})} \propto \prod_{k \neq i} \tilde{f}_k(\boldsymbol{\theta}) \quad (10.248)$$

This can be implemented by subtracting off the natural parameters of \tilde{f}_i from q . The cavity distribution represents the effect of all the factors except for f_i (which is approximated by \tilde{f}_i).

Next we (conceptually) compute the **tilted distribution** by multiplying the exact factor f_i onto the cavity distribution:

$$q_i^{\text{tilted}}(\boldsymbol{\theta}) = \frac{1}{Z_i} f_i(\boldsymbol{\theta}) q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) \quad (10.249)$$

where $Z_i = \int q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) f_i(\boldsymbol{\theta}) d\boldsymbol{\theta}$ is the normalization constant for the tilted distribution. This is the result of combining the current approximation, excluding factor i , with the exact f_i term.

Unfortunately, the resulting tilted distribution may be outside of our model family (e.g., if we combine a Gaussian prior with a non-Gaussian likelihood). So we will approximate the tilted distribution as follows:

$$q_i^{\text{proj}}(\boldsymbol{\theta}) = \underset{\tilde{q} \in \mathcal{Q}}{\operatorname{argmin}} D(q_i^{\text{tilted}} \parallel \tilde{q}) = \operatorname{proj}(q_i^{\text{tilted}}) \quad (10.250)$$

This can be thought of as projecting the tilted distribution into the approximation family. If $D(q_i^{\text{tilted}} \parallel q) = D_{\text{KL}}(q_i^{\text{tilted}} \parallel q)$, this can be done by moment matching. For example, suppose the cavity distribution is Gaussian, $q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) = \mathcal{N}_c(\boldsymbol{\theta} | \mathbf{r}_{-i}, \mathbf{Q}_{-i})$, using the canonical parameterization. Then the log of the tilted distribution is given by

$$\log q_i^{\text{tilted}}(\boldsymbol{\theta}) = \alpha \log f_i(\boldsymbol{\theta}) - \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{Q}_{-i} \boldsymbol{\theta} + \mathbf{r}_{-i}^\top \boldsymbol{\theta} + \text{const} \quad (10.251)$$

Let $\hat{\boldsymbol{\theta}}$ be a local maximum of this objective. If \mathcal{Q} is the set of Gaussians, we can compute the projected tilted distribution as a Gaussian with the following parameters:

$$\mathbf{Q}_{\setminus i} = -\nabla_{\boldsymbol{\theta}}^2 \log q_i^{\text{tilted}}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}}, \quad \mathbf{r}_{\setminus i} = \mathbf{Q}_{\setminus i} \hat{\boldsymbol{\theta}} \quad (10.252)$$

This is called **Laplace propagation** [SVE04]. For more general distributions, we can use Monte Carlo approximations; this is known as **blackbox EP** [HL+16a; Li+18].

Finally, we compute a local factor that, if combined with the cavity distribution, would give the same results as this projected distribution:

$$\tilde{f}_i^{\text{new}}(\boldsymbol{\theta}) = \frac{q_i^{\text{proj}}(\boldsymbol{\theta})}{q_{-i}^{\text{cavity}}(\boldsymbol{\theta})} \quad (10.253)$$

This can be done by subtracting the natural parameters. We see that $q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) \tilde{f}_i^{\text{new}}(\boldsymbol{\theta}) = q_i^{\text{proj}}(\boldsymbol{\theta})$, so combining this approximate factor with the cavity distribution results in a distribution which is the best possible approximation (within \mathcal{Q}) to the results of using the exact factor.

10.7.4 Example

Figure 10.19 illustrates the process of combining a very non-Gaussian likelihood f_i with a Gaussian cavity prior q_{-i}^{cavity} to yield a nearly Gaussian tilted distribution q_i^{tilted} , which can then be approximated by a Gaussian using projection.

Thus instead of trying to “Gaussianize” each likelihood term f_i in isolation (as is done, e.g., in EKF), we try to find the best local factor \tilde{f}_i (within some family) that achieves approximately the same effect, when combined with all the other terms (represented by the cavity distribution, q_{-i}), as using the exact factor f_i . That is, we choose a local factor that works well in the context of all the other factors.

10.7.5 Optimization issues

In practice, EP can be numerically unstable. For example, if we use Gaussians as our local factors, we might end up with negative variance when we subtract the natural parameters. To reduce the

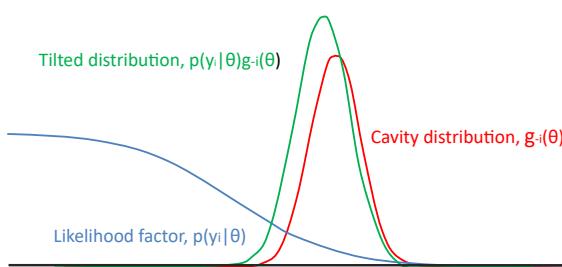


Figure 10.19: Combining a logistic likelihood factor $f_i = p(y_i | \theta)$ with the cavity prior, $q_{-i}^{cavity} = g_{-i}(\theta)$, to get the tilted distribution, $q_i^{tilted} = p(y_i | \theta)g_{-i}(\theta)$. Adapted from Figure 2 of [Gel+14b].

chance of this, it is common to use damping, in which we perform a partial update of each factor with a step size of δ . More precisely, we change the final step to be the following:

$$\tilde{f}_i^{\text{new}}(\theta) = \left(\tilde{f}_i(\theta) \right)^{1-\delta} \left(\frac{q_i^{\text{proj}}(\theta)}{q_{-i}^{\text{cavity}}} \right)^\delta \quad (10.254)$$

This can be implemented by scaling the natural parameters by δ . [ML02] suggest $\delta = 1/K$ as a safe strategy (where K is the number of factors), but this results in very slow convergence. [Gel+14b] suggest starting with $\delta = 0.5$, and then reducing to $\delta = 1/K$ over K iterations.

In addition to numerical stability, there is no guarantee that EP will converge in its vanilla form, although empirically it can work well, especially with log-concave factors f_i (e.g., as in GP classifiers).

10.7.6 Power EP and α -divergence

We also have choice about what divergence measure $D(q_i^{\text{tilted}} || q)$ to use when we approximate the tilted distribution. If we use $D_{\text{KL}}(q_i^{\text{tilted}} || q)$, we recover classic EP, as described above. If we use $D_{\text{KL}}(q || q_i^{\text{tilted}})$, we recover the reverse KL used in standard variational inference. We can generalize the above results by using α -divergences (Section 2.9.1.2), which allow us to interpolate between mode seeking and mode covering behavior, as shown in Figure 2.21. We can optimize the α -divergence by using the **power EP** method of [Min04].

Algorithmically, this is a fairly small modification to regular EP. In particular, we first compute the cavity distribution, $q_{-i}^{\text{cavity}} \propto \frac{q}{\tilde{f}_i^\alpha}$; we then approximate the tilted distribution, $q_i^{\text{proj}} = \text{proj}(q_{-i}^{\text{cavity}} f_i^\alpha)$; and finally we compute the new factor $\tilde{f}_i^{\text{new}} \propto \left(\frac{q_i^{\text{proj}}}{q_{-i}^{\text{cavity}}} \right)^{1/\alpha}$.

10.7.7 Stochastic EP

The main disadvantage of EP in the big data setting is that we need to store the $\tilde{f}_n(\theta)$ terms for each datapoint n , so we can compute the cavity distribution. If θ has D dimensions, and we use full covariance Gaussians, this requires $O(ND^2)$ memory.

1 The idea behind **stochastic EP** [LHLT15] is to approximate the local factors with a shared factor
2 that acts like an aggregated likelihood, i.e.,
3

4

$$\prod_{n=1}^N f_n(\boldsymbol{\theta}) \approx \tilde{f}(\boldsymbol{\theta})^N \quad (10.255)$$

5

6 where typically $f_n(\boldsymbol{\theta}) = p(\mathbf{x}_n | \boldsymbol{\theta})$. This exploits the fact that the posterior only cares about approximating the product of the likelihoods, rather than each likelihood separately. Hence it suffices for
7 $\tilde{f}(\boldsymbol{\theta})$ to approximate the average likelihood.
8

9 We can modify EP to this setting as follows. First, when computing the cavity distribution, we
10 use
11

12

$$q_{-1}(\boldsymbol{\theta}) \propto q(\boldsymbol{\theta}) / \tilde{f}(\boldsymbol{\theta}) \quad (10.256)$$

13

14 We then compute the tilted distribution
15

16

$$q_{\setminus n}(\boldsymbol{\theta}) \propto f_n(\boldsymbol{\theta}) q_{-1}(\boldsymbol{\theta}) \quad (10.257)$$

17

18 Next we derive the new local factor for this datapoint using moment matching:
19

20

$$\tilde{f}_n(\boldsymbol{\theta}) = \text{proj}(q_{\setminus n}(\boldsymbol{\theta})) / q_{-1}(\boldsymbol{\theta}) \quad (10.258)$$

21

22 Finally we perform a damped update of the average likelihood $\tilde{f}(\boldsymbol{\theta})$ using this new local factor:
23

24

$$\tilde{f}_{\text{new}}(\boldsymbol{\theta}) = \tilde{f}_{\text{old}}(\boldsymbol{\theta})^{1-1/N} \tilde{f}_n(\boldsymbol{\theta})^{1/N} \quad (10.259)$$

25

26 The ADF algorithm is similar to SEP, in that we compute the tilted distribution $q_{\setminus t} \propto f_t q_{t-1}$ and
27 then project it, without needing to keep the f_t factors. The difference is that instead of using the
28 cavity distribution $q_{-1}(\boldsymbol{\theta})$ as prior, it uses the posterior from the previous time step, q_{t-1} . This
29 avoids the need to compute and store \tilde{f} , but results in overconfidence in the batch setting.
30

31 10.7.8 Applications

32 EP has been used for a variety of problems, such as the following.
33

- 34 • The **TrueSkill** system from Microsoft, [HMG07], which is a system to rank players of the Xbox
35 360 system.
- 36 • Approximate inference in Gaussian processes with non-Gaussian likelihoods (see Section 18.4),
37 where [NR08] showed good results.
- 38 • The **probabilistic backpropagation** (PBP) algorithm of [HLA15a], which performs approximate
39 Bayesian inference for DNNs (see Section 17.6.2).
- 40 • Group sparse regression using spike-and-slab priors (Section 15.2.4), where [HLHLD13] showed
41 EP worked well.
- 42
- 43
- 44
- 45
- 46
- 47

11 Monte Carlo inference

11.1 Introduction

In this chapter, we discuss **Monte Carlo methods**, which are a stochastic approach to solving numerical integration problems. The name refers to the “Monte Carlo” casino in Monaco; this was used as a codename by von Neumann and Ulam, who invented the technique while working on the atomic bomb during WWII. Since then, the technique has become widely adopted in physics, statistics, machine learning, and many areas of science and engineering.

In this chapter, we give a brief introduction to some key concepts. In Chapter 12, we discuss MCMC, which is the most widely used MC method for high-dimensional problems. In Chapter 13, we discuss SMC, which is widely used for MC inference in state space models, but can also be applied more generally. For more details on MC methods, see e.g., [Liu01; RC04; KTB11; BZ20].

11.2 Monte Carlo integration

We often want to compute the expected value of some function of a random variable, $\mathbb{E}[f(\mathbf{X})]$. This requires computing the following integral:

$$\mathbb{E}[f(\mathbf{x})] = \int f(\mathbf{x})p(\mathbf{x})d\mathbf{x} \tag{11.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and $p(\mathbf{x})$ is the target distribution of \mathbf{X} .¹ In low dimensions (up to, say, 3), we can compute the above integral efficiently using **numerical integration**, which (adaptively) compute a grid, and then evaluate the function at each point on the grid.² But this does not scale to higher dimensions.

An alternative approach is to draw multiple random samples, $\mathbf{x}_n \sim p(\mathbf{x})$, and then to compute

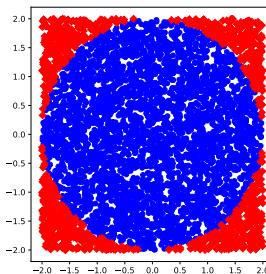
$$\mathbb{E}[f(\mathbf{x})] \approx \frac{1}{N_s} \sum_{n=1}^{N_s} f(\mathbf{x}_n) \tag{11.2}$$

This is called **Monte Carlo integration**. It has the advantage over numerical integration that the function is only evaluated in places where there is non-negligible probability, so it does not

1. In many cases, the target distribution may be the posterior $p(\mathbf{x}|\mathbf{y})$, which can be hard to compute; in such problems, we often work with the unnormalized distribution, $\tilde{p}(\mathbf{x}) = p(\mathbf{x}, \mathbf{y})$, instead, and then normalize the results using $Z = \int p(\mathbf{x}, \mathbf{y})d\mathbf{x} = p(\mathbf{y})$.

2. In 1d, numerical integration is called **quadrature**; in higher dimensions, it is called **cubature** [Sar13].

1
2
3
4
5
6
7
8
9
10



11 *Figure 11.1: Estimating π by Monte Carlo integration using 5000 samples. Blue points are inside the circle,*
12 *red points are outside. Generated by `mc_estimate_pi.py`.*

13

14

15 need to uniformly cover the entire space. In particular, it can be shown that the accuracy is in
16 principle independent of the dimensionality of \mathbf{x} , and only depends on the number of samples N_s
17 (see Section 11.2.2 for details). The catch is that we need a way to generate the samples $\mathbf{x}_n \sim p(\mathbf{x})$
18 in the first place. In addition, the estimator may have high variance. We will discuss this topic at
19 length in the sections below.

20

21 11.2.1 Example: estimating π by Monte Carlo integration

22

23 MC integration can be used for many applications, not just in ML and statistics. For example,
24 suppose we want to estimate π . We know that the area of a circle with radius r is πr^2 , but it is also
25 equal to the following definite integral:

$$26 \quad I = \int_{-r}^r \int_{-r}^r \mathbb{I}(x^2 + y^2 \leq r^2) dx dy \quad (11.3)$$

27 Hence $\pi = I/(r^2)$. Let us approximate this by Monte Carlo integration. Let $f(x, y) = \mathbb{I}(x^2 + y^2 \leq r^2)$
28 be an indicator function that is 1 for points inside the circle, and 0 outside, and let $p(x)$ and $p(y)$ be
29 uniform distributions on $[-r, r]$, so $p(x) = p(y) = 1/(2r)$. Then

$$30 \quad I = (2r)(2r) \int \int f(x, y) p(x) p(y) dx dy \quad (11.4)$$

$$31 \quad = 4r^2 \int \int f(x, y) p(x) p(y) dx dy \quad (11.5)$$

$$32 \quad \approx 4r^2 \frac{1}{N_s} \sum_{n=1}^{N_s} f(x_n, y_n) \quad (11.6)$$

33 Using 5000 samples, we find $\hat{\pi} = 3.10$ with standard error 0.09 compared to the true value of $\pi = 3.14$.

34 We can plot the points that are accepted or rejected as in Figure 11.1.

35

36 11.2.2 Accuracy of Monte Carlo integration

37

38 The accuracy of an MC approximation increases with sample size. This is illustrated in Figure 11.2.

39 On the top line, we plot a histogram of samples from a Gaussian distribution. On the bottom line,

40

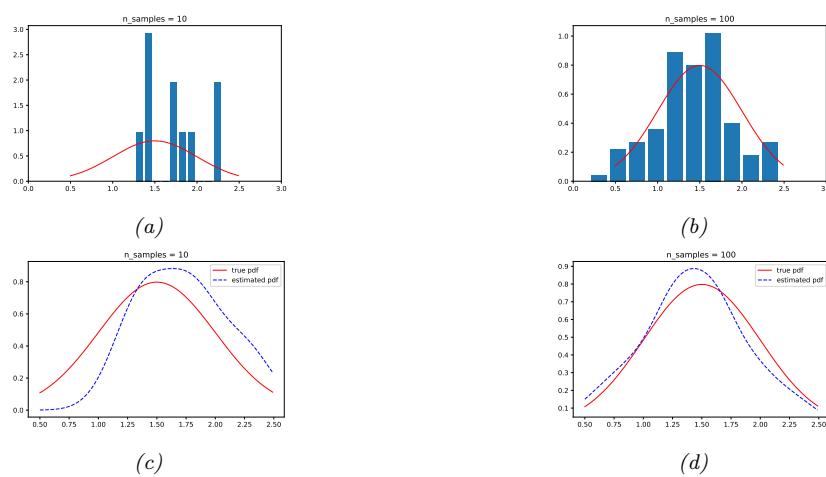


Figure 11.2: 10 and 100 samples from a Gaussian distribution, $\mathcal{N}(\mu = 1.5, \sigma^2 = 0.25)$. Solid red line is true pdf. Top row: histogram of samples. Bottom row: kernel density estimate derived from the samples. Generated by `mc_accuracy_demo.py`.

we plot a smoothed version of these samples, created using a kernel density estimate. This smoothed distribution is then evaluated on a dense grid of points and plotted. Note that this smoothing is just for the purposes of plotting, it is not used for the Monte Carlo estimate itself.

If we denote the exact mean by $\mu = \mathbb{E}[f(X)]$, and the MC approximation by $\hat{\mu}$, one can show that, with independent samples,

$$(\hat{\mu} - \mu) \rightarrow \mathcal{N}\left(0, \frac{\sigma^2}{N_s}\right) \quad (11.7)$$

where

$$\sigma^2 = \mathbb{V}[f(X)] = \mathbb{E}[f(X)^2] - \mathbb{E}[f(X)]^2 \quad (11.8)$$

This is a consequence of the central limit theorem. Of course, σ^2 is unknown in the above expression, but it can be estimated by MC:

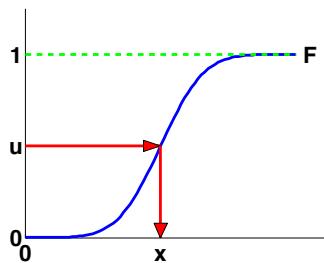
$$\hat{\sigma}^2 = \frac{1}{N_s} \sum_{n=1}^{N_s} (f(x_n) - \hat{\mu})^2 \quad (11.9)$$

Thus for large enough N_s we have

$$P\left\{\hat{\mu} - 1.96 \frac{\hat{\sigma}}{\sqrt{N_s}} \leq \mu \leq \hat{\mu} + 1.96 \frac{\hat{\sigma}}{\sqrt{N_s}}\right\} \approx 0.95 \quad (11.10)$$

The term $\sqrt{\frac{\hat{\sigma}^2}{N_s}}$ is called the (numerical or empirical) **standard error**, and is an estimate of our uncertainty about our estimate of μ .

1
2
3
4
5
6
7
8
9
10



11
12

Figure 11.3: Sampling using an inverse CDF.

13

14

If we want to report an answer which is accurate to within $\pm\epsilon$ with probability at least 95%, we need to use a number of samples N_s which satisfies $1.96\sqrt{\hat{\sigma}^2/N_s} \leq \epsilon$. We can approximate the 1.96 factor by 2, yielding $S \geq \frac{4\hat{\sigma}^2}{\epsilon^2}$.

The remarkable thing to note about the above results is that the error in the estimate, σ^2/S , is theoretically independent of the dimensionality of the integral. The catch is that sampling from high dimensional distributions can be hard. We turn to that topic next.

21

22 11.3 Generating random samples from simple distributions

23

We saw in Section 11.2 how we can evaluate $\mathbb{E}[f(X)]$ for different functions f of a random variable X using Monte Carlo integration. The main computational challenge is to efficiently generate samples from the probability distribution $p^*(\mathbf{x})$ (which may be a posterior, $p^*(\mathbf{x}) \propto p(\mathbf{x}|\mathcal{D})$). In this section, we discuss sampling methods that are suitable for parametric univariate distributions. These can be used as building blocks for sampling from more complex multivariate distributions.

29

30 11.3.1 Sampling using the inverse cdf

31

The simplest method for sampling from a univariate distribution is based on the **inverse probability transform**. Let F be a cdf of some distribution we want to sample from, and let F^{-1} be its inverse. Then we have the following result.

35

Theorem 11.3.1. If $U \sim U(0, 1)$ is a uniform rv, then $F^{-1}(U) \sim F$.

37 *Proof.*

38

$$39 \quad \Pr(F^{-1}(U) \leq x) = \Pr(U \leq F(x)) \quad (\text{applying } F \text{ to both sides}) \quad (11.11)$$

$$40 \quad = F(x) \quad (\text{because } \Pr(U \leq y) = y) \quad (11.12)$$

42 where the first line follows since F is a monotonic function, and the second line follows since U is 43 uniform on the unit interval. \square

44

Hence we can sample from any univariate distribution, for which we can evaluate its inverse cdf, as 46 follows: generate a random number $u \sim U(0, 1)$ using a **pseudo random number generator** (see 47

e.g., [Pre+88] for details). Let u represent the height up the y axis. Then “slide along” the x axis until you intersect the F curve, and then “drop down” and return the corresponding x value. This corresponds to computing $x = F^{-1}(u)$. See Figure 11.3 for an illustration.

For example, consider the exponential distribution

$$\text{Expon}(x|\lambda) \triangleq \lambda e^{-\lambda x} \mathbb{I}(x \geq 0) \quad (11.13)$$

The cdf is

$$F(x) = 1 - e^{-\lambda x} \mathbb{I}(x \geq 0) \quad (11.14)$$

whose inverse is the quantile function

$$F^{-1}(p) = -\frac{\ln(1-p)}{\lambda} \quad (11.15)$$

By the above theorem, if $U \sim \text{Unif}(0, 1)$, we know that $F^{-1}(U) \sim \text{Expon}(\lambda)$. So we can sample from the exponential distribution by first sampling from the uniform and then transforming the results using $-\ln(1-u)/\lambda$. (In fact, since $1-U \sim \text{Unif}(0, 1)$, we can just use $-\ln(u)/\lambda$.)

11.3.2 Sampling from a Gaussian (Box-Muller method)

In this section, we describe a method to sample from a Gaussian. The idea is we sample uniformly from a unit radius circle, and then use the change of variables formula to derive samples from a spherical 2d Gaussian. This can be thought of as two samples from a 1d Gaussian.

In more detail, sample $z_1, z_2 \in (-1, 1)$ uniformly, and then discard pairs that do not satisfy $z_1^2 + z_2^2 \leq 1$. The result will be points uniformly distributed inside the unit circle, so $p(\mathbf{z}) = \frac{1}{\pi} \mathbb{I}(z \text{ inside circle})$.

Now define

$$x_i = z_i \left(\frac{-2 \ln r^2}{r^2} \right)^{\frac{1}{2}} \quad (11.16)$$

for $i = 1 : 2$, where $r^2 = z_1^2 + z_2^2$. Using the multivariate change of variables formula, we have

$$p(x_1, x_2) = p(z_1, z_2) \left| \frac{\partial(z_1, z_2)}{\partial(x_1, x_2)} \right| = \left[\frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x_1^2) \right] \left[\frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x_2^2) \right] \quad (11.17)$$

Hence x_1 and x_2 are two independent samples from a univariate Gaussian. This is known as the **Box-Muller** method.

To sample from a multivariate Gaussian, we first compute the Cholesky decomposition of its covariance matrix, $\Sigma = \mathbf{L}\mathbf{L}^\top$, where \mathbf{L} is lower triangular. Next we sample $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ using the Box-Muller method. Finally we set $\mathbf{y} = \mathbf{L}\mathbf{x} + \boldsymbol{\mu}$. This is valid since

$$\text{Cov}[\mathbf{y}] = \mathbf{L}\text{Cov}[\mathbf{x}]\mathbf{L}^\top = \mathbf{L} \mathbf{I} \mathbf{L}^\top = \Sigma \quad (11.18)$$

11.4 Rejection sampling

Suppose we want to sample from the **target distribution**

$$p(\mathbf{x}) = \tilde{p}(\mathbf{x})/Z_p \quad (11.19)$$

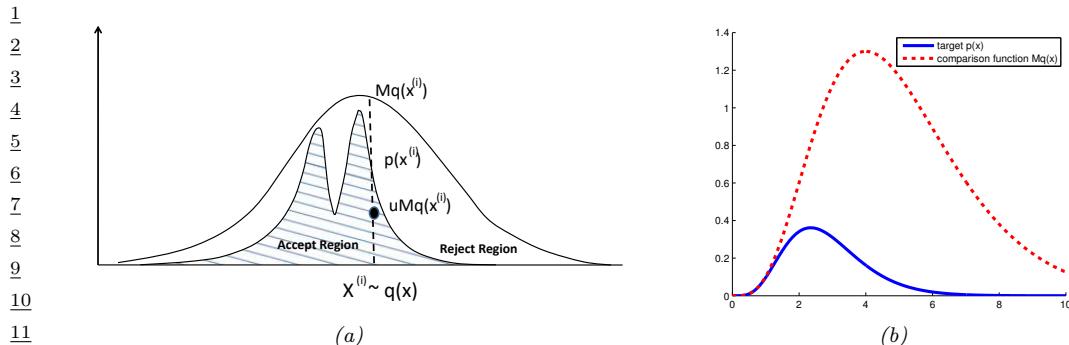


Figure 11.4: (a) Schematic illustration of rejection sampling. From Figure 2 of [And+03]. Used with kind permission of Nando de Freitas. (b) Rejection sampling from a $\text{Ga}(\alpha = 5.7, \lambda = 2)$ distribution (solid blue) using a proposal of the form $M\text{Ga}(k, \lambda - 1)$ (dotted red), where $k = \lfloor 5.7 \rfloor = 5$. The curves touch at $\alpha - k = 0.7$. Generated by [rejection_sampling_demo.py](#).

where $\tilde{p}(\mathbf{x})$ is the unnormalized version, and

$$Z_p = \int \tilde{p}(\mathbf{x}) d\mathbf{x} \quad (11.20)$$

is the (possibly unknown) normalization constant. One of the simplest approaches to this problem is **rejection sampling**, which we now explain.

11.4.1 Basic idea

In rejection sampling, we require access to a **proposal distribution** $q(\mathbf{x})$ which satisfies $Cq(\mathbf{x}) \geq \tilde{p}(\mathbf{x})$, for some constant C . The function $Cq(\mathbf{x})$ provides an upper envelope for \tilde{p} .

We can use the proposal distribution to generate samples from the target distribution as follows. We first sample $\mathbf{x}_0 \sim q(\mathbf{x})$, which corresponds to picking a random \mathbf{x} location, and then we sample $u_0 \sim \text{Unif}(0, Cq(\mathbf{x}_0))$, which corresponds to picking a random height (y location) under the envelope. If $u_0 > \tilde{p}(\mathbf{x}_0)$, we reject the sample, otherwise we accept it. This process is illustrated in 1d in Figure 11.4(a): the acceptance region is shown shaded, and the rejection region is the white region between the shaded zone and the upper envelope.

We now prove this procedure is correct. First note that the probability of any given sample \mathbf{x}_0 being accepted equals the probability of a sample $u_0 \sim \text{Unif}(0, Cq(\mathbf{x}_0))$ being less than or equal to $\tilde{p}(\mathbf{x}_0)$, i.e.,

$$q(\text{accept}|\mathbf{x}_0) = \int_0^{\tilde{p}(\mathbf{x}_0)} \frac{1}{Cq(\mathbf{x}_0)} du = \frac{\tilde{p}(\mathbf{x}_0)}{Cq(\mathbf{x}_0)} \quad (11.21)$$

Therefore

$$q(\text{propose and accept } \mathbf{x}_0) = q(\mathbf{x}_0)q(\text{accept}|\mathbf{x}_0) = q(\mathbf{x}_0) \frac{\tilde{p}(\mathbf{x}_0)}{Cq(\mathbf{x}_0)} = \frac{\tilde{p}(\mathbf{x}_0)}{C} \quad (11.22)$$

1
2 Integrating both sides give

3
4 $\int q(\mathbf{x}_0)q(\text{accept}|\mathbf{x}_0) d\mathbf{x}_0 = q(\text{accept}) = \frac{\int \tilde{p}(\mathbf{x}_0) d\mathbf{x}_0}{C} = \frac{Z_p}{C}$ (11.23)
5

6 Hence we see that the distribution of accepted points is given by the target distribution:

7
8 $q(\mathbf{x}_0|\text{accept}) = \frac{q(\mathbf{x}_0, \text{accept})}{q(\text{accept})} = \frac{\tilde{p}(\mathbf{x}_0)}{C} \frac{C}{Z_p} = \frac{\tilde{p}(\mathbf{x}_0)}{Z_p} = p(\mathbf{x}_0)$ (11.24)
9

10 How efficient is this method? If \tilde{p} is a normalized target distribution, the acceptance probability is
11 $1/C$. Hence we want to choose C as small as possible while still satisfying $Cq(x) \geq \tilde{p}(x)$.

13 11.4.2 Example

15 For example, suppose we want to sample from a Gamma distribution:³

16
17 $\text{Ga}(x|\alpha, \lambda) = \frac{1}{\Gamma(\alpha)} x^{\alpha-1} \lambda^\alpha \exp(-\lambda x)$ (11.25)
18

19
20 where $\Gamma(\alpha)$ is the gamma function. One can show that if $X_i \stackrel{iid}{\sim} \text{Expon}(\lambda)$, and $Y = X_1 + \dots + X_k$,
21 then $Y \sim \text{Ga}(k, \lambda)$. For non-integer shape parameters α , we cannot use this trick. However, we can
22 use rejection sampling using a $\text{Ga}(k, \lambda - 1)$ distribution as a proposal, where $k = \lfloor \alpha \rfloor$. The ratio has
23 the form

24
25 $\frac{p(x)}{q(x)} = \frac{\text{Ga}(x|\alpha, \lambda)}{\text{Ga}(x|k, \lambda - 1)} = \frac{x^{\alpha-1} \lambda^\alpha \exp(-\lambda x) / \Gamma(\alpha)}{x^{k-1} (\lambda - 1)^k \exp(-(\lambda - 1)x) / \Gamma(k)}$ (11.26)
26

27
28 $= \frac{\Gamma(k) \lambda^\alpha}{\Gamma(\alpha) (\lambda - 1)^k} x^{\alpha-k} \exp(-x)$ (11.27)

29 This ratio attains its maximum when $x = \alpha - k$. Hence

30
31 $C = \frac{\text{Ga}(\alpha - k|\alpha, \lambda)}{\text{Ga}(\alpha - k|k, \lambda - 1)}$ (11.28)
32

33 See Figure 11.4(b) for a plot.

36 11.4.3 Adaptive rejection sampling

37 We now describe a method that can automatically come up with a tight upper envelope $q(x)$ to
38 any log concave 1d density $p(x)$. The idea is to upper bound the log density with a piecewise linear
39 function, as illustrated in Figure 11.5(a). We choose the initial locations for the pieces based on a
40 fixed grid over the support of the distribution. We then evaluate the gradient of the log density at
41 these locations, and make the lines be tangent at these points.

42 Since the log of the envelope is piecewise linear, the envelope itself is piecewise exponential:

43
44 $q(x) = C_i \lambda_i \exp(-\lambda_i(x - x_{i-1})), \quad x_{i-1} < x \leq x_i$ (11.29)
45

46 3. This section is based on notes by Ioana A. Cosma, available at <http://users.aims.ac.za/~ioana/cp2.pdf>.

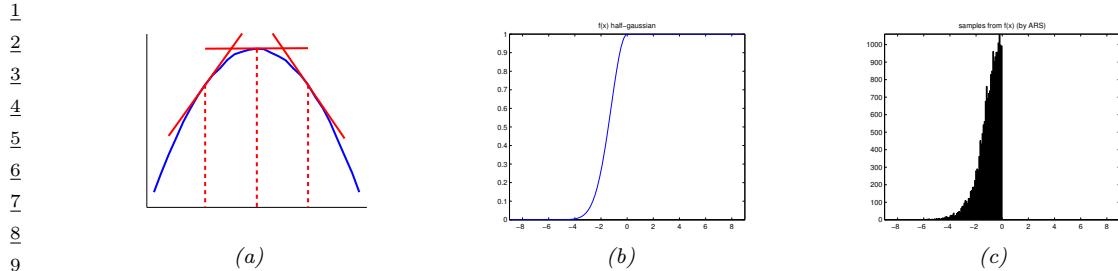


Figure 11.5: (a) Idea behind adaptive rejection sampling. We place piecewise linear upper (and lower) bounds on the log-concave density. Adapted from Figure 1 of [GW92]. Generated by `ars_envelope.py`. (b-c) Using ARS to sample from a half-Gaussian. Generated by `ars_demo.py`.

13

14

15 where x_i are the grid points. It is relatively straightforward to sample from this distribution. If the
16 sample x is rejected, we create a new grid point at x , and thereby refine the envelope. As the number
17 of grid points is increased, the tightness of the envelope improves, and the rejection rate goes down.
18 This is known as **adaptive rejection sampling** (ARS) [GW92]. Figure 11.5(b-c) gives an example
19 of the method in action. As with standard rejection sampling, it can be applied to unnormalized
20 distributions.

21

22 11.4.4 Rejection sampling in high dimensions

23

24 It is clear that we want to make our proposal $q(\mathbf{x})$ as close as possible to the target distribution $p(\mathbf{x})$,
25 while still being an upper bound. But this is quite hard to achieve, especially in high dimensions. To
26 see this, consider sampling from $p(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \sigma_p^2 \mathbf{I})$ using as a proposal $q(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \sigma_q^2 \mathbf{I})$. Obviously
27 we must have $\sigma_q^2 \geq \sigma_p^2$ in order to be an upper bound. In D dimensions, the optimum value is given
28 by $C = (\sigma_q/\sigma_p)^D$. The acceptance rate is $1/C$ (since both p and q are normalized), which decreases
29 exponentially fast with dimension. For example, if σ_q exceeds σ_p by just 1%, then in 1000 dimensions
30 the acceptance ratio will be about 1/20,000. This is a fundamental weakness of rejection sampling.

31

32 11.5 Importance sampling

33

34 In this section, we describe a Monte Carlo method known as **importance sampling** for approxi-
35 mating integrals of the form

36

$$37 \quad \mathbb{E}[\varphi(\mathbf{x})] = \int \varphi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} \quad (11.30)$$

38

39 where φ is called a **target function**, and $\pi(\mathbf{x})$ is the **target distribution**, often a conditional
40 distribution of the form $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$. Since in general it is difficult to draw from the target
41 distribution, we will instead draw from some **proposal distribution** $q(\mathbf{x})$ (which will usually
42 depend on \mathbf{y}). We then adjust for the inaccuracies of this by associating weights with each sample,
43 so we end up with a weighted MC approximation:

44

$$45 \quad \mathbb{E}[\varphi(\mathbf{x})] \approx \sum_{n=1}^N W_n \varphi(\mathbf{x}_n) \quad (11.31)$$

46

47

We discuss two cases, first when the target is normalized, and then when it is unnormalized. This will affect the ways the weights are computed, as well as statistical properties of the estimator.

11.5.1 Direct importance sampling

In this section, we assume that we can *evaluate* the normalized target distribution $\pi(\mathbf{x})$, but we cannot sample from it. So instead we will sample from the proposal $q(\mathbf{x})$. We can then write

$$\int \varphi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \int \varphi(\mathbf{x})\frac{\pi(\mathbf{x})}{q(\mathbf{x})}q(\mathbf{x})d\mathbf{x} \quad (11.32)$$

We require that the proposal be non-zero whenever the target is non-zero, i.e., the support of $q(\mathbf{x})$ needs to be greater or equal to the support of $\pi(\mathbf{x})$. If we draw N_s samples $\mathbf{x}_n \sim q(\mathbf{x})$, we can write

$$\mathbb{E}[\varphi(\mathbf{x})] \approx \frac{1}{N_s} \sum_{n=1}^{N_s} \frac{\pi(\mathbf{x}_n)}{q(\mathbf{x}_n)} \varphi(\mathbf{x}_n) = \frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \varphi(\mathbf{x}_n) \quad (11.33)$$

where we have defined the **importance weights** as follows:

$$\tilde{w}_n = \frac{\pi(\mathbf{x}_n)}{q(\mathbf{x}_n)} \quad (11.34)$$

The result is an unbiased estimate of the true mean $\mathbb{E}[\varphi(\mathbf{x})]$.

11.5.2 Self-normalized importance sampling

The disadvantage of direct importance sampling is that we need a way to evaluate the normalized target distribution π in order to compute the weights. It is often much easier to evaluate the **unnormalized target distribution**

$$\tilde{\gamma}(\mathbf{x}) = Z\pi(\mathbf{x}) \quad (11.35)$$

where

$$Z = \int \tilde{\gamma}(\mathbf{x})d\mathbf{x} \quad (11.36)$$

is the normalization constant. (For example, if $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, then $\tilde{\gamma}(\mathbf{x}) = p(\mathbf{x}, \mathbf{y})$ and $Z = p(\mathbf{y})$). The key idea is to also approximate the normalization constant Z with importance sampling. This method is called **self-normalized importance sampling**. The resulting estimate is a ratio of two estimates, and hence is biased. However as $N_s \rightarrow \infty$, the bias goes to zero, under some weak assumptions (see e.g., [RC04] for details).

In more detail, SNIS is based on this approximation:

$$\mathbb{E}[\varphi(\mathbf{x})] = \int \varphi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \frac{\int \varphi(\mathbf{x})\tilde{\gamma}(\mathbf{x})d\mathbf{x}}{\int \tilde{\gamma}(\mathbf{x})d\mathbf{x}} = \frac{\int \left[\frac{\tilde{\gamma}(\mathbf{x})}{q(\mathbf{x})} \right] \varphi(\mathbf{x})d\mathbf{x}}{\int \left[\frac{\tilde{\gamma}(\mathbf{x})}{q(\mathbf{x})} \right] q(\mathbf{x})d\mathbf{x}} \quad (11.37)$$

$$\approx \frac{\frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \varphi(\mathbf{x}_n)}{\frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n} \quad (11.38)$$

1 were we have defined the **unnormalized weights**

2

$$\tilde{w}_n = \frac{\tilde{\gamma}(\mathbf{x})}{q(\mathbf{x})} \quad (11.39)$$

3

4 We can write Equation (11.38) more compactly as

5

6

$$\mathbb{E}[\varphi(\mathbf{x})] \approx \sum_{n=1}^{N_s} W_n \varphi(\mathbf{x}_n) \quad (11.40)$$

7

8 where we have defined the **normalized weights** by

9

10

$$W_n = \frac{\tilde{w}_n}{\sum_{n'=1}^{N_s} \tilde{w}_{n'}} \quad (11.41)$$

11

12 This is equivalent to approximating the target distribution using a weighted sum of delta functions:

13

14

$$\pi(\mathbf{x}) \approx \sum_{n=1}^{N_s} W_n \delta(\mathbf{x} - \mathbf{x}_n) \triangleq \hat{\pi}(\mathbf{x}) \quad (11.42)$$

15

16 As a byproduct of this algorithm we get the following appoximation to the normalization constant:

17

18

$$Z \approx \frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \triangleq \hat{Z} \quad (11.43)$$

19

20 11.5.3 Choosing the proposal

21

22 The performance of importance sampling depends crucially on the quality of the proposal distribution.

23 As we mentioned, we require that the support of q cover the support of the target (i.e., $\tilde{\gamma}(\mathbf{x}) > 0 \implies$

24 $q(\mathbf{x}) > 0$). However, we also want the proposal to not be too “loose” or a “covering”. Ideally it should

25 also take into account properties of the target function φ as well, as shown in Figure 11.6. This can

26 yield subsantial benefits, as shown in the “**target aware Bayesian inference**” scheme of [Rai+20].

27 However, usually the target function φ is unknown or ignored, so we just try to find a “generally

28 useful” approximation to the target.

29 One way to come up with a good proposal is to learn one, by optimizing the variational lower

30 bound or ELBO (see Section 10.1.2). Indeed, if we fix the parameters of the generative model, we

31 can think of importance weighted autoencoders (Section 10.5.1) as learning a good IS proposal. More

32 details on this connection can be found in [DS18].

33

34

35 11.5.4 Annealed importance sampling (AIS)

36

37 In this section, we describe a method known as **annealed importance sampling** [Nea01] for

38 sampling from complex, possibly multimodal distributions. Assume we want to sample from some

39 target distribution $p_0(\mathbf{x}) \propto f_0(\mathbf{x})$ (where $f_0(\mathbf{x})$ is the unnormalized version), but we cannot do

40 so easily. However, suppose that there is an easier distribution which we *can* sample from, call it

41

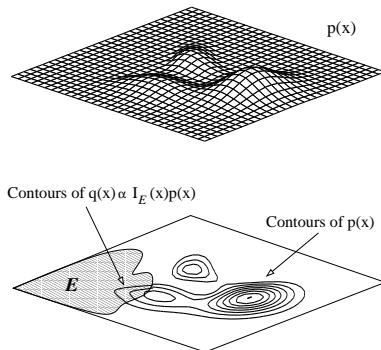


Figure 11.6: In importance sampling, we should sample from a distribution that takes into account regions where $\pi(\mathbf{x})$ has high probability and where $\varphi(\mathbf{x})$ is large. Here the function to be evaluated is an indicator function of a set, corresponding to a set of rare events in the tail of the distribution. From Figure 3 of [And+03]. Used with kind permission of Nando de Freitas.

$p_n(\mathbf{x}) \propto f_n(\mathbf{x})$; for example, this might be the prior. We now construct a sequence of intermediate distributions than move slowly from p_n to p_0 as follows:

$$f_j(\mathbf{x}) = f_0(\mathbf{x})^{\beta_j} f_n(\mathbf{x})^{1-\beta_j} \quad (11.44)$$

where $1 = \beta_0 > \beta_1 > \dots > \beta_n = 0$, where β_j is an inverse temperature. We will sample a set of points from f_n , and then from f_{n-1} , and so on, until we eventually sample from f_0 .

To sample from each f_j , suppose we can define a Markov chain $T_j(\mathbf{x}, \mathbf{x}') = p_j(\mathbf{x}'|\mathbf{x})$, which leaves p_0 invariant (i.e., $\int p_j(\mathbf{x}'|\mathbf{x})p_0(\mathbf{x})d\mathbf{x} = p_0(\mathbf{x}')$). (See Chapter 12 for details on how to construct such chains.) Given this, we can sample \mathbf{x} from p_0 as follows: sample $\mathbf{v}_n \sim p_n$; sample $\mathbf{v}_{n-1} \sim T_{n-1}(\mathbf{v}_n, \cdot)$; and continue in this way until we sample $\mathbf{v}_0 \sim T_0(\mathbf{v}_1, \cdot)$; finally we set $\mathbf{x} = \mathbf{v}_0$ and give it weight

$$w = \frac{f_{n-1}(\mathbf{v}_{n-1})}{f_n(\mathbf{v}_{n-1})} \frac{f_{n-2}(\mathbf{v}_{n-2})}{f_{n-1}(\mathbf{v}_{n-2})} \dots \frac{f_1(\mathbf{v}_1)}{f_2(\mathbf{v}_1)} \frac{f_0(\mathbf{v}_0)}{f_1(\mathbf{v}_0)} \quad (11.45)$$

This can be shown to be correct by viewing the algorithm as a form of importance sampling in an extended state space $\mathbf{v} = (\mathbf{v}_0, \dots, \mathbf{v}_n)$. Consider the following distribution on this state space:

$$p(\mathbf{v}) \propto \varphi(\mathbf{v}) = f_0(\mathbf{v}_0)\tilde{T}_0(\mathbf{v}_0, \mathbf{v}_1)\tilde{T}_2(\mathbf{v}_1, \mathbf{v}_2) \dots \tilde{T}_{n-1}(\mathbf{v}_{n-1}, \mathbf{v}_n) \quad (11.46)$$

$$\propto p(\mathbf{v}_0)p(\mathbf{v}_1|\mathbf{v}_0) \dots p(\mathbf{v}_n|\mathbf{v}_{n-1}) \quad (11.47)$$

where \tilde{T}_j is the reversal of T_j :

$$\tilde{T}_j(\mathbf{v}, \mathbf{v}') = T_j(\mathbf{v}', \mathbf{v})p_j(\mathbf{v}')/p_j(\mathbf{v}) = T_j(\mathbf{v}', \mathbf{v})f_j(\mathbf{v}')/f_j(\mathbf{v}) \quad (11.48)$$

It is clear that $\sum_{\mathbf{v}_1, \dots, \mathbf{v}_n} \varphi(\mathbf{v}) = f_0(\mathbf{v}_0)$, so by sampling from $p(\mathbf{v})$, we can effectively sample from $p_0(\mathbf{x})$.

We can sample on this extended state space using the above algorithm, which corresponds to the following proposal:

$$q(\mathbf{v}) \propto g(\mathbf{v}) = f_n(\mathbf{v}_n)T_{n-1}(\mathbf{v}_n, \mathbf{v}_{n-1}) \cdots T_2(\mathbf{v}_2, \mathbf{v}_1)T_0(\mathbf{v}_1, \mathbf{v}_0) \quad (11.49)$$

$$\propto p(\mathbf{v}_n)p(\mathbf{v}_{n-1}|\mathbf{v}_n) \cdots p(\mathbf{v}_1|\mathbf{v}_0) \quad (11.50)$$

One can show that the importance weights $w = \frac{\varphi(\mathbf{v}_0, \dots, \mathbf{v}_n)}{g(\mathbf{v}_0, \dots, \mathbf{v}_n)}$ are given by Equation (11.45). Since marginals of the sampled sequences from this extended model are equivalent to samples from $p_0(\mathbf{x})$, we see that we are using the correct weights.

11.5.4.1 Estimating normalizing constants using AIS

An important application of AIS is to evaluate a ratio of partition functions. Notice that $Z_0 = \int f_0(\mathbf{x})d\mathbf{x} = \int \varphi(\mathbf{v})d\mathbf{v}$, and $Z_n = \int f_n(\mathbf{x})d\mathbf{x} = \int g(\mathbf{v})d\mathbf{v}$. Hence

$$\frac{Z_0}{Z_n} = \frac{\int \varphi(\mathbf{v})d\mathbf{v}}{\int g(\mathbf{v})d\mathbf{v}} = \frac{\int \frac{\varphi(\mathbf{v})}{g(\mathbf{v})}g(\mathbf{v})d\mathbf{v}}{\int g(\mathbf{v})d\mathbf{v}} = \mathbb{E}_g \left[\frac{\varphi(\mathbf{v})}{g(\mathbf{v})} \right] \approx \frac{1}{S} \sum_{s=1}^S w_s \quad (11.51)$$

where $w_s = \varphi(\mathbf{v}_s)/g(\mathbf{v}_s)$. If f_0 is a prior and f_n is the posterior, we can estimate $Z_n = p(\mathcal{D})$ using the above equation, provided the prior has a known normalization constant Z_0 . This is generally considered the method of choice for evaluating difficult partition functions.

23

24

11.6 Controlling Monte Carlo variance

27 As we mentioned in Section 11.2.2, the standard error in a Monte Carlo estimate is $O(1/\sqrt{S})$, where
28 S is the number of (independent) samples. Consequently it may take many samples to reduce the
29 variance to a sufficiently small value. In this section, we discuss some ways to reduce the variance of
30 sampling methods. For more details, see e.g., [KTB11].
31

11.6.1 Rao-Blackwellisation

32 In this section, we discuss a useful technique for reducing the variance of MC estimators known as
33 **Rao-Blackwellisation**. To explain the method, suppose we have two rv's, X and Y , and we want
36 to estimate $\bar{f} = \mathbb{E}[f(X, Y)]$. The naive approach is to use an MC approximation
37

$$\hat{f}_{MC} = \frac{1}{S} \sum_{s=1}^S f(X_s, Y_s) \quad (11.52)$$

41 where $(X_s, Y_s) \sim p(X, Y)$. This is an unbiased estimator of \bar{f} . However, it may have high variance.
42

43 Now suppose we can analytically marginalize out Y , provided we know X , i.e., we can tractable
44 compute

$$45 \quad f_X(X_s) = \int dY p(Y|X_s) f(X_s, Y) = \mathbb{E}[f(X, Y)|X = X_s] \quad (11.53)$$

47

1
2 Let us define the Rao-Blackwellised estimator

3
4 $\hat{f}_{RB} = \frac{1}{S} \sum_{s=1}^S f_X(X_s)$ (11.54)
5

6
7 where $X_s \sim p(X)$. This is an unbiased estimator, since $\mathbb{E} [\hat{f}_{RB}] = \mathbb{E} [\mathbb{E} [f(X, Y) | X]] = \bar{f}$. However,
8 this estimate can have lower variance than the naive estimator. The intuitive reason is that we are
9 now sampling in a reduced dimensional space. Formally we can see this by using the law of iterated
10 variance to get
11

12 $\mathbb{V} [\mathbb{E} [f(X, Y) | X]] = \mathbb{V} [f(X, Y)] - \mathbb{E} [\mathbb{V} [f(X, Y)] | X] \leq \mathbb{V} [f(X, Y)]$ (11.55)
13

14 For some examples of this in practice, see Section 6.6.3.2, Section 13.5, and Section 12.3.8.

16 11.6.2 Control variates

18 Suppose we want to estimate $\mu = \mathbb{E} [f(X)]$ using an unbiased estimator $m(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S m(x_s)$,
19 where $x_s \sim p(X)$ and $\mathbb{E} [m(X)] = \mu$. (We abuse notation slightly and use m to refer to a function of
20 a single random variable as well as a set of samples.) Now consider the alternative estimator
21

22 $m^*(\mathcal{X}) = m(\mathcal{X}) + c(b(\mathcal{X}) - \mathbb{E} [b(\mathcal{X})])$ (11.56)
23

24 This is called a **control variate**, and b is called a **baseline**. (Once again we abuse notation and use
25 $b(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S b(x_s)$ and $m^*(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S m^*(x_s)$.)

26 It is easy to see that $m^*(\mathcal{X})$ is an unbiased estimator, since $\mathbb{E} [m^*(X)] = \mathbb{E} [m(X)] = \mu$. However,
27 it can have lower variance, provided b is correlated with m . To see this, note that
28

29 $\mathbb{V} [m^*(X)] = \mathbb{V} [m(X)] + c^2 \mathbb{V} [b(X)] + 2c \text{Cov} [m(X), b(X)]$ (11.57)
30

31 By taking the derivative of $\mathbb{V} [m^*(X)]$ wrt c and setting to 0, we find that the optimal value is
32

33 $c^* = -\frac{\text{Cov} [m(X), b(X)]}{\mathbb{V} [b(X)]}$ (11.58)
34

36 The corresponding variance of the new estimator is now

38 $\mathbb{V} [m^*(X)] = \mathbb{V} [m(X)] - \frac{\text{Cov} [m(X), b(X)]^2}{\mathbb{V} [b(X)]} = (1 - \rho_{m,b}^2) \mathbb{V} [m(X)] \leq \mathbb{V} [m(X)]$ (11.59)
39

41 where $\rho_{m,b}^2$ is the correlation of the basic estimator and the baseline function. If we can ensure
42 this correlation is high, we can reduce the variance. Intuitively, the CV estimator is exploiting
43 information about the errors in the estimate of a known quantity, namely $\mathbb{E} [b(X)]$, to reduce the
44 errors in estimating the unknown quantity, namely μ .

45 We give a simple worked example in Section 11.6.2.1. See Section 10.3.1 for an example of this
46 technique applied to blackbox variational inference.

1
2 **11.6.2.1 Example**

3 We now give a simple worked example of control variates.⁴ Consider estimating $\mu = \mathbb{E}[f(X)]$ where
4 $f(X) = 1/(1+X)$ and $X \sim \text{Unif}(0, 1)$. The exact value is
5

$$\mu = \int_0^1 \frac{1}{1+x} dx = \ln 2 \approx 0.693 \quad (11.60)$$

6 The naive MC estimate, using S samples, is $m(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S f(x_s)$. Using $S = 1500$, we find
7 $\mathbb{E}[m(\mathcal{X})] = 0.6935$ with standard error $\text{se} = 0.0037$.

8 Now let us use $b(X) = 1 + X$ as a baseline, so $b(\mathcal{X}) = (1/S) \sum_s (1 + x_s)$. This has expectation
9 $\mathbb{E}[b(X)] = \int_0^1 (1+x) dx = \frac{3}{2}$. The control variate estimator is given by
10

$$m^*(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S f(x_s) + c \left(\frac{1}{S} \sum_{s=1}^S b(x_s) - \frac{3}{2} \right) \quad (11.61)$$

11 The optimal value can be estimated from the samples of $m(x_s)$ and $b(x_s)$, and plugging into
12 Equation (11.58) to get $c^* \approx 0.4773$. Using $S = 1500$, we find $\mathbb{E}[m^*(\mathcal{X})] = 0.6941$ and $\text{se} = 0.0007$.

13 See also Section 11.6.3.1, where we analyze this example using antithetic sampling.

14
15 **11.6.3 Antithetic sampling**

16 In this section, we discuss **antithetic sampling**, which is a simple way to reduce variance.⁵ Suppose
17 we want to estimate $\theta = \mathbb{E}[Y]$. Let Y_1 and Y_2 be two samples. An unbiased estimate of θ is given by
18 $\hat{\theta} = (Y_1 + Y_2)/2$. The variance of this estimate is

$$\mathbb{V}[\hat{\theta}] = \frac{\mathbb{V}[Y_1] + \mathbb{V}[Y_2] + 2\text{Cov}[Y_1, Y_2]}{4} \quad (11.62)$$

19 so the variance is reduced if $\text{Cov}[Y_1, Y_2] < 0$. So whenever we sample Y_1 , we should set Y_2 to be its
20 “opposite”, but with the same mean.

21 For example, suppose $Y \sim \text{Unif}(0, 1)$. If we let y_1, \dots, y_n be iid samples from $\text{Unif}(0, 1)$, then we
22 can define $y'_i = 1 - y_i$. The distribution of y'_i is still $\text{Unif}(0, 1)$, but $\text{Cov}[y_i, y'_i] < 1$.

23
24 **11.6.3.1 Example**

25 To see why this can be useful, consider the example from Section 11.6.2.1. Let $\hat{\mu}_{\text{mc}}$ be the classic MC
26 estimate using $2N$ samples from $\text{Unif}(0, 1)$, and let $\hat{\mu}_{\text{anti}}$ be the MC estimate using the above antithetic
27 sampling scheme applied to N base samples from $\text{Unif}(0, 1)$. The exact value is $\mu = \ln 2 \approx 0.6935$.
28 For the classical method, with $N = 750$, we find $\mathbb{E}[\hat{\mu}_{\text{mc}}] = 0.69365$ with a standard error of 0.0037.
29 For the antithetic method, we find $\mathbb{E}[\hat{\mu}_{\text{anti}}] = 0.6939$ with a standard error of 0.0007, which matches
30 the control variate method of Section 11.6.2.1.

31
32 4. The example is from https://en.wikipedia.org/wiki/Control_variates, with modified notation. See [control_variates.py](#) for some code.

33 5. Our presentation is based on https://en.wikipedia.org/wiki/Antithetic_variates. See [antithetic_sampling.py](#) for the code.

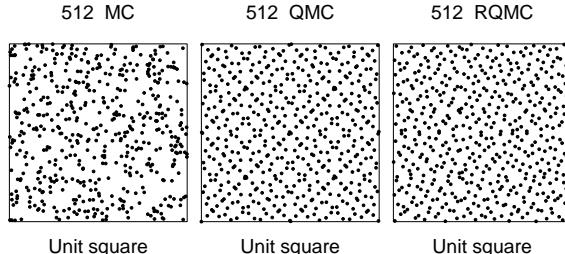


Figure 11.7: Illustration of Monte Carlo (MC), Quasi MC (QMC) from a Sobol sequence, and Randomized QMC using a scrambling method. Adapted from Figure 1 of [OR20]. Used with kind permission of Art Owen.

11.6.4 Quasi Monte Carlo (QMC)

Quasi Monte Carlo (see e.g., [Lem09; Owe13]) is an approach to numerical integration that replaces random samples with **low discrepancy sequences**, such as the **Halton sequence** (see e.g., [Owe17]) or **Sobol sequence**. Intuitively, these are **space filling** sequences of points, constructed to reduce the unwanted gaps and clusters that would arise among randomly chosen inputs. See Figure 11.7 for an example.⁶

More precisely, consider the problem of evaluating the following D -dimensional integral:

$$\bar{f} = \int_{[0,1]^D} f(\mathbf{x}) d\mathbf{x} \approx \hat{f}_N = \frac{1}{N} \sum_{n=1}^N f(\mathbf{x}_n) \quad (11.63)$$

Let $\epsilon_N = |\bar{f} - \hat{f}_N|$ be the error. In standard Monte Carlo, if we draw N independent samples, then we have $\epsilon_N \sim O\left(\frac{1}{\sqrt{N}}\right)$. In QMC, it can be shown that $\epsilon_N \sim O\left(\frac{(\log N)^D}{N}\right)$. For $N > 2^D$, the latter is smaller than the former.

One disadvantage of QMC is that it just provides a point estimate of \hat{f} , and does not give an uncertainty estimate. By contrast, in regular MC, we can estimate the MC standard error, as shown in discussed in Section 11.2.2. **Randomized QMC** (see e.g., [L'E18]) provides a solution to this problem. The basic idea is to repeat the QMC method R times, by perturbing the sequence of N points by a random amount. In particular, define

$$\mathbf{y}_{i,r} = \mathbf{x}_i + \mathbf{u}_r \pmod{1} \quad (11.64)$$

where $\mathbf{x}_1, \dots, \mathbf{x}_N$ is a low-discrepancy sequence, and $\mathbf{u}_r \sim \text{Unif}(0,1)^D$ is a random perturbation. The set $\{\mathbf{y}_j\}$ is low discrepancy, and satisfies that each $\mathbf{y}_j \sim \text{Unif}(0,1)^D$, for $j = 1 : N \times R$. This has much lower variance than standard MC. (Typically we take R to be a power of 2.) Recently, [OR20] proved a strong law of large numbers for RQMC.

QMC and RQMC can be used inside of MCMC inference (see e.g., [OT05]) and variational inference (see e.g., [BWM18]). It is also commonly used to select the initial set of query points for Bayesian optimization (Section 6.9).

⁶ More details on QMC can be found at http://roth.cs.kuleuven.be/wiki/Main_Page.

1 Another technique that can be used is **orthogonal Monte Carlo**, where the samples are condi-
2 tioned to be pairwise orthogonal, but with the marginal distributions matching the original ones (see
3 e.g., [Lin+20]).
4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

12 Markov Chain Monte Carlo inference

12.1 Introduction

In Chapter 11, we considered non-iterative Monte Carlo methods, including rejection sampling and importance sampling, which generate independent samples from some target distribution. The trouble with these methods is that they often do not work well in high dimensional spaces. In this chapter, we discuss a popular method for sampling from high-dimensional distributions known as **Markov chain Monte Carlo** or **MCMC**. In a survey by *SIAM News*¹, MCMC was placed in the top 10 most important algorithms of the 20th century.

The basic idea behind MCMC is to construct a Markov chain (Section 2.8) on the state space \mathcal{X} whose stationary distribution is the target density $p^*(\mathbf{x})$ of interest. (In a Bayesian context, this is usually a posterior, $p^*(\mathbf{x}) \propto p(\mathbf{x}|\mathcal{D})$, but MCMC can be applied to generate samples from any kind of distribution.) That is, we perform a random walk on the state space, in such a way that the fraction of time we spend in each state \mathbf{x} is proportional to $p^*(\mathbf{x})$. By drawing (correlated) samples $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$, from the chain, we can perform Monte Carlo integration wrt p^* .

Note that the initial samples from the chain do not come from the stationary distribution, and should be discarded; the amount of time it takes to reach stationarity is called the **mixing time** or **burn-in time**; reducing this is one of the most important factors in making the algorithm fast, as we will see.

The MCMC algorithm has an interesting history. It was discovered by physicists working on the atomic bomb at Los Alamos during World War II, and was first published in the open literature in [Met+53] in a chemistry journal. An extension was published in the statistics literature in [Has70], but was largely unnoticed. A special case (Gibbs sampling, Section 12.3) was independently invented in [GG84] in the context of Ising models (Section 4.3.2.1). But it was not until [GS90] that the algorithm became well-known to the wider statistical community. Since then it has become wildly popular in Bayesian statistics, and is becoming increasingly popular in machine learning.

In the rest of this chapter, we give a brief introduction to MCMC methods. For more details on the theory, see e.g., [GRS96; BZ20]. For more details on the implementation side, see e.g., [Lao+20].

12.2 Metropolis Hastings algorithm

In this section, we describe the most common kind of MCMC algorithm known as the **Metropolis Hastings** or **MH** algorithm.

1. Source: <http://www.siam.org/pdf/news/637.pdf>.

1 **12.2.1 Basic idea**

3 The basic idea in MH is that at each step, we propose to move from the current state \mathbf{x} to a new state
4 \mathbf{x}' with probability $q(\mathbf{x}'|\mathbf{x})$, where q is called the **proposal distribution** (also called the **kernel**).
5 The user is free to use any kind of proposal they want, subject to some conditions which we explain
6 below. This makes MH quite a flexible method.

7 Having proposed a move to \mathbf{x}' , we then decide whether to **accept** this proposal, or to reject it,
8 according to some formula, which ensures that the fraction of time spent in each state is proportional
9 to $p^*(\mathbf{x})$. If the proposal is accepted, the new state is \mathbf{x}' , otherwise the new state is the same as the
10 current state, \mathbf{x} (i.e., we repeat the sample).²

11 If the proposal is symmetric, so $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}|\mathbf{x}')$, the acceptance probability is given by the
12 following formula:

13

$$\frac{14}{15} A = \min(1, \frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})}) \quad (12.1)$$

16 We see that if \mathbf{x}' is more probable than \mathbf{x} , we definitely move there (since $\frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})} > 1$), but if \mathbf{x}' is
17 less probable, we may still move there anyway, depending on the relative probabilities. So instead of
18 greedily moving to only more probable states, we occasionally allow “downhill” moves to less probable
19 states. In Section 12.2.2, we prove that this procedure ensures that the fraction of time we spend in
20 each state \mathbf{x} is equal to $p^*(\mathbf{x})$.

21 If the proposal is asymmetric, so $q(\mathbf{x}'|\mathbf{x}) \neq q(\mathbf{x}|\mathbf{x}')$, we need the **Hastings correction**, given by
22 the following:

24

$$\frac{25}{26} A = \min(1, \alpha) \quad (12.2)$$

27

$$\alpha = \frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{p^*(\mathbf{x}')/q(\mathbf{x}'|\mathbf{x})}{p^*(\mathbf{x})/q(\mathbf{x}|\mathbf{x}')} \quad (12.3)$$

29 This correction is needed to compensate for the fact that the proposal distribution itself (rather than
30 just the target distribution) might favor certain states.

31 An important reason why MH is a useful algorithm is that, when evaluating α , we only need to
32 know the target density up to a normalization constant. In particular, suppose $p^*(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x})$, where
33 $\tilde{p}(\mathbf{x})$ is an unnormalized distribution and Z is the normalization constant. Then

34

$$\frac{35}{36} \alpha = \frac{(\tilde{p}(\mathbf{x}')/Z) q(\mathbf{x}|\mathbf{x}')}{(\tilde{p}(\mathbf{x})/Z) q(\mathbf{x}'|\mathbf{x})} \quad (12.4)$$

37 so the Z 's cancel. Hence we can sample from p^* even if Z is unknown.

39 A proposal distribution q is valid or admissible if it “covers” the support of the target. Formally,
40 we can write this as

41

$$\frac{42}{43} \text{supp}(p^*) \subseteq \cup_x \text{supp}(q(\cdot|x)) \quad (12.5)$$

43 With this, we can state the overall algorithm as in Algorithm 12.

44 45 2. Recently, a rejection-free, non-reversible MCMC algorithm, known as the **bouncy particle sampler**, has been
46 proposed [BCVD18], that can sometimes be more efficient.

Algorithm 12: Metropolis Hastings algorithm

```

1 Initialize  $x^0$  ;
2 for  $s = 0, 1, 2, \dots$  do
3   Define  $x = x^s$ 
4   Sample  $x' \sim q(x'|x)$ 
5   Compute acceptance probability
6
7   
$$\alpha = \frac{\tilde{p}(x')q(x|x')}{\tilde{p}(x)q(x'|x)}$$

8
9   Compute  $A = \min(1, \alpha)$ 
10  Sample  $u \sim U(0, 1)$ 
11  Set new sample to
12
13  
$$x^{s+1} = \begin{cases} x' & \text{if } u \leq A \text{ (accept)} \\ x^s & \text{if } u > A \text{ (reject)} \end{cases}$$

14
15
16
17
18
19
20
21
22
```

12.2.2 Why MH works

To prove that the MH procedure generates samples from p^* , we need a bit of Markov chain theory, as discussed in Section 2.8.4.

The MH algorithm defines a Markov chain with the following transition matrix:

$$p(\mathbf{x}'|\mathbf{x}) = \begin{cases} q(\mathbf{x}'|\mathbf{x})A(\mathbf{x}'|\mathbf{x}) & \text{if } \mathbf{x}' \neq \mathbf{x} \\ q(\mathbf{x}|\mathbf{x}) + \sum_{\mathbf{x}' \neq \mathbf{x}} q(\mathbf{x}'|\mathbf{x})(1 - A(\mathbf{x}'|\mathbf{x})) & \text{otherwise} \end{cases} \quad (12.6)$$

This follows from a case analysis: if you move to \mathbf{x}' from \mathbf{x} , you must have proposed it (with probability $q(\mathbf{x}'|\mathbf{x})$) and it must have been accepted (with probability $A(\mathbf{x}'|\mathbf{x})$); otherwise you stay in state \mathbf{x} , either because that is what you proposed (with probability $q(\mathbf{x}|\mathbf{x})$), or because you proposed something else (with probability $q(\mathbf{x}'|\mathbf{x})$) but it was rejected (with probability $1 - A(\mathbf{x}'|\mathbf{x})$).

Let us analyse this Markov chain. Recall that a chain satisfies **detailed balance** if

$$p(\mathbf{x}'|\mathbf{x})p^*(\mathbf{x}) = p(\mathbf{x}|\mathbf{x}')p^*(\mathbf{x}') \quad (12.7)$$

This means in the in-flow to state \mathbf{x}' from \mathbf{x} is equal to the out-flow from state \mathbf{x}' back to \mathbf{x} , and vice versa. We also showed that if a chain satisfies detailed balance, then p^* is its stationary distribution. Our goal is to show that the MH algorithm defines a transition function that satisfies detailed balance and hence that p^* is its stationary distribution. (If Equation (12.7) holds, we say that p^* is an **invariant** distribution wrt the Markov transition kernel q .)

Theorem 12.2.1. *If the transition matrix defined by the MH algorithm (given by Equation (12.6)) is ergodic and irreducible, then p^* is its unique limiting distribution.*

Proof. Consider two states \mathbf{x} and \mathbf{x}' . Either

$$p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) < p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}') \quad (12.8)$$

1
2 or
3

4 $p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) > p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')$ (12.9)

5
6 We will ignore ties (which occur with probability zero for continuous distributions). Without loss of
7 generality, assume that $p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) > p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')$. Hence

8
9 $\alpha(\mathbf{x}'|\mathbf{x}) = \frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} < 1$ (12.10)

10
11 Hence we have $A(\mathbf{x}'|\mathbf{x}) = \alpha(\mathbf{x}'|\mathbf{x})$ and $A(\mathbf{x}|\mathbf{x}') = 1$.

12 Now to move from \mathbf{x} to \mathbf{x}' we must first propose \mathbf{x}' and then accept it. Hence

13
14
15 $p(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}'|\mathbf{x})A(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}'|\mathbf{x})\frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})}q(\mathbf{x}|\mathbf{x}')$ (12.11)

16
17 Hence

18
19 $p^*(\mathbf{x})p(\mathbf{x}'|\mathbf{x}) = p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')$ (12.12)

20
21 The backwards probability is

22
23 $p(\mathbf{x}|\mathbf{x}') = q(\mathbf{x}|\mathbf{x}')A(\mathbf{x}|\mathbf{x}') = q(\mathbf{x}|\mathbf{x}')$ (12.13)

24
25 since $A(\mathbf{x}|\mathbf{x}') = 1$. Inserting this into Equation (12.12) we get

26
27 $p^*(\mathbf{x})p(\mathbf{x}'|\mathbf{x}) = p^*(\mathbf{x}')p(\mathbf{x}|\mathbf{x}')$ (12.14)

28
29 so detailed balance holds wrt p^* . Hence, from Theorem 2.8.3, p^* is a stationary distribution.

30
31 Furthermore, from Theorem 2.8.2, this distribution is unique, since the chain is ergodic and irreducible.

32 \square

33 34 12.2.3 Proposal distributions

35
36 In this section, we discuss some common proposal distributions. Note, however, that good proposal
37 design is often intimately dependent on the form of the target distribution (most often the posterior).

38

39 12.2.3.1 Independence sampler

40

41 If we use a proposal of the form $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}')$, where the new state is independent of the old
42 state, we get a method known as the **independence sampler**, which is similar to importance
43 sampling (Section 11.5). The function $q(\mathbf{x}')$ can be any suitable distribution, such as a Gaussian.
44 This has non-zero probability density on the entire state space, and hence is a valid proposal for any
45 unconstrained continuous state space.

46

47

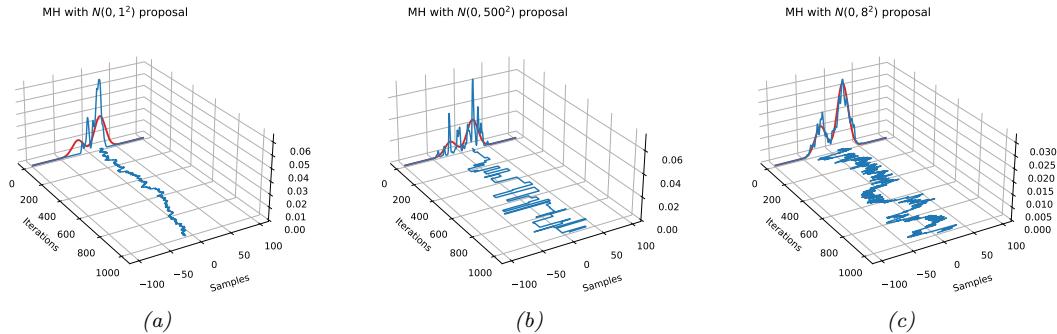


Figure 12.1: An example of the Metropolis Hastings algorithm for sampling from a mixture of two 1D Gaussians ($\mu = (-20, 20)$, $\pi = (0.3, 0.7)$, $\Sigma = (100, 100)$), using a Gaussian proposal with standard deviation of $\tau \in \{1, 8, 500\}$. (a) When $\tau = 1$, the chain gets trapped near the starting state and fails to sample from the mode at $\mu = -20$. (b) When $\tau = 500$, the chain is very “sticky”, so its effective sample size is low (as reflected by the rough histogram approximation at the end). (c) Using a variance of $\tau = 8$ is just right and leads to a good approximation of the true distribution (shown in red). Compare to Figure 12.7. Generated by `mcmc_gmm_demo.py`.

12.2.3.2 Random walk Metropolis (RWM) algorithm

The **random walk Metropolis** algorithm corresponds to MH with the following proposal distribution:

$$q(\mathbf{x}' | \mathbf{x}) = \mathcal{N}(\mathbf{x}' | \mathbf{x}, \tau^2 \mathbf{I}) \quad (12.15)$$

Here τ^2 is a scale factor chosen to facilitate rapid mixing. [RR01b] prove that, if the posterior is Gaussian, the asymptotically optimal value is to use $\tau^2 = 2.38^2/D$, where D is the dimensionality of \mathbf{x} ; this results in an acceptance rate of 0.234, which (in this case) is the optimal tradeoff between exploring widely enough to cover the distribution without being rejected too often. (See [Béd08] for a more recent account of optimal acceptance rates for random walk Metropolis methods.)

Figure 12.1 shows an example where we use RWM to sample from a mixture of two 1D Gaussians. This is a somewhat tricky target distribution, since it consists of two well separated modes. It is very important to set the variance of the proposal τ^2 correctly: If the variance is too low, the chain will only explore one of the modes, as shown in Figure 12.1(a), but if the variance is too large, most of the moves will be rejected, and the chain will be very **sticky**, i.e., it will stay in the same state for a long time. This is evident from the long stretches of repeated values in Figure 12.1(b). If we set the proposal’s variance just right, we get the trace in Figure 12.1(c), where the samples clearly explore the support of the target distribution.

1 **12.2.3.3 Composing proposals**

3 If there are several proposals that might be useful, one can combine them using a **mixture proposal**,
4 which is a convex combination of base proposals:
5

$$\underline{6} \quad q(\mathbf{x}'|\mathbf{x}) = \sum_{k=1}^K w_k q_k(\mathbf{x}'|\mathbf{x}) \quad (12.16)$$

9 where w_k are the mixing weights. As long as each q_k is an individually valid proposal, and each
10 $w_k > 0$, then the overall mixture proposal will also be valid. In particular, if each proposal is
11 reversible, so it satisfies detailed balance (Section 2.8.4.4), then so does the mixture.
12

13 It is also possible to compose individual proposals by chaining them together to get

$$\underline{14} \quad q(\mathbf{x}'|\mathbf{x}) = \sum_{\mathbf{x}_1} \cdots \sum_{\mathbf{x}_{K-1}} q_1(\mathbf{x}_1|\mathbf{x}) q_2(\mathbf{x}_2|\mathbf{x}_1) \cdots q_K(\mathbf{x}| \mathbf{x}_{K-1}) \quad (12.17)$$

17 A common example is where each base proposal only updates a subset of the variables (see e.g.,
18 Section 12.3). This does not satisfy detailed balance, even if the components do, although it is still
19 valid as an overall proposal. We can restore detailed balance by symmetrizing the order of the base
20 transitions, by using $q_1, q_2, \dots, q_K, q_K, \dots, q_1$.
21

22 **12.2.3.4 Data-driven MCMC**

24 In the case where the target distribution is a posterior, $p^*(\mathbf{x}) = p(\mathbf{x}|\mathcal{D})$, it is helpful to condition the
25 proposal not just on the previous hidden state, but also the visible data, i.e., to use $q(\mathbf{x}'|\mathbf{x}, \mathcal{D})$. This
26 is called **data-driven MCMC** (see e.g., [TZ02; Jih+12]).

27 One way to create such a proposal is to train a recognition network to propose states using
28 $q(\mathbf{x}'|\mathbf{x}, \mathcal{D}) = f(\mathbf{x})$. If the state space is high-dimensional, it might be hard to predict all the hidden
29 components, so we can alternatively train individual “experts” to predict specific pieces of the hidden
30 state. For example, in the context of estimating the 3d pose of a person from an image, we might
31 combine a face detector with a limb detector. We can then use a mixture proposal of the form
32

$$\underline{33} \quad q(\mathbf{x}'|\mathbf{x}, \mathcal{D}) = \pi_0 q_0(\mathbf{x}'|\mathbf{x}) + \sum_k \pi_k q_k(x'_k|f_k(\mathcal{D})) \quad (12.18)$$

35 where q_0 is a standard data-independent proposal (e.g., random walk), and q_k updates the k 'th
36 component of the state space.
37

38 The overall procedure is a form of **generate and test**: the discriminative proposals $q(\mathbf{x}'|\mathbf{x}, \mathcal{D})$
39 generate new hypotheses, which are then “tested” by computing the posterior ratio $\frac{p(\mathbf{x}'|\mathcal{D})}{p(\mathbf{x}|\mathcal{D})}$, to see if
40 the new hypothesis is better or worse. (See also Section 13.4, where we discuss learning proposal
41 distributions for particle filters.)
42

43 **12.2.3.5 Adaptive MCMC**

44 One can change the parameters of the proposal as the algorithm is running to increase efficiency.
45 This is called **adaptive MCMC**. This allows one to start with a broad covariance (say), allowing
46

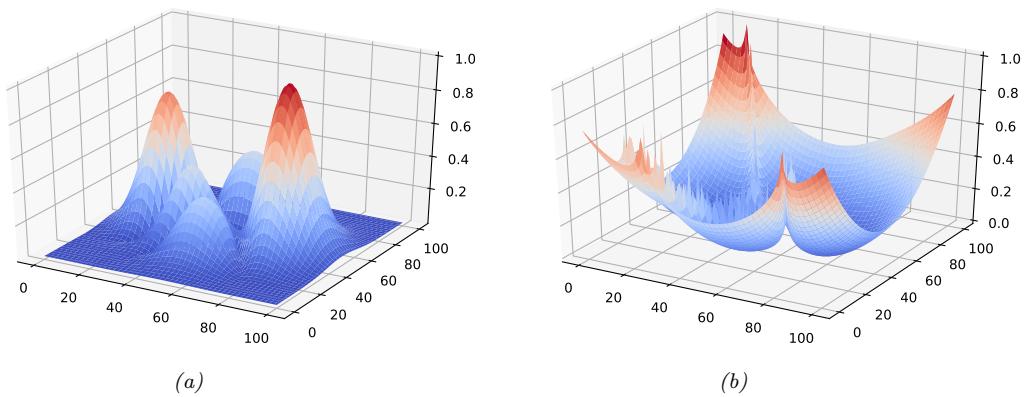


Figure 12.2: (a) A peaky distribution. (b) Corresponding energy function. Generated by [simulated_annealing_2d_demo.ipynb](#).

large moves through the space until a mode is found, followed by a narrowing of the covariance to ensure careful exploration of the region around the mode.

However, one must be careful not to violate the Markov property; thus the parameters of the proposal should not depend on the entire history of the chain. It turns out that a sufficient condition to ensure this is that the adaption is “faded out” gradually over time. See e.g., [AT08] for details.

12.2.4 Initialization

It is necessary to start MCMC in an initial state that has non-zero probability. A natural approach is to first find an optimizer to find a local mode. However, at such points the gradients of the log joint are zero, which can cause problems for some gradient-based MCMC methods, such as HMC (Section 12.5), so it can be better to start “close” to a MAP estimate (see e.g., [HFM17, Sec 7.]).

12.2.5 Simulated annealing

Simulated annealing [KJV83; LA87] is a **stochastic local search** algorithm that attempts to find the global minimum of a black-box function $\mathcal{E}(\mathbf{x})$, where $\mathcal{E}()$ is known as the **energy function**. The method works by converting the energy to an (unnormalized) probability distribution over states by defining $p(\mathbf{x}) = \exp(-\mathcal{E}(\mathbf{x}))$, and then using a variant of the **Metropolis Hastings** algorithm to sample from a set of probability distributions, designed so that at the final step, the method samples from one of the modes of the distribution, i.e., it finds one of the most likely states, or lowest energy states. This approach can be used for both discrete and continuous optimization.

Annealing is a physical process of heating a solid until thermal stresses are released, then cooling it very slowly until the crystals are perfectly arranged, achieving a minimum energy state. Depending on how fast or slow the temperature is cooled, the results will have worse or better the quality. We can apply this approach to probability distributions, to control the number of modes (low energy

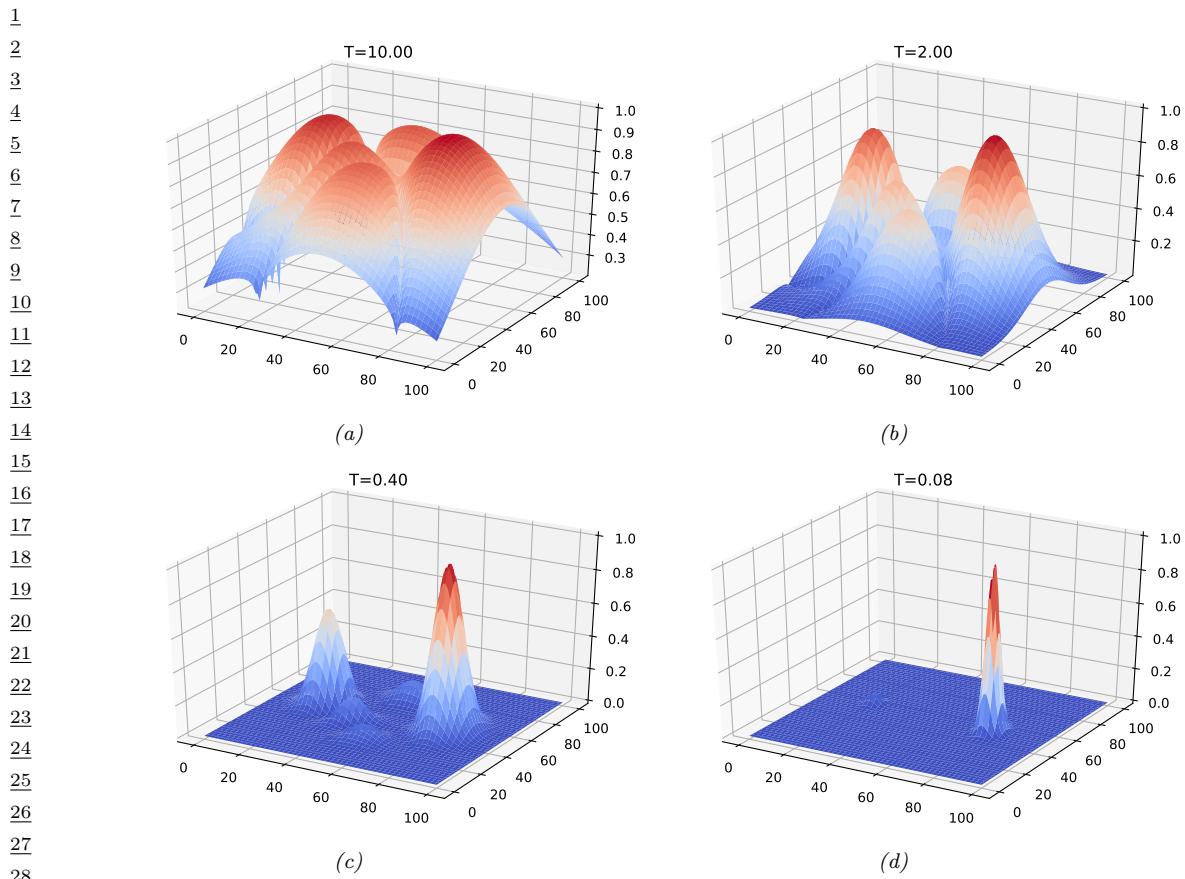


Figure 12.3: Annealed version of the distribution in Figure 12.2a at different temperatures. Generated by [simulated_annealing_2d_demo.ipynb](#).

states) that they have, by defining

$$p_T(\mathbf{x}) = \exp(-\mathcal{E}(\mathbf{x})/T) \quad (12.19)$$

where T is the temperature, which is reduced over time. As an example, consider the **peaks function**:

$$p(x, y) \propto |3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2}| \quad (12.20)$$

This is plotted in Figure 12.2a. The corresponding energy is in Figure 12.2b. We plot annealed versions of this distribution in Figure 12.3. At high temperatures, $T \gg 1$, the surface is approximately flat, and hence it is easy to move around (i.e., to avoid local optima). As the temperature cools, the largest peaks become larger, and the smallest peaks disappear. By cooling slowly enough, it is possible to “track” the largest peak, and thus find the global optimum (minimum energy state). This is an example of a **continuation method**.

47

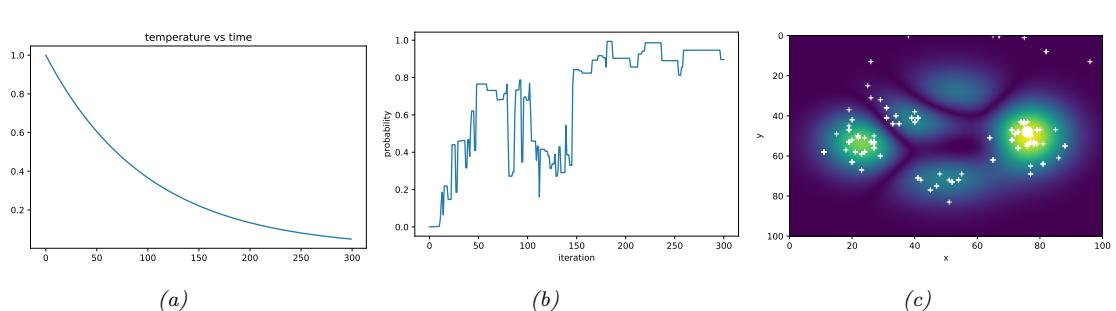


Figure 12.4: Simulated annealing applied to the distribution in Figure 12.2a. (a) Temperature vs time. (b) Probability of each visited point vs time. (c) Visited samples, superimposed on the target distribution. The big white dot is the highest probability point found. Generated by simulated annealing 2d demo.ipynb.

In more detail, at each step, we sample a new state according to some proposal distribution $\mathbf{x}' \sim q(\cdot | \mathbf{x}_t)$. For real-valued parameters, this is often simply a random walk proposal centered on the current iterate, $\mathbf{x}' = \mathbf{x}_t + \boldsymbol{\epsilon}_{t+1}$, where $\boldsymbol{\epsilon}_{t+1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. (The matrix $\boldsymbol{\Sigma}$ is often diagonal, and may be updated over time using the method in [Cor+87].) Having proposed a new state, we compute the acceptance probability

$$\alpha_{t+1} = \exp(-(E(\mathbf{x}') - E(\mathbf{x}_t))/T_t) \quad (12.21)$$

where T_t is the temperature of the system. We then accept the new state (i.e., set $\mathbf{x}_{t+1} = \mathbf{x}'$) with probability $\min(1, \alpha_{t+1})$, otherwise we stay in the current state (i.e., set $\mathbf{x}_{t+1} = \mathbf{x}_t$). This means that if the new state has lower energy (is more probable), we will definitely accept it, but if it has higher energy (is less probable), we might still accept, depending on the current temperature. Thus the algorithm allows “downhill” moves in probability space (uphill in energy space), but less frequently as the temperature drops.

The rate at which the temperature changes over time is called the **cooling schedule**. It has been shown [Haj88] that if one cools according to a logarithmic schedule, $T_t \propto 1/\log(t+1)$, then the method is guaranteed to find the global optimum under certain assumptions. However, this schedule is often too slow. In practice it is common to use an **exponential cooling schedule** of the form $T_{t+1} = \gamma T_t$, where $\gamma \in (0, 1]$ is the cooling rate. Cooling too quickly means one can get stuck in a local maximum, but cooling too slowly just wastes time. The best cooling schedule is difficult to determine; this is one of the main drawbacks of simulated annealing.

In Figure 12.4a, we show a cooling schedule using $\gamma = 0.9$. If we combine this with a Gaussian random walk proposal with $\sigma = 10$ to the peaky distribution in Figure 12.2a, we get the results shown in Figure 12.4b and Figure 12.4c. We see that the algorithm concentrates its samples near the global optimum (the peak on the middle right).

12.3 Gibbs sampling

The major problem with MH is the need to choose the proposal distribution, and the fact that the acceptance rate may be low. In this section, we describe an MH method that exploits conditional independence properties of a graphical model to automatically create a good proposal, with acceptance

¹ probability 1. This method is known as **Gibbs sampling**.³ (In physics, this method is known as
² **Glauber dynamics** or the **heat bath** method.) This is the MCMC analog of coordinate descent.
³

⁴ 12.3.1 Basic idea

⁶ The idea behind Gibbs sampling is to sample each variable in turn, conditioned on the values of all
⁷ the other variables in the distribution. For example, if we have $D = 3$ variables, we use
⁸

- ⁹ • $x_1^{s+1} \sim p(x_1|x_2^s, x_3^s)$
- ¹⁰ • $x_2^{s+1} \sim p(x_2|x_1^{s+1}, x_3^s)$
- ¹¹ • $x_3^{s+1} \sim p(x_3|x_1^{s+1}, x_2^{s+1})$

¹³ This readily generalizes to D variables. If x_i is a visible variable, we do not sample it, since its value
¹⁴ is already known.

¹⁵ The expression $p(x_i|\mathbf{x}_{-i})$ is called the **full conditional** for variable i . In general, x_i may only
¹⁶ depend on some of the other variables. If we represent $p(\mathbf{x})$ as a graphical model, we can infer the
¹⁷ dependencies by looking at i 's Markov blanket, which are its neighbors in the graph. Thus to sample
¹⁸ x_i , we only need to know the values of i 's neighbors.

¹⁹ We can sample some of the nodes in parallel, without affecting correctness. In particular, suppose
²⁰ we can create a **coloring** of the (moralized) undirected graph, such that no two neighboring nodes
²¹ have the same color. (In general, computing an optimal coloring is NP-complete, but we can use
²² efficient heuristics such as those in [Kub04].) Then we can sample all the nodes of the same color in
²³ parallel, and cycle through the colors sequentially [Gon+11].
²⁴

²⁵ 12.3.2 Gibbs sampling is a special case of MH

²⁷ It turns out that Gibbs sampling is a special case of MH where we use a sequence of proposals of the
²⁸ form

$$\begin{aligned} \text{²⁹ } q(\mathbf{x}'|\mathbf{x}) &= p(x'_i|\mathbf{x}_{-i})\mathbb{I}(x'_{-i} = \mathbf{x}_{-i}) \end{aligned} \tag{12.22}$$

³¹ That is, we move to a new state where x_i is sampled from its full conditional, but \mathbf{x}_{-i} is left
³² unchanged.

³³ We now prove that the acceptance rate of each such proposal is 1, so the overall algorithm also
³⁴ has an acceptance rate of 100%. We have

$$\begin{aligned} \text{³⁵ } \alpha &= \frac{p(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{p(x'_i|\mathbf{x}'_{-i})p(\mathbf{x}'_{-i})p(x_i|\mathbf{x}'_{-i})}{p(x_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x'_i|\mathbf{x}_{-i})} \end{aligned} \tag{12.23}$$

$$\begin{aligned} \text{³⁸ } &= \frac{p(x'_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x_i|\mathbf{x}_{-i})}{p(x_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x'_i|\mathbf{x}_{-i})} = 1 \end{aligned} \tag{12.24}$$

⁴⁰ where we exploited the fact that $\mathbf{x}'_{-i} = \mathbf{x}_{-i}$, and that $q(\mathbf{x}'|\mathbf{x}) = p(x'_i|\mathbf{x}_{-i})$.

⁴¹ The fact that the acceptance rate is 100% does not necessarily mean that Gibbs will converge
⁴² rapidly, since it only updates one coordinate at a time (see Section 12.3.7). However, if we can group
⁴³ together correlated variables, then we can sample them as a group, which can significantly help
⁴⁴ mixing.
⁴⁵

⁴⁶ 3. Josiah Willard Gibbs, 1839–1903, was an American physicist.

⁴⁷

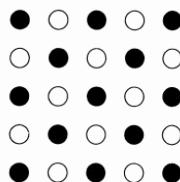


Figure 12.5: Illustration of checkerboard pattern for a 2d MRF. This allows for parallel updates.

12.3.3 Example: Gibbs sampling for Ising models

In Section 4.3.2.1, we discuss Ising models and Potts models, which are pairwise MRFs with a 2d grid structure defined over a set of variables:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{i \sim j} \psi_{ij}(x_i, x_j | \boldsymbol{\theta}) \quad (12.25)$$

where $i \sim j$ means i and j are neighbors in the graph.

To apply Gibbs sampling to such a model, we just need to iteratively sample from each full conditional:

$$p(x_i | \mathbf{x}_{-i}, \boldsymbol{\theta}) \propto \prod_{j \in \text{nbr}(i)} \psi_{ij}(x_i, x_j) \quad (12.26)$$

Note that although Gibbs sampling is a sequential algorithm, we can sometimes exploit conditional independence properties to perform parallel updates [RS97a]. In the case of a 2d grid, we can color code nodes using a checkerboard pattern shown in Figure 12.5. This has the property that the black nodes are conditionally independent of each other given the white nodes, and vice versa. Hence we can sample all the black nodes in parallel (as a single group), and then sample all the white nodes, etc.

To perform the sampling, we need to compute the full conditional in Equation (12.26). In the case of an Ising model with edge potentials $\psi(x_i, x_j) = \exp(Jx_i x_j)$, where $x_i \in \{-1, +1\}$, the full conditional becomes

$$p(x_i = +1 | \mathbf{x}_{-i}, \boldsymbol{\theta}) = \frac{\prod_{j \in \text{nbr}(i)} \psi_{ij}(x_i = +1, x_j)}{\prod_{j \in \text{nbr}(i)} \psi(x_i = +1, x_j) + \prod_{j \in \text{nbr}(i)} \psi(x_i = -1, x_j)} \quad (12.27)$$

$$= \frac{\exp[J \sum_{j \in \text{nbr}(i)} x_j]}{\exp[J \sum_{j \in \text{nbr}(i)} x_j] + \exp[-J \sum_{j \in \text{nbr}(i)} x_j]} \quad (12.28)$$

$$= \frac{\exp[J\eta_i]}{\exp[J\eta_i] + \exp[-J\eta_i]} = \sigma(2J\eta_i) \quad (12.29)$$

where J is the coupling strength, $\eta_i \triangleq \sum_{j \in \text{nbr}(i)} x_j$ and $\sigma(u) = 1/(1 + e^{-u})$ is the sigmoid function. (If we use $x_i \in \{0, 1\}$, this becomes $p(x_i = +1 | \mathbf{x}_{-i}) = \sigma(J\eta_i)$.) It is easy to see that $\eta_i = x_i(a_i - d_i)$, where a_i is the number of neighbors that agree with (have the same sign as) node i , and d_i is the

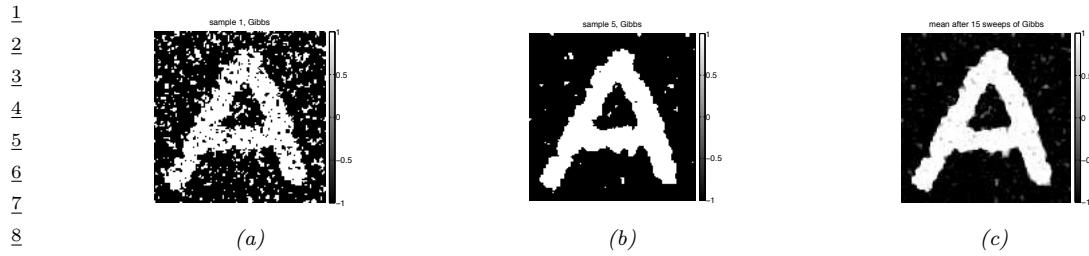


Figure 12.6: Example of image denoising using Gibbs sampling. We use an Ising prior with $J = 1$ and a Gaussian noise model with $\sigma = 2$. (a) Sample from the posterior after one sweep over the image. (b) Sample after 5 sweeps. (c) Posterior mean, computed by averaging over 15 sweeps. Compare to Figure 10.3 which shows the results of mean field inference. Generated by `ising_image_denoise_demo.py`.

¹⁵ number of neighbors who disagree. If this number is equal, the “forces” on x_i cancel out, so the full conditional is uniform. Some samples from this model are shown in Figure 4.16.

One application of Ising models is as a prior for binary image denoising problems. In particular, suppose \mathbf{y} is a noisy version of \mathbf{x} , and we wish to compute the posterior $p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$, where $p(\mathbf{x})$ is an Ising prior, and $p(\mathbf{y}|\mathbf{x}) = \prod_i p(y_i|x_i)$ is a per-site likelihood term. Suppose this is a Gaussian. Let $\psi_i(x_i) = \mathcal{N}(y_i|x_i, \sigma^2)$ be the corresponding “local evidence” term. The full conditional becomes

$$p(x_i = +1 | \boldsymbol{x}_{-i}, \boldsymbol{y}, \boldsymbol{\theta}) = \frac{\exp[J\eta_i]\psi_i(+1)}{\exp[J\eta_i]\psi_i(+1) + \exp[-J\eta_i]\psi_i(-1)} \quad (12.30)$$

$$= \sigma \left(2J\eta_i - \log \frac{\psi_i(+1)}{\psi_i(-1)} \right) \quad (12.31)$$

²⁸ Now the probability of x_i entering each state is determined both by compatibility with its neighbors
²⁹ (the Ising prior) and compatibility with the data (the local likelihood term).

See Figure 12.6 for an example of this algorithm applied to a simple image denoising problem. The results are similar to the mean field results in Figure 10.3.

12.3.4 Example: Gibbs sampling for Potts models

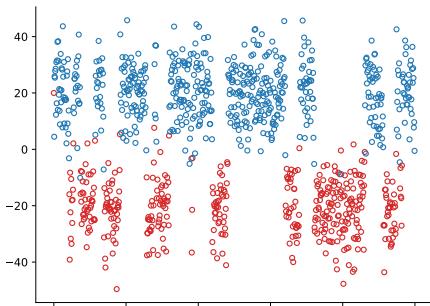
³⁵ We can extend Section 12.3.3 to the Potts models as follows. Recall that the model has the following
³⁶ form:

$$p(\mathbf{x}) = \frac{1}{Z} \exp(-\mathcal{E}(\mathbf{x})) \quad (12.32)$$

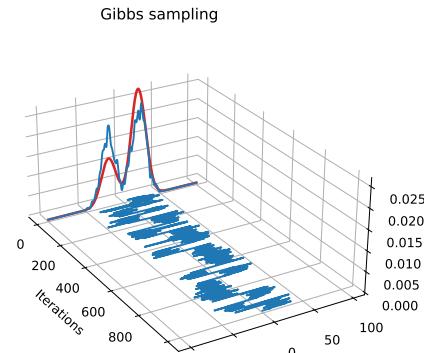
$$\mathcal{E}(\mathbf{x}) = -J \sum_{i \sim j} \mathbb{I}(x_i = x_j) \quad (12.33)$$

43 For a node i with neighbors $\text{nbr}(i)$, the full conditional is thus given by

$$p(x_i = k | \mathbf{x}_{-i}) = \frac{\exp(J \sum_{n \in \text{nbr}(i)} \mathbb{I}(x_n = k))}{\sum_{k'} \exp(J \sum_{n \in \text{nbr}(i)} \mathbb{I}(x_n = k'))} \quad (12.34)$$



(a)



(b)

Figure 12.7: (a) Some samples from a mixture of two 1d Gaussians generated using Gibbs sampling. Color denotes the value of z , vertical location denotes the value of x . Horizontal axis represents time (sample number). (b) Traceplot of x over time, and the resulting empirical distribution is shown in blue. The true distribution is shown in red. Compare to Figure 12.1. Generated by `mcmc_gmm_demo.py`.

So if $J > 0$, a node i is more likely to enter a state k if most of its neighbors are already in state k , corresponding to an attractive MRF. If $J < 0$, a node i is more likely to enter a different state from its neighbors, corresponding to a repulsive MRF. See Figure 4.17 for some samples from this model created using this method.

12.3.5 Example: Gibbs sampling for GMMs

In this section, we consider sampling from a Bayesian Gaussian mixture model of the form

$$p(z = k, \mathbf{x} | \boldsymbol{\theta}) = \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.35)$$

$$p(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \boldsymbol{\alpha}) \prod_{k=1}^K \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_0, \mathbf{V}_0) \text{IW}(\boldsymbol{\Sigma}_k, \mathbf{S}_0, \nu_0) \quad (12.36)$$

12.3.5.1 Known parameters

Suppose, initially, that the parameters $\boldsymbol{\theta}$ are known. We can easily draw independent samples from $p(\mathbf{x} | \boldsymbol{\theta})$ by using ancestral sampling: first sample z and then \mathbf{x} . However, for illustrative purposes, we will use Gibbs sampling to draw correlated samples. The full conditional for $p(\mathbf{x} | z = k, \boldsymbol{\theta})$ is just $\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, and the full conditional for $p(z = k | \mathbf{x})$ is given by Bayes rule:

$$p(z = k | \mathbf{x}, \boldsymbol{\theta}) = \frac{\pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} \quad (12.37)$$

An example of this procedure, applied to a mixture of two 1D Gaussians with means at -20 and $+20$, is shown in Figure 12.7. We see that the samples are auto-correlated, meaning that if we are

in state 1, we will likely stay in that state for a while, and generate values near μ_1 ; then we will stochastically jump to state 2, and stay near there for a while, etc. By contrast, independent samples from the joint would not be correlated at all. This means that MCMC samples carry less information than independent samples, a fact we return to in Section 12.6.2.3.

In Section 12.3.5.2, we modify this example to sample the parameters of the GMM from their posterior, $p(\boldsymbol{\theta}|\mathcal{D})$, instead of sampling from $p(\mathcal{D}|\boldsymbol{\theta})$.

12.3.5.2 Unknown parameters

Now suppose the parameters are unknown, so we want to fit the model to data. If we use a conditionally conjugate factored prior, then the full joint distribution is given by

$$p(\mathbf{x}, \mathbf{z}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu}, \boldsymbol{\Sigma})p(\mathbf{z}|\boldsymbol{\pi})p(\boldsymbol{\pi}) \prod_{k=1}^K p(\boldsymbol{\mu}_k)p(\boldsymbol{\Sigma}_k) \quad (12.38)$$

$$= \left(\prod_{i=1}^N \prod_{k=1}^K (\pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{\mathbb{I}(z_i=k)} \right) \times \quad (12.39)$$

$$\text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}) \prod_{k=1}^K \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_0, \mathbf{V}_0) \text{IW}(\boldsymbol{\Sigma}_k | \mathbf{S}_0, \nu_0) \quad (12.40)$$

We use the same prior for each mixture component.

The full conditionals are as follows. For the discrete indicators, we have

$$p(z_i = k | \mathbf{x}_i, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \propto \pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.41)$$

For the mixing weights, we have (using results from Section 3.2.2)

$$p(\boldsymbol{\pi}|\mathbf{z}) = \text{Dir}(\{\alpha_k + \sum_{i=1}^N \mathbb{I}(z_i = k)\}_{k=1}^K) \quad (12.42)$$

For the means, we have (using results from Section 3.2.4.1)

$$p(\boldsymbol{\mu}_k | \boldsymbol{\Sigma}_k, \mathbf{z}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_k, \mathbf{V}_k) \quad (12.43)$$

$$\mathbf{V}_k^{-1} = \mathbf{V}_0^{-1} + N_k \boldsymbol{\Sigma}_k^{-1} \quad (12.44)$$

$$\mathbf{m}_k = \mathbf{V}_k (\boldsymbol{\Sigma}_k^{-1} N_k \bar{\mathbf{x}}_k + \mathbf{V}_0^{-1} \mathbf{m}_0) \quad (12.45)$$

$$N_k \triangleq \sum_{i=1}^N \mathbb{I}(z_i = k) \quad (12.46)$$

$$\bar{\mathbf{x}}_k \triangleq \frac{\sum_{i=1}^N \mathbb{I}(z_i = k) \mathbf{x}_i}{N_k} \quad (12.47)$$

For the covariances, we have (using results from Section 3.2.4.2)

$$p(\boldsymbol{\Sigma}_k | \boldsymbol{\mu}_k, \mathbf{z}, \mathbf{x}) = \text{IW}(\boldsymbol{\Sigma}_k | \mathbf{S}_k, \nu_k) \quad (12.48)$$

$$\mathbf{S}_k = \mathbf{S}_0 + \sum_{i=1}^N \mathbb{I}(z_i = k) (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T \quad (12.49)$$

$$\nu_k = \nu_0 + N_k \quad (12.50)$$

12.3.6 Sampling from the full conditionals

When implementing Gibbs sampling, we have to sample from the full conditionals. Sometimes this is difficult, especially when we are not using a conjugate prior. We discuss some possible solutions below.

12.3.6.1 Adaptive rejection Metropolis sampling

One approach to sample from $x_i^{s+1} \sim p(x_i | \mathbf{x}_{1:i-1}^{s+1}, \mathbf{x}_{i+1:D}^s)$ is to use adaptive rejection sampling (Section 11.4.3). This is known as **adaptive rejection Metropolis sampling** [GBT95].

12.3.6.2 Metropolis within Gibbs

Another possibility is to use the MH algorithm; this is called **Metropolis within Gibbs**. In more detail, it works as follows. To sample from $x_i^{s+1} \sim p(x_i | \mathbf{x}_{1:i-1}^{s+1}, \mathbf{x}_{i+1:D}^s)$, we proceed in 3 steps:

1. Propose $x'_i \sim q(x'_i | x_i^s)$

2. Compute the acceptance probability $A_i = \min(1, \alpha_i)$ where

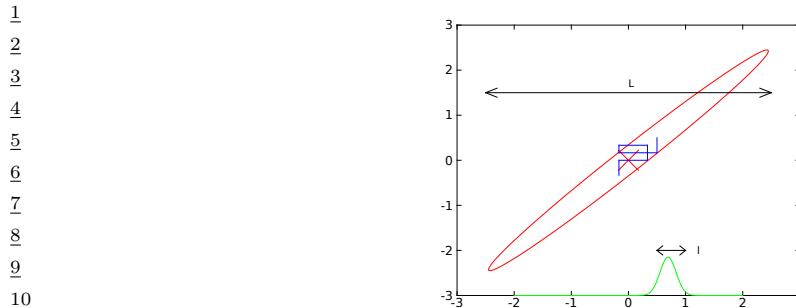
$$\alpha_i = \frac{p(\mathbf{x}_{1:i-1}^{s+1}, x'_i, \mathbf{x}_{i+1:D}^s) / q(x'_i | x_i^s)}{p(\mathbf{x}_{1:i-1}^s, x_i^s, \mathbf{x}_{i+1:D}^s) / q(x_i^s | x'_i)} \quad (12.51)$$

3. Sample $u \sim U(0, 1)$ and set $x_i^{s+1} = x'_i$ if $u < A_i$, and set $x_i^{s+1} = x_i^s$ otherwise.

12.3.7 Blocked Gibbs sampling

Gibbs sampling can be quite slow, since it only updates one variable at a time (so-called **single site updating**). If the variables are highly correlated, it will take a long time to move away from the current state. This is illustrated in Figure 12.8, where we illustrate sampling from a 2d Gaussian. If the variables are highly correlated, the algorithm will move very slowly through the state space. In particular, the size of the moves is controlled by the variance of the conditional distributions. If this is ℓ in the x_1 direction, and the support of the distribution is L along this dimension, then we need $O((L/\ell)^2)$ steps to obtain an independent sample.

In some cases we can efficiently sample groups of variables at a time. This is called **blocked Gibbs sampling** [JKK95; WY02], and can make much bigger moves through the state space.



11 *Figure 12.8: Illustration of potentially slow sampling when using Gibbs sampling for a skewed 2D Gaussian.*
12 Adapted from Figure 11.11 of [Bis06]. Generated by [gibbs_gauss_demo.py](#).

17 12.3.7.1 Example: Blocked Gibbs for HMMs

18 Suppose we want to perform Bayesian inference for a state-space model, such as an HMM, i.e., we
19 want to sample from

$$\begin{aligned} \text{21} \\ \text{22} \quad p(\theta, z|x) &\propto p(\theta) \prod_{t=1}^T p(x_t|z_t, \theta)p(z_t|z_{t-1}, \theta) \end{aligned} \tag{12.52}$$

25 We can use blocked Gibbs sampling, where we alternate between sampling from $p(\theta|z, x)$ and
26 $p(z|x, \theta)$; The former is easy to do (assuming conjugate priors), since all variables in the model are
27 observed. The latter can be done using the forwards-filtering backwards-sampling (Section 8.3.7).
28 For details, see [Sco02].

31 12.3.8 Collapsed Gibbs sampling

32 We can sometimes gain even greater speedups by analytically integrating out some of the unknown
33 quantities. This is called a **collapsed Gibbs sampler**, and it tends to be more efficient, since it is
34 sampling in a lower dimensional space, which results in lower variance, as discussed in Section 11.6.1.
35 It can also be a useful stepping stone for building samplers for “infinite” models, as we discuss in
36 Chapter 33. We give some examples below.

39 12.3.8.1 Example: collapsed Gibbs for GMMs

41 Consider a GMM with a fully conjugate prior. This can be represented as a PGM-D as shown in
42 Figure 12.9a. Since the prior is conjugate, we can analytically integrate out the model parameters μ_k ,
43 Σ_k and π , so the only remaining hidden variables are the discrete indicator variables z . However,
44 once we integrate out π , all the z_i nodes become inter-dependent. Similarly, once we integrate out
45 $\theta_k = (\mu_k, \Sigma_k)$, all the x_i nodes become inter-dependent, as shown in Figure 12.9b. Nevertheless, we
46 can easily compute the full conditionals, and hence implement a Gibbs sampler.

47

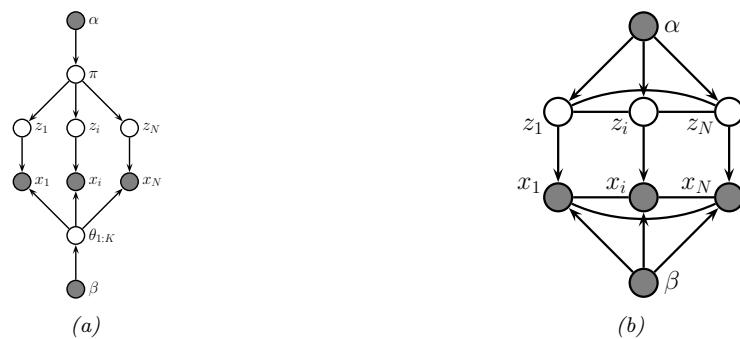


Figure 12.9: (a) A mixture model represented as an “unrolled” PGM-D. (b) After integrating out the continuous latent parameters.

In particular, the full conditional for the latent indicators is given by

$$p(z_i = k | \mathbf{z}_{-i}, \mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}, \boldsymbol{\beta}) p(\mathbf{x} | z_i = k, \mathbf{z}_{-i}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \quad (12.53)$$

$$\begin{aligned} &\propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}) p(\mathbf{x}_i | \mathbf{x}_{-i}, z_i = k, \mathbf{z}_{-i}, \boldsymbol{\beta}) \\ &= p(\mathbf{x}_{-i} | z_i = k, \mathbf{z}_{-i}, \boldsymbol{\beta}) \end{aligned} \quad (12.54)$$

$$\propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}) p(\mathbf{x}_i | \mathbf{x}_{-i}, z_i = k, \mathbf{z}_{-i}, \boldsymbol{\beta}) \quad (12.55)$$

where $\boldsymbol{\beta} = (\mathbf{m}_0, \mathbf{V}_0, \mathbf{S}_0, \nu_0)$ are the hyper-parameters for the class-conditional densities. The first term can be obtained by integrating out $\boldsymbol{\pi}$. Suppose we use a symmetric prior of the form $\boldsymbol{\pi} \sim \text{Dir}(\boldsymbol{\alpha})$, where $\alpha_k = \alpha/K$. From Equation (3.68) we have

$$p(z_1, \dots, z_N | \alpha) = \frac{\Gamma(\alpha)}{\Gamma(N + \alpha)} \prod_{k=1}^K \frac{\Gamma(N_k + \alpha/K)}{\Gamma(\alpha/K)} \quad (12.56)$$

Hence

$$p(z_i = k | \mathbf{z}_{-i}, \alpha) = \frac{p(\mathbf{z}_{1:N} | \alpha)}{p(\mathbf{z}_{-i} | \alpha)} = \frac{\frac{1}{\Gamma(N + \alpha)}}{\frac{1}{\Gamma(N + \alpha - 1)}} \times \frac{\Gamma(N_k + \alpha/K)}{\Gamma(N_{k,-i} + \alpha/K)} \quad (12.57)$$

$$= \frac{\Gamma(N + \alpha - 1)}{\Gamma(N + \alpha)} \frac{\Gamma(N_{k,-i} + 1 + \alpha/K)}{\Gamma(N_{k,-i} + \alpha/K)} = \frac{N_{k,-i} + \alpha}{N + \alpha - 1} \quad (12.58)$$

where $N_{k,-i} \triangleq \sum_{n \neq i} \mathbb{I}(z_n = k) = N_k - 1$, and where we exploited the fact that $\Gamma(x + 1) = x\Gamma(x)$.

To obtain the second term in Equation (12.55), which is the posterior predictive distribution for \mathbf{x}_i given all the other data and all the assignments, we use the fact that

$$p(\mathbf{x}_i | \mathbf{x}_{-i}, \mathbf{z}_{-i}, z_i = k, \boldsymbol{\beta}) = p(\mathbf{x}_i | \mathcal{D}_{-i,k}, \boldsymbol{\beta}) \quad (12.59)$$

where $\mathcal{D}_{-i,k} = \{\mathbf{x}_j : z_j = k, j \neq i\}$ is all the data assigned to cluster k except for \mathbf{x}_i . If we use a conjugate prior for $\boldsymbol{\theta}_k$, we can compute $p(\mathbf{x}_i | \mathcal{D}_{-i,k}, \boldsymbol{\beta})$ in closed form. Furthermore, we can efficiently update these predictive likelihoods by caching the sufficient statistics for each cluster. To compute

1 the above expression, we remove \mathbf{x}_i 's statistics from its current cluster (namely z_i), and then evaluate
 2 \mathbf{x}_i under each cluster's posterior predictive distribution. Once we have picked a new cluster, we add
 3 \mathbf{x}_i 's statistics to this new cluster.
 4

5 Some pseudo-code for one step of the algorithm is shown in Algorithm 13, based on [Sud06, p94].
 6 (We update the nodes in random order to improve the mixing time, as suggested in [RS97b].) We
 7 can initialize the sample by sequentially sampling from $p(z_i|z_{1:i-1}, \mathbf{x}_{1:i})$. In the case of GMMs, both
 8 the naive sampler and collapsed sampler take $O(NKD)$ time per step.
 9

10 **Algorithm 13:** Collapsed Gibbs sampler for a mixture model

11 1 **for** each $i = 1 : N$ in random order **do**
 12 2 Remove \mathbf{x}_i 's sufficient statistics from old cluster z_i ;
 13 3 **for** each $k = 1 : K$ **do**
 14 4 | Compute $p_k(\mathbf{x}_i|\boldsymbol{\beta}) = p(\mathbf{x}_i|\{\mathbf{x}_j : z_j = k, j \neq i\}, \boldsymbol{\beta})$;
 15 5 | Compute $p(z_i = k|z_{-i}, \alpha) \propto (N_{k,-i} + \alpha/K)p_k(\mathbf{x}_i)$;
 16 6 | Sample $z_i \sim p(z_i|\cdot)$;
 17 7 | Add \mathbf{x}_i 's sufficient statistics to new cluster z_i

19
 20 A comparison of this method with the standard Gibbs sampler is shown in Figure 12.10. The
 21 vertical axis is the data log probability at each iteration, computed using
 22

23
$$\log p(\mathcal{D}|\mathbf{z}, \boldsymbol{\theta}) = \sum_{i=1}^N \log [\pi_{z_i} p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i})] \quad (12.60)$$

 24
 25

26 (To compute this quantity using the collapsed sampler, we have to sample $\boldsymbol{\theta} = (\boldsymbol{\pi}, \boldsymbol{\theta}_{1:K})$ given the
 27 data and the current assignment \mathbf{z} .) We see that the collapsed sampler does indeed generally work
 28 better than the vanilla sampler.

29 However, the primary advantage of using the collapsed sampler is that it extends to the case where
 30 we have an “infinite” number of mixture components, i.e., where we use a non-parametric Bayesian
 31 formulation with a prior such as the **Dirichlet process**. For details, see Section 33.2.4.1.

32
 33 **12.3.8.2 Example: Collapsed Gibbs sampling for LDA**

34
 35 In the supplementary material, we give an example of collapsed Gibbs sampling for fitting the **latent**
 36 **Dirichlet allocation (LDA)** model of [BNJ03a].

37

38 **12.4 Auxiliary variable MCMC**

39

40 Sometimes we can dramatically improve the efficiency of sampling by introducing dummy **auxiliary**
 41 **variables**, in order to reduce correlation between the original variables. If the original variables
 42 are denoted by \mathbf{x} , and the auxiliary variables by \mathbf{v} , we require that $\sum_{\mathbf{v}} p(\mathbf{x}, \mathbf{v}) = p(\mathbf{x})$, and that
 43 $p(\mathbf{x}, \mathbf{v})$ is easier to sample from than just $p(\mathbf{x})$. If we meet these two conditions, we can sample in the
 44 enlarged model, and then throw away the sampled \mathbf{v} values, thereby recovering samples from $p(\mathbf{x})$.
 45 Annealed importance sampling (Section 11.5.4) is one example. We give some MCMC examples
 46 below.

47

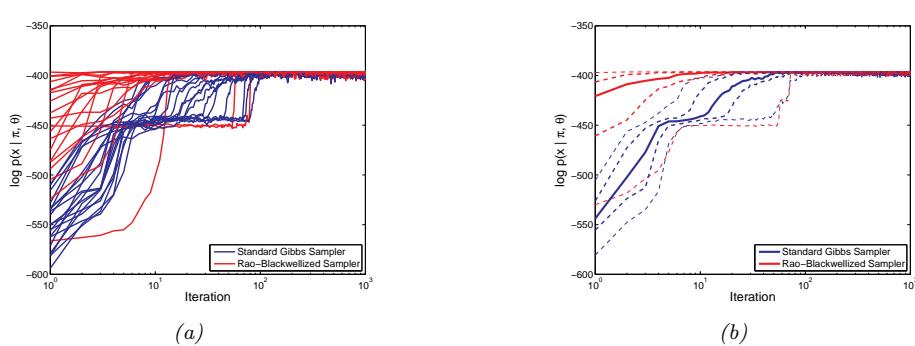


Figure 12.10: Comparison of collapsed (red) and vanilla (blue) Gibbs sampling for a mixture of $K = 4$ two-dimensional Gaussians applied to $N = 300$ data points. We plot log probability of the data vs iteration. (a) 20 different random initializations. (b) logprob averaged over 100 different random initializations. Solid line is the median, thick dashed in the 0.25 and 0.75 quantiles, and thin dashed are the 0.05 and 0.95 quintiles. From Figure 2.20 of [Sud06]. Used with kind permission of Erik Sudderth.

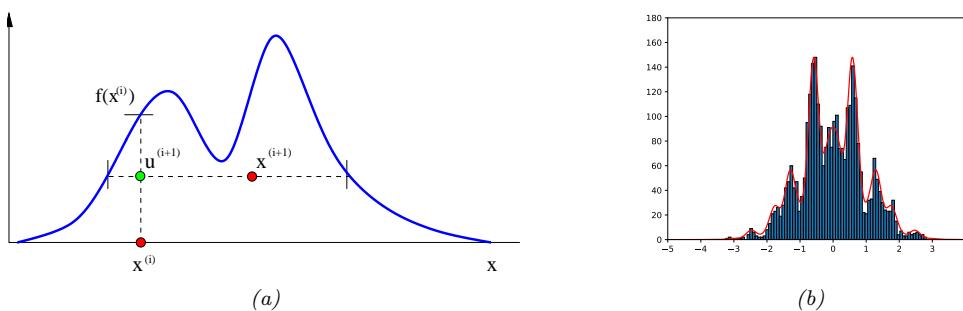


Figure 12.11: Slice sampling. (a) Illustration of one step of the algorithm in 1d. Given a previous sample x^i , we sample u^{i+1} uniformly on $[0, f(x^i)]$, where $f = \tilde{p}$ is the (unnormalized) target density. We then sample x^{i+1} along the slice where $f(x) \geq u^{i+1}$. From Figure 15 of [And+03]. Used with kind permission of Nando de Freitas. (b) Output of slice sampling applied to a 1d distribution. Generated by `slice_sampling_demo_1d.py`.

12.4.1 Slice sampling

Consider sampling from a univariate, but multimodal, distribution $p(x) = \tilde{p}(x)/Z_p$, where $\tilde{p}(x)$ is unnormalized, and $Z_p = \int \tilde{p}(x)dx$. We can sometimes improve the ability to make large moves by adding a uniform auxiliary variable v . We define the joint distribution as follows:

$$\hat{p}(x, v) = \begin{cases} 1/Z_p & \text{if } 0 \leq v \leq \tilde{p}(x) \\ 0 & \text{otherwise} \end{cases} \quad (12.61)$$

The marginal distribution over x is given by

$$\int \hat{p}(x, v) dv = \int_0^{\tilde{p}(x)} \frac{1}{Z_p} dv = \frac{\tilde{p}(x)}{Z_p} = p(x) \quad (12.62)$$

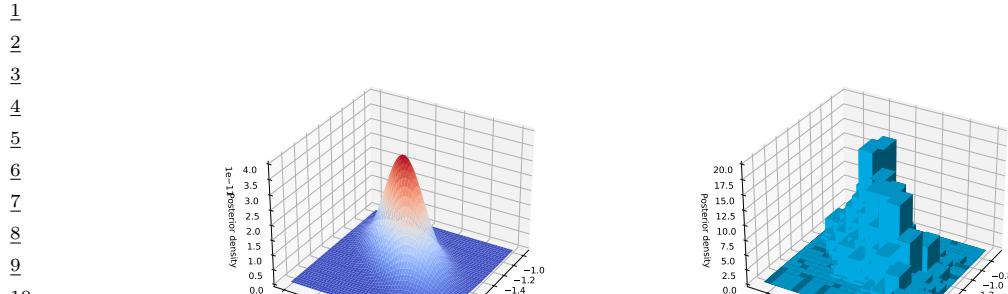


Figure 12.12: Binomial regression for 1d data. Left: Grid approximation to posterior. Right: Slice sampling approximation. Generated by [slice_sampling_demo_2d.py](#).

so we can sample from $p(x)$ by sampling from $\hat{p}(x, v)$ and then ignoring v .

We can sample from $\hat{p}(x, v)$ using Gibbs sampling. The full conditionals have the form

$$p(v|x) = U_{[0, \tilde{p}(x)]}(v) \quad (12.63)$$

$$p(x|v) = U_A(x) \quad (12.64)$$

where $A = \{x : \tilde{p}(x) \geq v\}$ is the set of points on or above the chosen height v . This corresponds to a slice through the distribution, hence the term **slice sampling** [Nea03]. See Figure 12.11(a).

In practice, it can be difficult to identify the set A . So we can use the following approach: construct an interval $x_{min} \leq x \leq x_{max}$ around the current point x^s of some width. We then test to see if each end point lies within the slice. If it does, we keep extending in that direction until it lies outside the slice. This is called **stepping out**. A candidate value x' is then chosen uniformly from this region. If it lies within the slice, it is kept, so $x^{s+1} = x'$. Otherwise we shrink the region such that x' forms one end and such that the region still contains x^s . Then another sample is drawn. We continue in this way until a sample is accepted.

To apply the method to multivariate distributions, we can sample one extra auxiliary variable for each dimension. The advantage of slice sampling over Gibbs is that it does not need a specification of the full-conditionals, just the unnormalized joint. The advantage of slice sampling over MH is that it does not need a user-specified proposal distribution (although it does require a specification of the width of the stepping out interval).

Figure 12.11(b) illustrates the algorithm in action on a synthetic 1d problem. Figure 12.12 illustrates its behavior on a slightly harder problem, namely binomial logistic regression. The model has the form $y_i \sim \text{Bin}(n_i, \text{logit}(\beta_1 + \beta_2 x_i))$. We use a vague Gaussian prior for the β_j 's. Figure 12.12(a) shows a grid-based approximation to the posterior, and Figure 12.12(b) shows a sample-based approximation. In this example, the grid is faster to compute, but for any problem with more than 2 dimensions, the grid approach is infeasible.

47

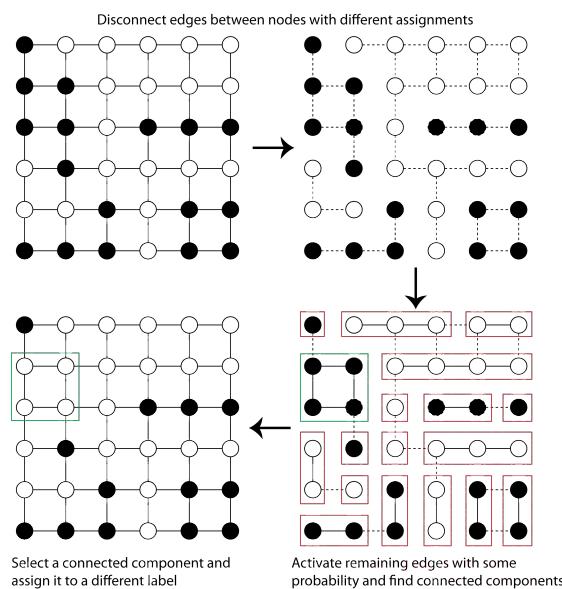


Figure 12.13: Illustration of the Swendsen Wang algorithm on a 2d grid. Used with kind permission of Kevin Tang.

12.4.2 Swendsen Wang

Consider an Ising model of the following form: $p(\mathbf{x}) = \frac{1}{Z} \prod_e \Psi(\mathbf{x}_e)$, where $\mathbf{x}_e = (x_i, x_j)$ for edge $e = (i, j)$, $x_i \in \{+1, -1\}$, and the edge potential is defined by $[e^J, e^{-J}, e^{-J}, e^J]$, where J is the edge strength. In Section 12.3.3, we discussed how to apply Gibbs sampling to this model. However, this can be slow when J is large in absolute value, because neighboring states can be highly correlated. The **Swendsen Wang** algorithm [SW87] is an auxiliary variable MCMC sampler which mixes much faster, at least for the case of attractive or ferromagnetic models, with $J > 0$.

Suppose we introduce auxiliary binary variables, one per edge.⁴ These are called **bond variables**, and will be denoted by \mathbf{v} . We then define an extended model $p(\mathbf{x}, \mathbf{v})$ of the form $p(\mathbf{x}, \mathbf{v}) = \frac{1}{Z'} \prod_e \Psi(\mathbf{x}_e, v_e)$, where $v_e \in \{0, 1\}$, and we define the new edge potentials as follows:

$$\Psi(\mathbf{x}_e, v_e = 0) = \begin{pmatrix} e^{-J} & e^{-J} \\ e^{-J} & e^{-J} \end{pmatrix}, \quad \Psi(\mathbf{x}_e, v_e = 1) = \begin{pmatrix} e^J - e^{-J} & 0 \\ 0 & e^J - e^{-J} \end{pmatrix} \quad (12.65)$$

It is clear that $\sum_{v_e=0}^1 \Psi(\mathbf{x}_e, v_e) = \Psi(\mathbf{x}_e)$, and hence that $\sum_{\mathbf{v}} p(\mathbf{x}, \mathbf{v}) = p(\mathbf{x})$, as required.

Fortunately, it is easy to apply Gibbs sampling to this extended model. The full conditional $p(\mathbf{v}|\mathbf{x})$ factorizes over the edges, since the bond variables are conditionally independent given the node variables. Furthermore, the full conditional $p(v_e|\mathbf{x}_e)$ is simple to compute: if the nodes on either end of the edge are in the same state ($x_i = x_j$), we set the bond v_e to 1 with probability $p = 1 - e^{-2J}$,

4. Our presentation of the method is based on some notes by David Mackay, available from <http://www.inference.phy.cam.ac.uk/mackay/itila/swendsen.pdf>.

1 otherwise we set it to 0. In Figure 12.13 (top right), the bonds that could be turned on (because
2 their corresponding nodes are in the same state) are represented by dotted edges. In Figure 12.13
3 (bottom right), the bonds that are randomly turned on are represented by solid edges.
4

5 To sample $p(\mathbf{x}|\mathbf{v})$, we proceed as follows. Find the connected components defined by the graph
6 induced by the bonds that are turned on. (Note that a connected component may consist of a
7 singleton node.) Pick one of these components uniformly at random. All the nodes in each such
8 component must have the same state, since the off-diagonal terms in the $\Psi(\mathbf{x}_e, v_e = 1)$ factor are 0.
9 Pick a state ± 1 uniformly at random, and force all the variables in this component to adopt this new
10 state. This is illustrated in Figure 12.13 (bottom left), where the green square denotes the selected
11 connected component, and we choose to force all nodes within it to enter the white state.

12 It should be intuitively clear that Swendsen Wang makes much larger moves through the state space
13 than Gibbs sampling. The gains are exponentially large for certain settings of the edge parameter.
14 More precisely, let the edge strength be parameterized by J/T , where $T > 0$ is a computational
15 temperature. For large T , the nodes are roughly independent, so both methods work equally well.
16 However, as T approaches a **critical temperature** T_c , the typical states of the system have very
17 long correlation lengths, and Gibbs sampling takes a very long time to generate independent samples.
18 As the temperature continues to drop, the typical states are either all on or all off. The frequency
19 with which Gibbs sampling moves between these two modes is exponentially small. By contrast, SW
20 mixes rapidly at all temperatures.

21 Unfortunately, if any of the edge weights are negative, $J < 0$, the system is **frustrated**, and there
22 are exponentially many modes, even at low temperature. SW does not work very well in this setting,
23 since it tries to force many neighboring variables to have the same state. In fact, computation in this
24 regime is provably hard for any algorithm [JS93; JS96].
25

26 12.5 Hamiltonian Monte Carlo (HMC) 27

28 Many MCMC algorithms perform poorly in high dimensional spaces, because they rely on a form
29 of random search based on local perturbations. In this section, we discuss a method known as
30 **Hamiltonian Monte Carlo** or **HMC**, that leverages gradient information to guide the local moves.
31 This is an auxiliary variable method (Section 12.4) deriving from physics [Dua+87; Nea93; Mac03;
32 Nea10; Bet17].⁵ In particular, the method builds on **Hamiltonian mechanics**, which we describe
33 below.
34

35 12.5.1 Hamiltonian mechanics

36 Consider a particle rolling around an energy landscape. We can characterize the motion of the particle
37 in terms of its position $\boldsymbol{\theta} \in \mathbb{R}^D$ (often denoted by \mathbf{q}) and its momentum $\mathbf{v} \in \mathbb{R}^D$ (often denoted by
38 \mathbf{p}). The set of possible values for $(\boldsymbol{\theta}, \mathbf{v})$ is called the **phase space**. We define the **Hamiltonian**
39 function for each point in phase space as follows:
40

$$\mathcal{H}(\boldsymbol{\theta}, \mathbf{v}) \triangleq \mathcal{E}(\boldsymbol{\theta}) + \mathcal{K}(\mathbf{v}) \quad (12.66)$$

41 42 43 44 45 5. The method was originally called **hybrid MC** [Dua+87]. It was introduced to the statistics community in [Nea93],
46 and was renamed to Hamiltonian MC in [Mac03].
47

where $\mathcal{E}(\boldsymbol{\theta})$ is the **potential energy**, $\mathcal{K}(\mathbf{v})$ is the **kinetic energy**, and the Hamiltonian is the total energy. In a physical setting, the potential energy is due to the pull of gravity, and the momentum is due to the motion of the particle. In a statistical setting, we often take the potential energy to be

$$\mathcal{E}(\boldsymbol{\theta}) = -\log \tilde{p}(\boldsymbol{\theta}) \quad (12.67)$$

where $\tilde{p}(\boldsymbol{\theta})$ is a possibly unnormalized distribution, such as $p(\boldsymbol{\theta}, \mathcal{D})$, and the kinetic energy to be

$$\mathcal{K}(\mathbf{v}) = \frac{1}{2} \mathbf{v}^\top \Sigma^{-1} \mathbf{v} \quad (12.68)$$

The energy of the moving particle is preserved. To see this, suppose the particle is started with zero momentum on the side of a slope; it will be pulled downwards, and its potential energy (from its initial height) will be transferred into kinetic energy (motion). Conversely, if the particle is moving along a flat plain and then encounters a slope upwards, it will slow down as it ascends, transferring its kinetic energy into potential energy.

More formally, the trajectory of a particle within an energy level set can be obtained by solving the following continuous time differential equations, known as **Hamilton's equations**:

$$\begin{aligned} \frac{d\boldsymbol{\theta}}{dt} &= \frac{\partial \mathcal{H}}{\partial \mathbf{v}} = \frac{\partial \mathcal{K}}{\partial \mathbf{v}} \\ \frac{d\mathbf{v}}{dt} &= -\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}} = -\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} \end{aligned} \quad (12.69)$$

To see why energy is conserved, note that

$$\frac{d\mathcal{H}}{dt} = \sum_{i=1}^D \left[\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{d\boldsymbol{\theta}_i}{dt} + \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} \frac{d\mathbf{v}_i}{dt} \right] = \sum_{i=1}^D \left[\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} - \frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} \right] = 0 \quad (12.70)$$

Note that the mapping from $(\boldsymbol{\theta}(t), \mathbf{v}(t))$ to $(\boldsymbol{\theta}(t+s), \mathbf{v}(t+s))$ for some time increment s is unique and is therefore invertible. Furthermore, this mapping is volume preserving, so has a Jacobian determinant of 1. (See e.g., [BZ20, p287] for a proof.) These facts will be important later when we turn this system into an MCMC algorithm.

12.5.2 Integrating Hamilton's equations

In this section, we discuss how to simulate Hamilton's equations in discrete time.

12.5.2.1 Euler's method

The simplest way to model the time evolution is to update the position and momentum simultaneously by a small amount, known as the step size η :

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \eta \frac{d\mathbf{v}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_t) = \mathbf{v}(t) - \eta \frac{\partial \mathcal{E}(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}} \quad (12.71)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \frac{d\boldsymbol{\theta}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_t) = \boldsymbol{\theta}_t + \eta \frac{\partial \mathcal{K}(\mathbf{v}_t)}{\partial \mathbf{v}} \quad (12.72)$$

1 If the kinetic energy has the form in Equation (12.68) then the second expression simplifies to
2

$$\underline{3} \quad \theta_{t+1} = \theta_t + \eta \Sigma^{-1} v_{t+1} \quad (12.73)$$

5 This is known as **Euler's method**. Unfortunately, Euler's method does not preserve volume, and
6 can lead to inaccurate approximations after only a few steps.
7

8 12.5.2.2 Modified Euler's method 9

10 A simple improve to Euler's method first updates the momentum, and then updates the position
11 using the new momentum:

$$\underline{12} \quad v_{t+1} = v_t + \eta \frac{d\mathbf{v}}{dt}(\theta_t, v_t) = v_t - \eta \frac{\partial \mathcal{E}(\theta_t)}{\partial \theta} \quad (12.74)$$

$$\underline{13} \quad \theta_{t+1} = \theta_t + \eta \frac{d\theta}{dt}(\theta_t, v_{t+1}) = \theta_t + \eta \frac{\partial \mathcal{K}(v_{t+1})}{\partial v} \quad (12.75)$$

17 This is known as the **Modified Euler's method**. (We can equivalently perform the updates in
18 the opposite order.) This small change ensures the mapping is **symplectic** (volume preserving, see
19 https://en.wikipedia.org/wiki/Symplectic_integrator), and leads to more accurate results.
20

21 12.5.2.3 Leapfrog integrator 22

23 Due to the asymmetry of the updates, the modified Euler method is not reversible. We can fix this
24 by performing a “half” update of the momentum, a full update of the position, and then another
25 “half” update of the momentum:

$$\underline{26} \quad v_{t+1/2} = v_t - \frac{\eta}{2} \frac{\partial \mathcal{E}(\theta_t)}{\partial \theta} \quad (12.76)$$

$$\underline{27} \quad \theta_{t+1} = \theta_t + \eta \frac{\partial \mathcal{K}(v_{t+1/2})}{\partial v} \quad (12.77)$$

$$\underline{28} \quad v_{t+1} = v_{t+1/2} - \frac{\eta}{2} \frac{\partial \mathcal{E}(\theta_{t+1})}{\partial \theta} \quad (12.78)$$

33 This is known as the **Leapfrog integrator**. If we perform multiple leapfrog steps, it is equivalent
34 to performing a half step update of \mathbf{v} at the beginning and end of the trajectory, and alternating
35 between full step updates of θ and \mathbf{v} in between.
36

37 It can be shown that the leapfrog integrator is volume preserving. Furthermore, if we reverse
38 the momentum at the end (by replacing it with $-\mathbf{v}$), the method is also reversible. However, if
39 the kinetic energy satisfies $\mathcal{K}(\mathbf{v}) = \mathcal{K}(-\mathbf{v})$, this reversal is not needed. Unfortunately, this method
40 does not exactly conserve energy, due to the finite step size. We will correct for this by treating the
41 method as a proposal distribution, and using the MH acceptance criterion to ensure we sample from
42 the right distribution, as we discuss in Section 12.5.3.
43

44 12.5.2.4 Higher order integrators 45

46 It is possible to define higher order integrators that are more accurate, but take more steps. For
47 details, see [BRSS18].

1 **12.5.3 The HMC algorithm**

2 We now describe how to use Hamiltonian dynamics to define an MCMC sampler in the expanded
3 state space $(\boldsymbol{\theta}, \mathbf{v})$. The target distribution has the form
4

5

$$p(\boldsymbol{\theta}, \mathbf{v}) = \frac{1}{Z} \exp[-\mathcal{H}(\boldsymbol{\theta}, \mathbf{v})] = \frac{1}{Z} \exp \left[-\mathcal{E}(\boldsymbol{\theta}) - \frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v} \right] \quad (12.79)$$

6 The marginal distribution over the latent variables of interest has the form
7

8

$$p(\boldsymbol{\theta}) = \int p(\boldsymbol{\theta}, \mathbf{v}) d\mathbf{v} = \frac{1}{Z_q} e^{-\mathcal{E}(\boldsymbol{\theta})} \int \frac{1}{Z_p} e^{-\frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v}} d\mathbf{v} = \frac{1}{Z_q} e^{-\mathcal{E}(\boldsymbol{\theta})} \quad (12.80)$$

9 Suppose the previous state of the Markov chain is $(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$. To sample the next state, we
10 proceed as follows. First we sample a new random momentum, $\mathbf{v}_{t-1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ and then we compute
11 the new state of the inner (proposal) loop: $(\boldsymbol{\theta}'_0, \mathbf{v}'_0) = (\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$. We then perform L leapfrog
12 steps to get the final proposed state $(\boldsymbol{\theta}^*, \mathbf{v}^*) = (\boldsymbol{\theta}_L, \tilde{\mathbf{v}}_L)$. Next we compute the MH acceptance
13 probability
14

15

$$\alpha = \min \left(1, \frac{p(\boldsymbol{\theta}^*, \mathbf{v}^*)}{p(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})} \right) = \min (1, \exp [-\mathcal{H}(\boldsymbol{\theta}^*, \mathbf{v}^*) + \mathcal{H}(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})]) \quad (12.81)$$

16 (The transition probabilities cancel since the proposal is reversible.) Finally, we accept the proposal
17 by setting $(\boldsymbol{\theta}_t, \mathbf{v}_t) = (\boldsymbol{\theta}^*, \mathbf{v}^*)$ with probability α , otherwise we set $(\boldsymbol{\theta}_t, \mathbf{v}_t) = (\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$. (In practice
18 we don't need to keep the momentum term, it is only used inside of the leapfrog algorithm.) See
19 Algorithm 14 for the pseudocode.

20 **Algorithm 14:** Hamiltonian Monte Carlo

```

21 1 for  $t = 1 : T$  do
22 2   Generate random momentum  $\mathbf{v}_{t-1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$  ;
23 3   Set  $(\boldsymbol{\theta}'_0, \mathbf{v}'_0) = (\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$  ;
24 4   Half step for momentum:  $\mathbf{v}'_{\frac{1}{2}} = \mathbf{v}'_0 - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}'_0)$  ;
25 5   for  $l = 1 : L - 1$  do
26 6      $\boldsymbol{\theta}'_l = \boldsymbol{\theta}'_{l-1} + \eta \boldsymbol{\Sigma}^{-1} \mathbf{v}'_{l-1/2}$  ;
27 7      $\mathbf{v}'_{l+1/2} = \mathbf{v}'_{l-1/2} - \eta \nabla \mathcal{E}(\boldsymbol{\theta}'_l)$ 
28 8   Full step for location:  $\boldsymbol{\theta}'_L = \boldsymbol{\theta}'_{L-1} + \eta \boldsymbol{\Sigma}^{-1} \mathbf{v}'_{L-1/2}$  ;
29 9   Half step for momentum:  $\mathbf{v}'_L = \mathbf{v}'_{L-1/2} - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}'_L)$  ;
30 10  Compute proposal  $(\boldsymbol{\theta}^*, \mathbf{v}^*) = (\boldsymbol{\theta}'_L, \mathbf{v}'_L)$  ;
31 11  Compute  $\alpha = \min (1, \exp [-\mathcal{H}(\boldsymbol{\theta}^*, \mathbf{v}^*) + \mathcal{H}(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})])$  ;
32 12  Set  $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}^*$  with probability  $\alpha$ , otherwise  $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t$ .

```

33 We need to sample a new momentum at each iteration to satisfy ergodicity. To see why, recall that
34 $\mathcal{H}(\boldsymbol{\theta}, \mathbf{v})$ stays approximately constant as we move through phase space. If $\mathcal{H}(\boldsymbol{\theta}, \mathbf{v}) = \mathcal{E}(\boldsymbol{\theta}) + \frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v}$,
35 then clearly $\mathcal{E}(\boldsymbol{\theta}) \leq h = \mathcal{H}(\boldsymbol{\theta}, \mathbf{v})$ for all locations $\boldsymbol{\theta}$ along the trajectory. Thus the sampler cannot
36 reach states where $\mathcal{E}(\boldsymbol{\theta}) > h$. To ensure the sampler explores the full space, we must pick a random
37 momentum at the start of each iteration.

38

1 **12.5.4 Tuning HMC**

3 We need to specify three hyperparameters for HMC: the number of leapfrog steps L , the step size η ,
4 and the covariance Σ .

6 **12.5.4.1 Choosing the number of steps using NUTS**

8 We want to choose the number of leapfrog steps L to be large enough that the algorithm explores
9 the level set of constant energy, but without doubling back on itself, which would waste computation,
10 due to correlated samples. Fortunately, there is an algorithm, known as the **No-U-Turn Sampler**
11 or **NUTS** algorithm [HG14], which can adaptively choose L for us.

12 **12.5.4.2 Choosing the step size**

14 When $\Sigma = \mathbf{I}$, the ideal step size η should be roughly equal to the width of $\mathcal{E}(\boldsymbol{\theta})$ in the most constrained
15 direction of the local energy landscape. For a locally quadratic potential, this corresponds to the
16 square root of the smallest marginal standard deviation of the local covariance matrix. (If we think
17 of the energy surface as a valley, this corresponds to the direction with the steepest sides.) A step
18 size much larger than this will cause moves that are likely to be rejected because they move to places
19 which increase the potential energy too much. On the other hand, if the step size is too low, the
20 proposal distribution will not move much from the starting position, and the algorithm will be very
21 slow.

22 In [BZ20, Sec 9.5.4] they recommend the following heuristic for picking η : set $\Sigma = \mathbf{I}$ and $L = 1$,
23 and then vary η until the acceptance rates are in the range of 40%–80%. Of course, different step
24 sizes might be needed in different parts of the state space. In this case, we can use learning rate
25 schedules from the optimization literature, such as cyclical schedules [Zha+20d].

27 **12.5.4.3 Choosing the covariance (inverse mass) matrix**

29 To allow for larger step sizes, we can use a smarter choice for Σ , also called the **inverse mass**
30 **matrix**. A natural choice is to use the Hessian at the current location, to capture the local geometry:

32
$$\Sigma(\boldsymbol{\theta}) = \nabla^2 \mathcal{E}(\boldsymbol{\theta}) \tag{12.82}$$

34 Since this is not always positive definite, an alternative, that can be used for some problems, is to
35 use the Fisher information matrix (Section 2.6), given by

36
$$\Sigma(\boldsymbol{\theta}) = -\mathbb{E}_{p(\mathbf{x}|\boldsymbol{\theta})} [\nabla^2 \log p(\mathbf{x}|\boldsymbol{\theta})] \tag{12.83}$$

38 Since the initial momentum is sampled from $\mathcal{N}(\mathbf{0}, \Sigma)$, this ensures that the noise that we add to
39 the system follows the local covariance structure, taking bigger moves along “flat” directions, thus
40 exploring along valley floors, where there is highest posterior uncertainty.

41 One way to estimate a fixed Σ is to run HMC with $\Sigma = \mathbf{I}$ for a **warmup** period, and then to run for
42 a few more steps, so we can compute the empirical covariance matrix using $\Sigma = \mathbb{E}[(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})^T]$.
43 Another approach is to use an L-BFGS [ZS11]. In [Hof+19] they propose a method called **NeuTra**
44 **HMC** algorithm which “neutralizes” bad geometry by learning an inverse autoregressive flow model
45 (Section 24.2.4.3) in order to map the warped distribution to an isotropic Gaussian. This is often an
46 order of magnitude faster than vanilla HMC.

47

1 **12.5.5 Riemann Manifold HMC**

2 If we let the covariance matrix change as we move position, so Σ is a function of θ , the method
3 is known as **Riemann Manifold HMC** [GC11; Bet13], since the moves follow a curved manifold,
4 rather than the flat manifold induced by a constant Σ . In the RM-HMC case, the kinetic energy has
5 to be generalized to
6

7

$$\mathcal{K}(\theta, v) = \frac{1}{2} \log((2\pi)^D |\Sigma(\theta)|) + \frac{1}{2} v^\top \Sigma(\theta) v \quad (12.84)$$

8 Unfortunately the Hamiltonian updates of θ and v are no longer separable, making the RM-HMC
9 algorithm more complex (e.g., it requires a fixed point iteration and third order derivatives).
10

11 A simplification to this method is to update Σ at each step of the outer loop, but to keep it
12 constant during the leapfrog integration inner loop. In this case, we can use the standard HMC
13 method. (Updating Σ between proposals does not violate the Markov property, since the momentum
14 terms are not shared across outer loop steps, and the inner loop is reversible for any value of Σ .)
15

16 Since estimating a high dimensional covariance matrix can be computationally and statistically
17 inefficient, it is more common to reparameterize the problem in terms of $\Sigma = \Sigma^{1/2} \Sigma^{\frac{1}{2}}$, where $\Sigma^{\frac{1}{2}}$ is a
18 matrix square root of Σ . (For example, if $\Sigma = \mathbf{U} \Lambda \mathbf{U}^\top$, then $\Sigma^{\frac{1}{2}} = \mathbf{U} \Lambda^{\frac{1}{2}} \mathbf{U}^\top$, where $\Lambda^{\frac{1}{2}} = \text{diag}(\sqrt{\lambda_d})$
19 is a diagonal matrix with the square root of the eigenvalues.)
20

21 Instead of sampling from $v \sim \mathcal{N}(\mathbf{0}, \Sigma)$, we can equivalently sample from $v \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, providing
22 we modify the leapfrog steps as follows:

23

$$v_{t+1/2} = v_t - \frac{\eta}{2} \Sigma^{-\frac{1}{2}} \nabla \mathcal{E}(\theta_t) \quad (12.85)$$

24

$$\theta_{t+1} = \theta_t + \eta \Sigma^{-\frac{1}{2}} v_{t+1/2} \quad (12.86)$$

25

$$v_{t+1} = v_{t+1/2} - \frac{\eta}{2} \Sigma^{-\frac{1}{2}} \nabla \mathcal{E}(\theta_{t+1}) \quad (12.87)$$

26 The advantage of this formulation is that it is often easier to directly estimate $\Sigma^{-\frac{1}{2}}$ using methods
27 such as [ZS11]. Furthermore, if we use a diagonal approximation, we can avoid all matrix inversions.
28

29 **12.5.6 Langevin Monte Carlo (MALA)**

30 A special case of HMC occurs when we take $L = 1$ leapfrog steps. This is known as **Langevin**
31 **Monte Carlo (LMC)**, or **Metropolis Adjusted Langevin Algorithm (MALA)** [RT96]. The
32 inner loop now consists of the following steps:

- 33 • Sample momentum $v'_0 \sim \mathcal{N}(\mathbf{0}, \Sigma)$
- 34 • Set $\theta'_0 = \theta_{t-1}$
- 35 • Take half step for momentum: $v'_{\frac{1}{2}} = v'_0 - \frac{\eta}{2} \nabla \mathcal{E}(\theta'_0)$.
- 36 • Take full step for location: $\theta'_1 = \theta'_0 + \eta \Sigma^{-1} v'_{\frac{1}{2}}$.
- 37 • Take half step for momentum: $v'_1 = v'_{\frac{1}{2}} - \frac{\eta}{2} \nabla \mathcal{E}(\theta'_1)$.
- 38 • Propose $(\theta^*, v^*) = (\theta'_1, v'_1)$.
- 39 • Perform MH update.

40 We can simplify the 3 equations into 2 by substituting $v'_{\frac{1}{2}}$ directly into θ'_1 and v'_1 . This gives rise to
41 the simplified algorithm shown in Algorithm 15.
42

1
2 **Algorithm 15:** Langevin Monte Carlo
3 1 **for** $t = 1 : T$ **do**
4 2 Generate random momentum $\mathbf{v}_{t-1} \sim \mathcal{N}(\mathbf{0}, \Sigma)$;
5 3 $\boldsymbol{\theta}^* = \boldsymbol{\theta}_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\boldsymbol{\theta}_{t-1}) + \eta \Sigma^{-1} \mathbf{v}_{t-1}$;
6 4 $\mathbf{v}^* = \mathbf{v}_{t-1} - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_{t-1}) - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}^*)$;
7 5 Compute $\alpha = \min(1, \exp[-\mathcal{H}(\boldsymbol{\theta}^*, \mathbf{v}^*)] / \exp[-\mathcal{H}(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})])$;
8 6 Set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}^*$ with probability α , otherwise $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t$.
9

10

11

12

13 LMC has a different properties than “full” HMC: since the momenta are discarded after each step,
14 successive proposals are not encouraged to move in the same direction, and so LMC explores the
15 landscape in a random walk fashion (albeit guided by the gradient). On the other hand, the method
16 is simpler than HMC.

17 A further simplification of LMC is to eliminate the MH acceptance step. In this case, the update
18 becomes

19
20
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\boldsymbol{\theta}_{t-1}) + \eta \Sigma^{-1} \mathbf{v}_{t-1} \quad (12.88)$$

21
22
$$= \boldsymbol{\theta}_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\boldsymbol{\theta}_{t-1}) + \eta \sqrt{\Sigma^{-1}} \boldsymbol{\epsilon}_{t-1} \quad (12.89)$$

23
24 where $\mathbf{v}_{t-1} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ and $\boldsymbol{\epsilon}_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. This is just like gradient descent with added noise. If we
25 set Σ to be the Fisher information matrix, this becomes natural gradient descent (Section 6.4) with
26 added noise. If we approximate the gradient with a stochastic gradient, we get a method known as
27 SGLD, or Stochastic Gradient Langevin Descent (see Section 12.7.1 for details).

28 Now suppose $\Sigma = \mathbf{I}$, and we set $\eta = \sqrt{2}$. In continuous time, we get the following stochastic
29 differential equation (SDE), known as **Langevin diffusion**:

30

31
$$d\boldsymbol{\theta}_t = -\nabla \mathcal{E}(\boldsymbol{\theta}_t) dt + \sqrt{2} d\mathbf{B}_t \quad (12.90)$$

32

33 where \mathbf{B}_t represents D -dimensional **Brownian motion**. If we use this to generate the samples, the
34 method is known as the **Unadjusted Langevin Algorithm** or **ULA** [Par81]. It can be shown that
35 this process has $\pi(\boldsymbol{\theta})$ as its stationary distribution [RT96].

36

37 12.5.7 Connection between SGD and Langevin sampling

38

39 In this section, we discuss a deep connection between stochastic gradient descent (SGD) and Langevin
40 sampling, following the presentation of [BZ20, Sec 10.2.3].

41 Consider the minimization of the additive loss

42
43
$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^N \mathcal{L}_n(\boldsymbol{\theta}) \quad (12.91)$$

44

45
46 For example, we may define $\mathcal{L}_n(\boldsymbol{\theta}) = -\log p(y_n | \mathbf{x}_n, \boldsymbol{\theta})$. We will use a minibatch approximation to
47

1
2 the gradients:

3
4 $\nabla_B \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{n \in \mathcal{S}} \nabla \mathcal{L}_n(\boldsymbol{\theta})$ (12.92)
5

6 where $\mathcal{S} = \{i_1, \dots, i_B\}$ is a randomly chosen set of indices of size B . For simplicity of analysis, we
7 assume the indices are chosen with replacement from $\{1, \dots, N\}$.

8 Let us define

9 $\mathbf{v}_t = \sqrt{\eta} (\nabla \mathcal{L}(\boldsymbol{\theta}_t) - \nabla_B \mathcal{L}(\boldsymbol{\theta}_t))$ (12.93)
10

11 Then we can rewrite the SGD update as

12 $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_B \mathcal{L}(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t) + \sqrt{\eta} \mathbf{v}_t$ (12.94)
13

14 The **diffusion term** \mathbf{v}_t has mean 0, since

15
16 $\mathbb{E} [\nabla_B \mathcal{L}(\boldsymbol{\theta})] = \frac{1}{B} \sum_{j=1}^B \mathbb{E} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})] = \frac{1}{B} \sum_{j=1}^B \nabla \mathcal{L}(\boldsymbol{\theta}) = \nabla \mathcal{L}(\boldsymbol{\theta})$ (12.95)
17
18

19 To compute the variance of the diffusion term, note that

20
21 $\mathbb{V} [\nabla_B \mathcal{L}(\boldsymbol{\theta})] = \frac{1}{B^2} \sum_{j=1}^B \mathbb{V} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})]$ (12.96)
22
23

24 where

25 $\mathbb{V} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})] = \mathbb{E} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta}) \nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})^\top] - \mathbb{E} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})] \mathbb{E} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})^\top]$ (12.97)
26

27 $= \left(\frac{1}{N} \sum_{n=1}^N \nabla \mathcal{L}_n(\boldsymbol{\theta}) \nabla \mathcal{L}_n(\boldsymbol{\theta})^\top \right) - \nabla \mathcal{L}(\boldsymbol{\theta}) \nabla \mathcal{L}(\boldsymbol{\theta})^\top \triangleq \mathbf{D}(\boldsymbol{\theta})$ (12.98)
28
29

30 where $\mathbf{D}(\boldsymbol{\theta})$ is called the **diffusion matrix**. Hence $\mathbb{V} [\mathbf{v}_t] = \frac{\eta}{B} \mathbf{D}(\boldsymbol{\theta}_t)$.

31 [LTW15] prove that the following continuous time stochastic differential equation (SDE) is a
32 first-order approximation of minibatch SGD (assuming the loss function is Lipschitz continuous):

33
34 $d\boldsymbol{\theta}(t) = -\nabla \mathcal{L}(\boldsymbol{\theta}(t)) dt + \sqrt{\frac{\eta}{B} \mathbf{D}(\boldsymbol{\theta}_t)} d\mathbf{B}(t)$ (12.99)
35

36 where $\mathbf{B}(t)$ is **Brownian motion**. (See also [Hu+17] for additional analysis.) The scale factor for
37 the noise, $\tau = \frac{\eta}{B}$, plays the role of **temperature**. Thus we see that using a smaller batch size is
38 like using a larger temperature; the added noise ensures that SGD avoids going into narrow ravines,
39 and instead spends most of its time in **flat minima** which have better generalization performance
40 [Kes+17]. See Section 17.5.1 for more discussion of this point.

41 The covariance structure of the noise, $\mathbf{D}(\boldsymbol{\theta})$, can also be analysed [Zha+18b]. Consider an SGD
42 process near a local minimum, where $\nabla \mathcal{L}(\boldsymbol{\theta}) \approx \mathbf{0}$. In this case, the diffusion matrix equals the
43 empirical Fisher information matrix:

44
45 $\mathbf{D}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \nabla \mathcal{L}_n(\boldsymbol{\theta}) \nabla \mathcal{L}_n(\boldsymbol{\theta})^\top \approx \mathbb{E} [\nabla \mathcal{L}_n(\boldsymbol{\theta}) \nabla \mathcal{L}_n(\boldsymbol{\theta})^\top] = \mathbf{F}(\boldsymbol{\theta})$ (12.100)
46
47

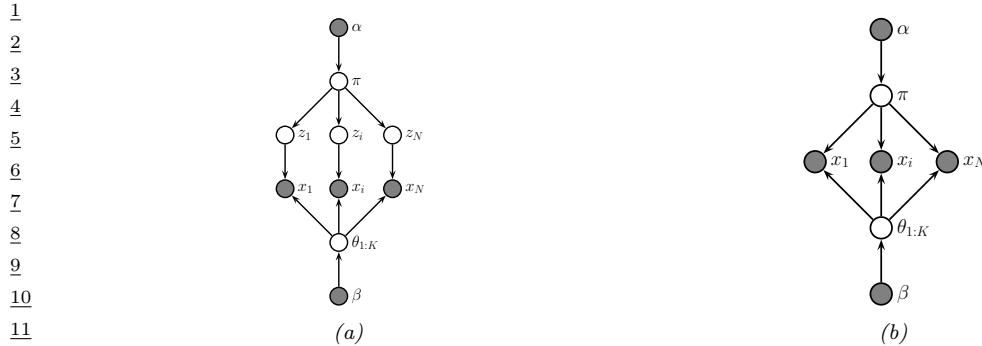


Figure 12.14: (a) A mixture model. (b) After integrating out the discrete latent variables.

From Equation (12.99), we see that SGD adds a noise term of the form $\sqrt{\mathbf{F}}\epsilon$, while from Equation (12.90), we see that Langevin MC adds a noise term of the form $\sqrt{\mathbf{F}^{-1}}\epsilon$, where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Thus SGD takes larger steps in directions with high eigenvalues, and smaller steps in directions with low eigenvalues, where LMC does the opposite.

The eigenvectors of the Fisher information matrix reflect the local structure of the loss landscape. Directions with large eigenvalues have large curvature (steep slopes), and correspond to directions where the output may change a lot in response to small changes in the parameters. These are the least robust directions. By “probing” for such directions, SGD will seek out “flat” local minima where nearly all directions are equally unconstrained. Such locations often generalize better (see Section 17.5.1 for more discussion of this point).

We see from Equation (12.99) that SGD generates a sequence of random iterates. It is natural to ask what the steady state distribution is. If $\mathbf{D}(\boldsymbol{\theta}) = c\mathbf{I}$ for some constant c , then one can show [CS18] that the steady state has the form $p(\boldsymbol{\theta}) \propto \exp[-\mathcal{L}(\boldsymbol{\theta})/\tau]$. However, in general we will not have $\mathbf{D} \propto \mathbf{I}$. In this case, it is not clear what distribution SGD is sampling from. It is arguably better to use a sampler which does what we want, rather than relying on SGD, whose distribution depends on the unknown (and uncontrollable) covariance of the data distribution.

12.5.8 Applying HMC to constrained parameters

To apply HMC, we require that all the latent quantities be continuous (real-valued) and have unconstrained support, i.e., $\boldsymbol{\theta} \in \mathbb{R}^D$, so discrete latent variables need to be marginalized out.⁶ For example, consider a GMM. We can easily write the likelihood without discrete latents as follows:

$$p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.101)$$

The corresponding “collapsed” model is shown in Figure 12.14(b). (Note that this is the opposite of Section 12.3.8.1, where we integrated out the continuous parameters in order to apply Gibbs sampling to the discrete latents.) We can apply similar techniques to other discrete latent variable

⁶ Recently [NDL20; Zho20] present extensions of HMC to handle discrete variables.

models. For example, to apply HMC to HMMs, we can use the forwards algorithm (Section 8.3.1) to efficiently compute

$$p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{\mathbf{z}_{1:T}} p(\mathbf{x}_n, \mathbf{z}_{n,1:T} | \boldsymbol{\theta}) \quad (12.102)$$

In addition to marginalizing out any discrete latent variables, we need to ensure the remaining continuous latent variables are unconstrained. This often requires performing a change of variables using a bijector. For example, instead of sampling the discrete probability vector from the probability simplex $\boldsymbol{\pi} \in \mathbb{S}^K$, we should sample the logits $\boldsymbol{\eta} \in \mathbb{R}^K$. After sampling, we can transform back, since bijectors are invertible.

12.5.9 Speeding up HMC

Although HMC uses gradient information to explore the typical set, sometimes the geometry of the typical set can be difficult to sample from. Recently [Hof+19] proposed the **NeuTra** HMC algorithm which “neutralizes” bad geometry by learning an inverse autoregressive flow model (Section 24.2.4.3) in order to map the warped distribution to an isotropic Gaussian. This is often an order of magnitude faster than vanilla HMC.

An orthogonal issue to the geometry of the space is the cost of evaluating the target distribution, $\mathcal{E}(\boldsymbol{\theta}) = -\log \tilde{p}(\boldsymbol{\theta})$. For many ML applications, this has the form $\log \tilde{p}(\boldsymbol{\theta}) = \log p_0(\boldsymbol{\theta}) + \sum_{n=1}^N \log p(\boldsymbol{\theta}_n | \boldsymbol{\theta})$. This takes $O(N)$ time to compute. We can speed this up by subsampling the data, although this introduces noise into the system. See Section 12.7 for details.

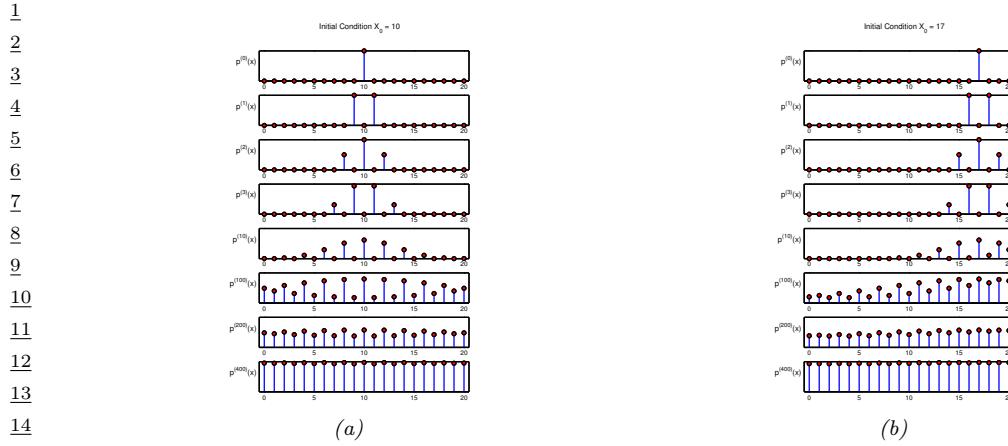
S

12.6 MCMC convergence

We start MCMC from an arbitrary initial state. As we explained in Section 2.8.4, the samples will be coming from the chain’s stationary distribution only when the chain has “forgotten” where it started from. The amount of time it takes to enter the stationary distribution is called the mixing time (see Section 12.6.1 for details). Samples collected before the chain has reached its stationary distribution do not come from p^* , and are usually thrown away. The initial period, whose samples will be ignored, is called the **burn-in phase**.

For example, consider a uniform distribution on the integers $\{0, 1, \dots, 20\}$. Suppose we sample from this using a symmetric random walk. In Figure 12.15, we show two runs of the algorithm. On the left, we start in state 10; on the right, we start in state 17. Even in this small problem it takes over 200 steps until the chain has “forgotten” where it started from. Proposal distributions that make larger changes can converge faster. For example, [BD92; Man] prove that it takes about 7 riffle shuffles to properly mix a deck of 52 cards (i.e., to ensure the distribution is uniform).

In Section 12.6.1 we discuss how to compute the mixing time theoretically. In practice, this can be very hard [BBM10] (this is one of the fundamental weaknesses of MCMC), so in Section 12.6.2, we discuss practical heuristics.



16 *Figure 12.15: Illustration of convergence to the uniform distribution over $\{0, 1, \dots, 20\}$ using a symmetric*
17 *random walk starting from (left) state 10, and (right) state 17. Adapted from Figures 29.14 and 29.15 of*
18 *[Mac03]. Generated by `random_walk_integers.py`.*

12.6.1 Mixing rates of Markov chains

22 The amount of time it takes for a Markov chain to converge to the stationary distribution, and forget
23 its initial state, is called the **mixing time**. More formally, we say that the mixing time from state
24 x_0 is the minimal time such that, for any constant $\epsilon > 0$, we have that

$$26 \quad \tau_\epsilon(x_0) \triangleq \min\{t : \|\delta_{x_0}(x)T^t - p^*\|_1 \leq \epsilon\} \quad (12.103)$$

27 where $\delta_{x_0}(x)$ is a distribution with all its mass in state x_0 , T is the transition matrix of the chain
28 (which depends on the target p^* and the proposal q), and $\delta_{x_0}(x)T^t$ is the distribution after t steps.
29 The mixing time of the chain is defined as

$$31 \quad \tau_\epsilon \triangleq \max_{x_0} \tau_\epsilon(x_0) \quad (12.104)$$

33 The mixing time is determined by the eigengap $\gamma = \lambda_1 - \lambda_2$, which is the difference of the first
34 and second eigenvalues of the transition matrix. In particular, one can show $\tau_\epsilon = O(\frac{1}{\gamma} \log \frac{n}{\epsilon})$, where
35 n is the number of states. Since computing the transition matrix (and hence its eigenvalues) can be
36 hard to do, especially for high dimensional and/or continuous state spaces, it is useful to find other
37 ways to estimate the mixing time.

38 One way to estimate the mixing time is to examine the geometry of the state space. For example,
39 consider the chain in Figure 12.16. We see that the state space consists of two “islands”, each of
40 which is connected via a narrow “bottleneck”. (If they were completely disconnected, the chain
41 would not be ergodic, and there would no longer be a unique stationary distribution, as discussed
42 in Section 2.8.4.3.) We define the **conductance** ϕ of a chain as the minimum probability, over all
43 subsets S of states, of transitioning from that set to its complement:

$$45 \quad \phi \triangleq \min_{S: 0 \leq p^*(S) \leq 0.5} \frac{\sum_{x \in S, x' \in S^c} T(x \rightarrow x')}{p^*(S)}, \quad (12.105)$$

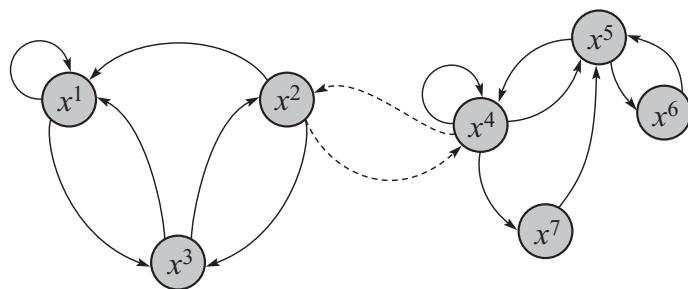


Figure 12.16: A Markov chain with low conductance. The dotted arcs represent transitions with very low probability. From Figure 12.6 of [KF09a]. Used with kind permission of Daphne Koller.

One can show that $\tau_\epsilon \leq O\left(\frac{1}{\phi^2} \log \frac{n}{\epsilon}\right)$. Hence chains with low conductance have high mixing time. For example, distributions with well-separated modes usually have high mixing time. Simple MCMC methods, such as MH and Gibbs, often do not work well in such cases, and more advanced algorithms, such as parallel tempering, are necessary (see e.g., [ED05; Kat+06; BZ20]).

12.6.2 Practical convergence diagnostics

Computing the mixing time of a chain is in general quite difficult, since the transition matrix is usually very hard to compute. In practice various heuristics have been proposed to diagnose convergence — see e.g., [Gey92; CC96; BR98; Veh+19]. Strictly speaking, these methods do not diagnose convergence, but rather non-convergence. That is, if the chain has not converged, this will be detected, but the method may claim the chain has not converged when in fact it has. This is a flaw common to all convergence diagnostics, since diagnosing convergence is computationally intractable in general [BBM10]. Despite this warning, we briefly summarize some “best practices” (based on [Veh+19]) below.

12.6.2.1 Trace plots

One of the simplest approaches to assessing if the method has converged is to run multiple chains (typically 3 or 4) from very different **overdispersed** starting points, and to plot the samples of some quantity of interest, such as the value of a certain component of the state vector, or some event such as the value taking on an extreme value. This is called a **trace plot**. If the chain has mixed, it should have “forgotten” where it started from, so the trace plots should converge to the same distribution, and thus overlap with each other.

To illustrate this, we will consider a very simple, but enlightening, example from [McE20, Sec 9.5]. The model is a univariate Gaussian, $y_i \sim \mathcal{N}(\alpha, \sigma)$, with just 2 observations, $y_1 = -1$ and $y_2 = +1$. We first consider a very diffuse prior, $\alpha \sim \mathcal{N}(0, 1000)$ and $\sigma \sim \text{Expon}(0.0001)$, both of which allow for very large values of α and σ . We fit the model using HMC using 3 chains and 500 samples. The result is shown in Figure 12.17. On the right, we show the trace plots for α and σ for 3 different chains. We see that they do not overlap much with each other. In addition, the numerous black vertical lines at the bottom indicate that HMC had many divergences.

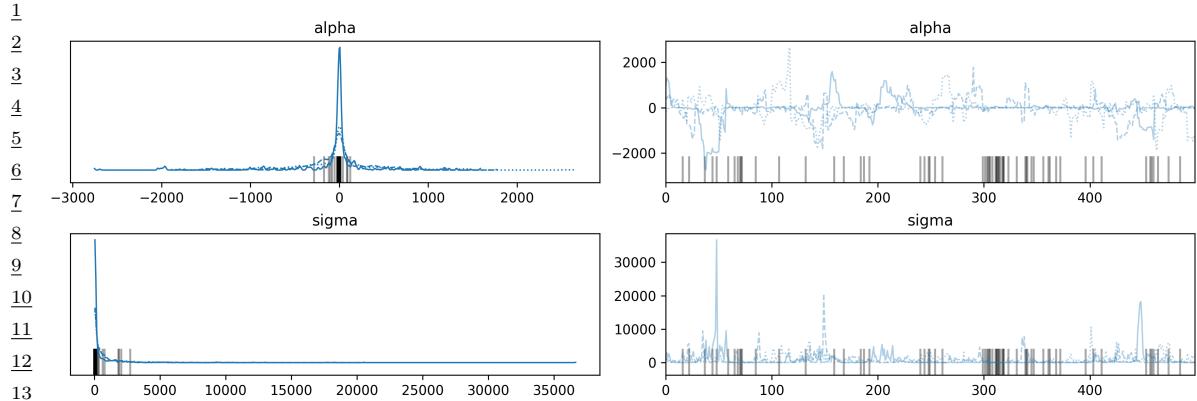


Figure 12.17: Marginals (left) and trace plot (right) for the univariate Gaussian using the diffuse prior. Black vertical lines indicate HMC divergences. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss_numpyro.py`.

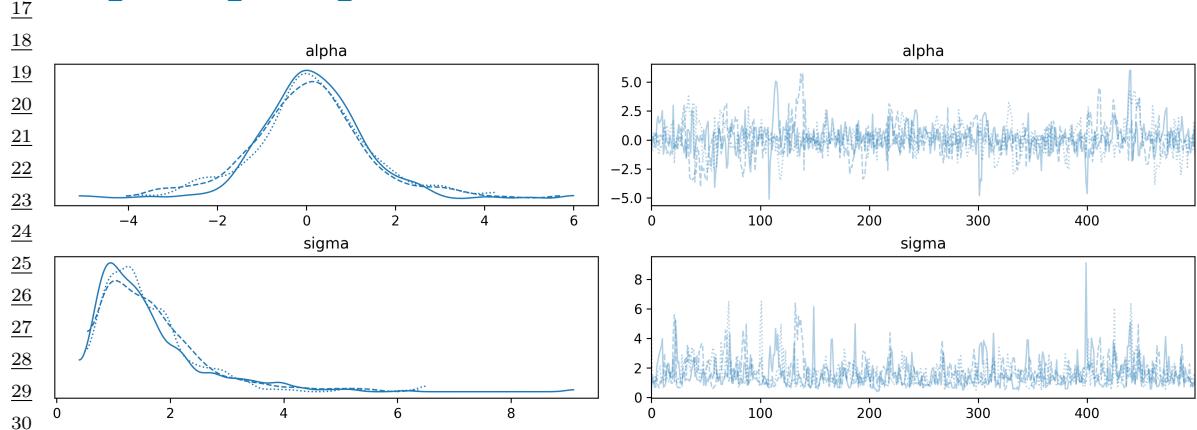


Figure 12.18: Marginals (left) and trace plot (right) for the univariate Gaussian using the sensible prior. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss_numpyro.py`.

The problem is caused by the overly diffuse priors, which do not get overwhelmed by the likelihood because we only have 2 data points. Thus the posterior is also diffuse, and has support over infinitely large values. We can fix this by using slightly stronger priors, that keep the parameters close to more sensible values. For example, suppose we use $\alpha \sim \mathcal{N}(1, 10)$ and $\sigma \sim \text{Expon}(1)$. Now we get the results in Figure 12.18. On the right we see that the traceplots overlap. On the left, we see that the marginal distributions from each chain have support over a reasonable interval, and have a peak at the “right” place (the MLE for α is 0, and for σ is 1). And we don’t see any divergence warnings.

Since trace plots of converging chains correspond to overlapping lines, it can be hard to distinguish success from failure. An alternative plot, known as a **trace rank plot**, was recently proposed in [Veh+19]. (In [McE20], this is called a **trankplot**, a term we borrow.) The idea is to compute the rank of each sample of a variable based on all the samples from all the chains. We then plot

47

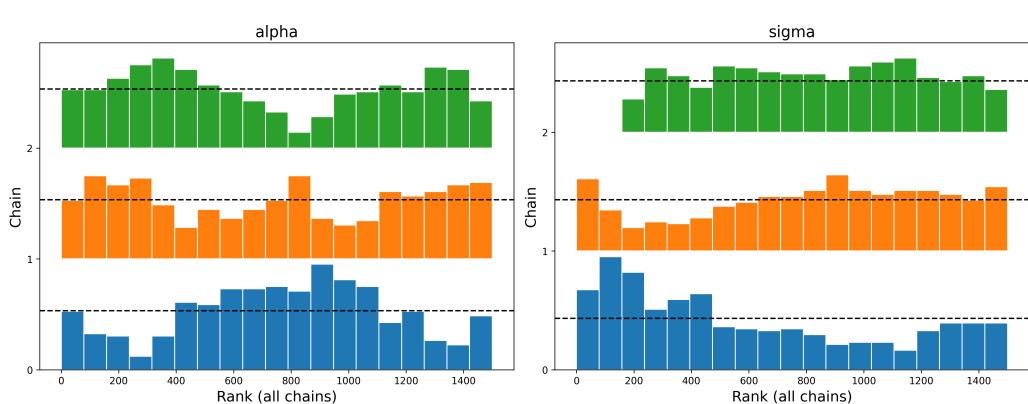


Figure 12.19: Trace rank plot for the univariate Gaussian using the diffuse prior. Black vertical lines indicate HMC divergences. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss_numpyro.py`.

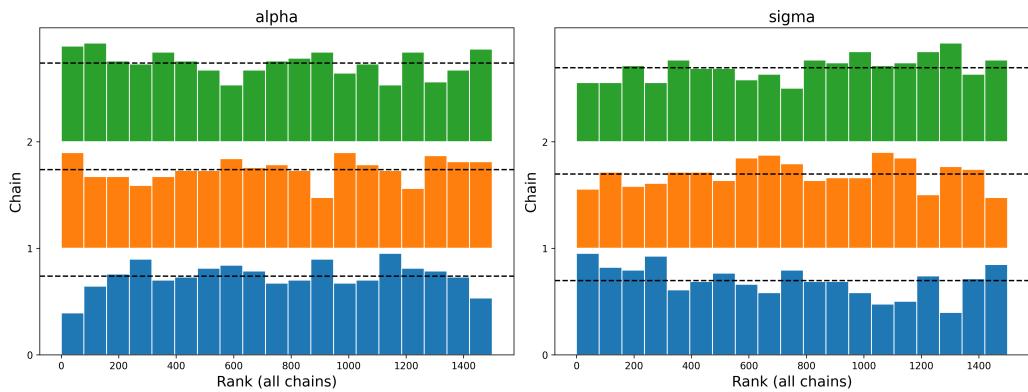


Figure 12.20: Trace rank plot for the univariate Gaussian using the sensible prior. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss_numpyro.py`.

a histogram of the ranks for each chain separately. If the chains have converged, the distribution over ranks should be uniform, since there should be no preference for high or low scoring samples amongst the chains.

The traceplot for the model with the diffuse prior is shown in Figure 12.19. (The x-axis is from 1 to the total number of samples, which in this example is 1500, since we use 3 chains and draw 500 samples from each.) We can see that the different chains are clearly not mixing. The traceplot for the model with the sensible prior is shown in Figure 12.20; this looks much better.

12.6.2.2 Estimated potential scale reduction (EPSR)

In this section, we discuss a way to assess convergence more quantitatively. The basic idea is that, if a set of simulations have not mixed well, then the variance of all the chains combined together will

1 be higher than the variance of the individual chains. So we will compare the variance of the quantity
2 of interest, call it x , computed between and within chains.
3

4 More precisely, suppose we have M chains, and we draw N samples (post burnin) from each. Let
5 x_{nm} denote the quantity of interest derived from the n 'th sample from the m 'th chain. We compute
6 the between and within-sequence variances as follows:
7

$$\underline{8} \quad B = \frac{N}{M-1} \sum_{m=1}^M (\bar{x}_{\cdot m} - \bar{x}_{\cdot \cdot})^2, \text{ where } \bar{x}_{\cdot m} = \frac{1}{N} \sum_{n=1}^N x_{nm}, \quad \bar{x}_{\cdot \cdot} = \frac{1}{M} \sum_{m=1}^M \bar{x}_{\cdot m} \quad (12.106)$$

$$\underline{9} \quad W = \frac{1}{M} \sum_{m=1}^M s_m^2, \text{ where } s_m^2 = \frac{1}{N-1} \sum_{n=1}^N (x_{nm} - \bar{x}_{\cdot m})^2 \quad (12.107)$$

10
11 The formula for s_m^2 is the usual unbiased estimate for the variance from a set of N samples; W is
12 just the average of this. The formula for B is similar, but scaled up by N since it is based on the
13 variance of $\bar{x}_{\cdot m}$, which are averaged over N values.

14 Let $\mathbb{V}[x] = \sigma^2$ be the true variance of the quantity of interest under the target distribution. The
15 within-chain variance W will generally underestimate V , because each chain may not have fully
16 explored the target distribution. However, one can show that the following estimate will overestimate
17 V , and hence is a conservative estimate:
18

$$\underline{19} \quad \hat{V}^+ \triangleq \frac{N-1}{N} W + \frac{1}{N} B \quad (12.108)$$

20 One can also show that $\hat{V}^+ \rightarrow \sigma^2$ as $N \rightarrow \infty$, so this is a consistent estimator.
21

22 We can estimate the benefits of running the chains for longer by comparing \hat{V}^+ to W . As $N \rightarrow \infty$,
23 this ratio will tend to 1. Thus we define the **estimated potential scale reduction**, also known as
24 **R-hat**, as follows:
25

$$\underline{26} \quad \hat{R} \triangleq \sqrt{\frac{\hat{V}^+}{W}} \quad (12.109)$$

27
28 In [Veh+19], they recommend checking if $\hat{R} < 1.01$ before declaring convergence.
29

30 For example, consider the \hat{R} values for various samplers for our univariate GMM example. In
31 particular, consider the 3 MH samplers in Figure 12.1, and the Gibbs sampler in Figure 12.7. The \hat{R}
32 values are 1.493, 1.039, 1.005 and 1.007. So this diagnostic has correctly identified that the first two
33 samplers are unreliable. The third sampler seems to be mixing, but the samples are highly correlated,
34 which reduces the effective sample size; we discuss this in Section 12.6.2.3.
35

36 Although the above definition can detect non-convergence of the kind shown in Figure 12.21(a),
37 it will fail on non-stationary chains of the kind shown in Figure 12.21(b). For this reason, it is
38 recommended to split each chain in the first and second halves, thus doubling M (but halving N),
39 and then to compute \hat{R} ; this is called the **split- \hat{R}** statistic.
40

41 In addition, the above definition can fail when estimating quantities which do not have finite means
42 or variances. To combat this, [Veh+19] suggest first computing the ranks r_{nm} of each sample x_{nm} ,
43 and then normalizing them using the inverse of the normal cdf, $z_{nm} = \Phi^{-1}((r_{nm} - 0.5)/S)$, and
44 finally computing the (split) \hat{R} from the z_{nm} values.
45

46
47

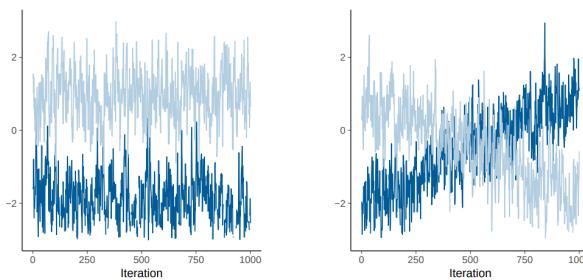


Figure 12.21: Examples of two distinct challenges that arise when trying to assess if Markov chain samples have converged. (a) Each chain looks stationary, but they have not converged to a common distribution. (b) The two sequences cover a common distribution, but are non-stationary. From Figure 1 of [Veh+19]. Used with kind permission of Aki Vehtari.

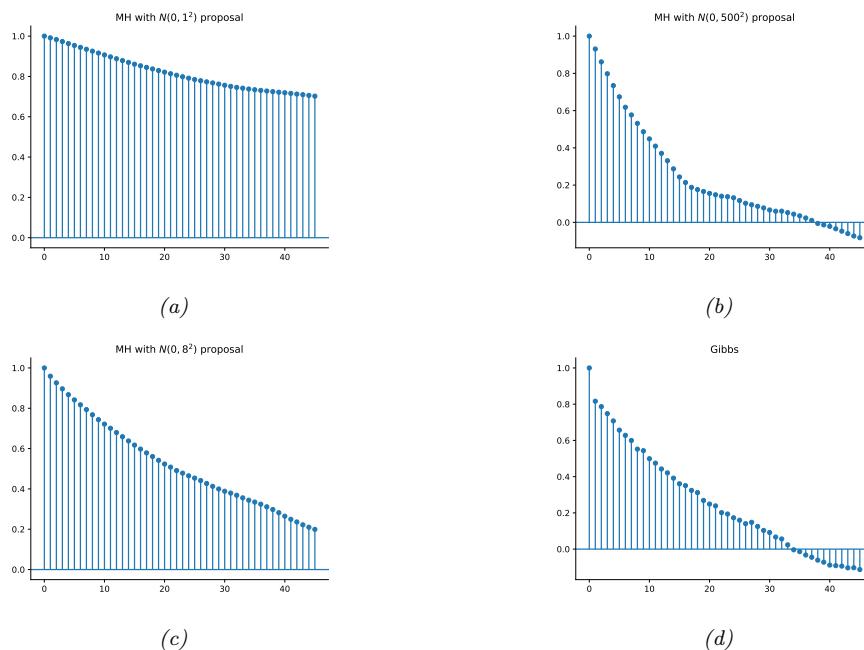


Figure 12.22: Autocorrelation functions for various MCMC samplers for the mixture of two 1D Gaussians. (a-c) These are the MH samplers in Figure 12.1. (d) This is the Gibbs sampler in Figure 12.7. Generated by `mcmc_gmm_demo.py`.

1
2 **12.6.2.3 Effective sample size**

3 Suppose we draw N independent samples from the target distribution, and let $\hat{x} = \frac{1}{N} \sum_{n=1}^N x_n$ be
4 our empirical estimate. Hence

5

$$\mathbb{V}[\hat{x}] = \frac{1}{N^2} \mathbb{V}\left[\sum_{n=1}^N x_n\right] = \frac{1}{N^2} \sum_{n=1}^N \mathbb{V}[x_n] = \frac{1}{N} \sigma^2 \quad (12.110)$$

6 where $\sigma^2 = \mathbb{V}[X]$. If the samples are correlated, as they usually will be when drawn from a Markov
7 chain, the variance of the estimate will be higher, as we show below.

8 Recall that for N (not necessarily independent) random variables we have

9

$$\mathbb{V}\left[\sum_{n=1}^N x_n\right] = \sum_{i=1}^N \sum_{j=1}^N \text{Cov}[x_i, x_j] = \sum_{i=1}^N \mathbb{V}[x_i] + 2 \sum_{1 \leq i < j \leq N} \text{Cov}[x_i, x_j] \quad (12.111)$$

10 where the second equality follows from the fact that $\text{Cov}[x_i, x_i] = \mathbb{V}[x_i]$. Let $\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$ be our
11 estimate based on these correlated samples. The variance of this estimate is given by

12

$$\mathbb{V}[\bar{x}] = \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N \text{Cov}[x_i, x_j] \quad (12.112)$$

13 We now try to rewrite this in a more convenient form. First recall that the correlation of x_i and
14 x_j is given by

15

$$\text{corr}[x_i, x_j] = \frac{\text{Cov}[x_i, x_j]}{\sqrt{\mathbb{V}[x_i] \mathbb{V}[x_j]}} \quad (12.113)$$

16 Since we assume we are drawing samples from the target distribution, we have $\mathbb{V}[x_i] = \sigma^2$, and hence

17

$$\mathbb{V}[\bar{x}] = \frac{\sigma^2}{N^2} \sum_{i=1}^N \sum_{j=1}^N \text{corr}[x_i, x_j] \quad (12.114)$$

18 For a fixed i , we can think of $\text{corr}[x_i, x_j]$ as a function of j . This will usually decay as j gets further
19 from i . Hence

20

$$\sum_{j=1}^N \text{corr}[x_i, x_j] \approx \sum_{\ell=-\infty}^{\infty} \text{corr}[x_i, x_{i+\ell}] = 1 + 2 \sum_{\ell=1}^{\infty} \text{corr}[x_i, x_{i+\ell}] \quad (12.115)$$

21 Since we assume the samples are coming from a stationary distribution, the index i does not matter.

22 Thus we can rewrite the above sum in terms of the **autocorrelation function**, defined as

23

$$\rho(\ell) \triangleq \text{corr}[x_0, x_\ell] \quad (12.116)$$

24 where ℓ is called the **lag**. The ACF can be computed efficiently by convolving the signal \mathbf{x} with itself.

25

In Figure 12.22, we plot the ACF for our four samplers for the Gaussian mixture model. We see that the ACF of the Gibbs sampler (bottom right) dies off to 0 much more rapidly than the MH samplers. Intuitively this indicates that each Gibbs sample is “worth” more than each MH sample. We quantify this below.

Define the **autocorrelation time** as $\rho = 1 + 2 \sum_{\ell=1}^{\infty} \rho(\ell)$. Hence the variance of our estimate can be rewritten as

$$\mathbb{V}[\bar{x}] = \frac{\sigma^2}{N^2} \sum_{i=1}^N \rho = \frac{\sigma^2}{N} \rho \quad (12.117)$$

By contrast, the variance of the estimate from independent samples is $\mathbb{V}[\hat{x}] = \frac{\sigma^2}{N}$. Thus we define the **effective sample size** our MCMC sampler to be

$$N_{\text{eff}} \triangleq \frac{N}{\rho} = \frac{N}{1 + 2 \sum_{\ell=1}^{\infty} \rho(\ell)} \quad (12.118)$$

In practice, we truncate the sum at lag L , which is the last integer at which $\rho(L)$ is positive. Also, if we run M chains, the numerator should be NM , so we get the following estimate:

$$\hat{N}_{\text{eff}} = \frac{NM}{1 + 2 \sum_{\ell=1}^L \hat{\rho}(\ell)} \quad (12.119)$$

We discuss how to estimate $\hat{\rho}(\ell)$ from multiple chains in Section 12.6.2.4.

In [Veh+19], they propose various extensions of the above estimator. In particular, they proposed to use rank statistics, to make the estimate more robust. They also discuss how to estimate the ESS for tail quantities, which is useful for estimating the reliability of Monte Carlo credible intervals.

12.6.2.4 Estimating the ACF from multiple chains using variograms

The obvious empirical estimator for the ACF in Equation (12.116) can be biased [Has97]. Furthermore, it is not clear how to use this estimator when we have multiple chains. So in this section, we present some estimation methods from the spatial statistics community that have been adopted by the Stan library [Car+17].

We start with some definitions and terminology. We say a stochastic process $\{X_t, t \in \mathcal{T}\}$ is **first order stationary** if the joint distribution of $(X_{t_1}, \dots, X_{t_n})$ is the same as the joint distribution of $(X_{t_1+\ell}, \dots, X_{t_n+\ell})$ for any set of indices t_1, \dots, t_n and offset ℓ . (Note that the index set \mathcal{T} could be one or two dimensional.) In the case where $n = 2$, this is equivalent to saying that $\text{Cov}[X_{t_1}, X_{t_2}]$ only depends on $t_2 - t_1$. We say the process is **second order stationary** if $\mathbb{E}[X_t] = \mu$ for all t , and $\text{Cov}[X_t, X_{t+\ell}] = \gamma(\ell)$ for all t , where $\gamma(\ell)$ is the **autocovariance function**. We define the autocorrelation function as $\rho(\ell) = \frac{\gamma(\ell)}{\gamma(0)}$, where $\gamma(0) = \sigma^2 = \mathbb{V}[X]$. If $\rho(\ell) = 0$ for all $\ell \neq 0$, the samples are uncorrelated; this corresponds to a **white noise process**.

Now define the **variogram** to be $V(\ell) \triangleq \mathbb{V}[X_{t+\ell} - X_t]$. (Another quantity that is used in the literature is the **semivariogram**, $C(\ell) \triangleq V(\ell)/2$.) Now we connect this to the ACF. For a stationary process we have

$$V(\ell) = \mathbb{V}[X_{t+\ell} - X_t] = \mathbb{V}[X_{t+\ell}] + \mathbb{V}[X_t] - 2\text{Cov}[X_{t+\ell}, X_t] = 2\sigma^2 - 2\rho(\ell) \quad (12.120)$$

1 so $\gamma(\ell) = \sigma^2 - \frac{V(\ell)}{2}$. We can compute an unbiased estimate of the variogram from N samples using
2

3

$$\hat{V}(\ell) = \frac{1}{N-\ell} \sum_{t=1}^{N-\ell} (X_{t+\ell} - X_t)^2 \quad (12.121)$$

4

5 since
6

7

$$\mathbb{V}[(X_{t+\ell} - X_t)] = \mathbb{E}[(X_{t+\ell} - X_t)^2] - \mathbb{E}[(X_{t+\ell} - X_t)]^2 = \mathbb{E}[(X_{t+\ell} - X_t)^2] \quad (12.122)$$

8

9 If we have M chains, and N samples from each, we can use
10

11

$$\hat{V}(\ell) = \frac{1}{M(N-\ell)} \sum_{m=1}^M \sum_{n=\ell+1}^N (X_{n,m} - X_{n-\ell,m})^2 \quad (12.123)$$

12

13 We can estimate σ^2 from multiple chains using \hat{V}^+ from Equation (12.108). Putting these together
14 gives our estimate for the ACF⁷
15

16

$$\hat{\rho}(\ell) = \frac{\gamma(\ell)}{\gamma(0)} = \frac{\sigma^2 - \hat{V}(\ell)/2}{\sigma^2} = 1 - \frac{\hat{V}(\ell)}{2\hat{V}^+} \quad (12.124)$$

17

22 12.6.3 Improving speed of convergence

23 There are many possible things you could try if the \hat{R} value is too large, and/or the effective sample
24 size is too low. Here is a brief list:
25

- 26
- 27 • Try using a non-centered parameterization (see Section 12.6.4).

28 - 29 • Try sampling variables in groups or blocks (see Section 12.3.7).

30 - 31 • Try using Rao-Blackwellisation, i.e., analytically integrating out some of the variables (see
32 Section 12.3.8).

33

 - 34 • Try adding auxiliary variables (see Section 12.4).

35

 - 36 • Try using adaptive proposal distributions (see Section 12.2.3.5).

37 More details can be found in [Rob+18].

38 12.6.4 Non-centered parameterizations and Neal's funnel

39 A common problem that arises with hierarchical Bayesian models is when a set of parameters at
40 one level of the model have a tight dependence on parameters at the level above. We saw examples
41 of this in the hierarchical Gaussian 8-schools example in Section 3.5.2.2 and the hierarchical radon
42 regression example in Section 15.5.3.2.

43

44 7. See also <https://bit.ly/2vhA1xa>.

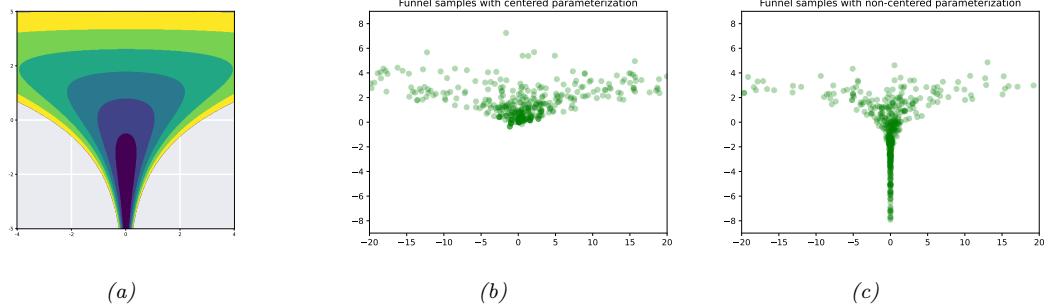


Figure 12.23: Neal’s funnel. (a) Joint density. Generated by `neals_funnel.py`. (a) HMC samples from centered representation. (b) HMC samples from non-centered representation. Generated by `funnel_numpyro.py`.

A simple toy model that captures the essence of the problem is the following distribution:

$$\nu \sim \mathcal{N}(0, 3) \tag{12.125}$$

$$x \sim \mathcal{N}(0, \exp(\nu)) \tag{12.126}$$

The corresponding joint density $p(x, \nu)$ is shown in Figure 12.23a. This is known **Neal’s funnel**, named after [Nea03]. It is hard for a sampler to “descend” in the narrow “neck” of the distribution, corresponding to areas where the variance ν is small [BG13].

Fortunately, we can represent this model in an equivalent way that makes it easier to sample from, providing we use a **non-centered parameterization** [PR03]. This has the form

$$\nu \sim \mathcal{N}(0, 3) \tag{12.127}$$

$$z \sim \mathcal{N}(0, 1) \tag{12.128}$$

$$x = z \exp(\nu) \tag{12.129}$$

This is easier to sample from, since $p(z, \nu)$ is a product of 2 independent Gaussians, and we can derive x deterministically from these Gaussian samples. The advantage of this reparameterization is shown in Figure 12.23.

Now consider a multidimensional example, where the regression weights β have a multivariate Gaussian prior, $\beta \sim \mathcal{N}(\mu, \Sigma)$. Let us parameterize Σ in terms of the standard deviations σ and the correlation matrix R , as in Equation (3.175). A centered representation would be as follows:

$$\mu \sim p(\mu) \tag{12.130}$$

$$\sigma \sim p(\sigma) \tag{12.131}$$

$$R \sim \text{LKJ}(R|\eta) \tag{12.132}$$

$$\Sigma = \text{diag}(\sigma) R \text{ diag}(\sigma) \tag{12.133}$$

$$\beta \sim \mathcal{N}(\mu, \Sigma) \tag{12.134}$$

1 We can write the non-centered representation as follows:
2

3 $\boldsymbol{\mu} \sim p(\boldsymbol{\mu})$ (12.135)
4

5 $\boldsymbol{\sigma} \sim p(\boldsymbol{\sigma})$ (12.136)
6

7 $\mathbf{L} \sim \text{LKJchol}(\mathbf{L}|\eta)$ (12.137)
8

9 $\mathbf{R} = \mathbf{L}\mathbf{L}^\top$ (12.138)
10

11 $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (12.139)
12

13 $\boldsymbol{\beta} = \boldsymbol{\mu} + \boldsymbol{\sigma}\mathbf{L}\mathbf{z}$ (12.140)
14

15 Here $\mathbf{L}\mathbf{L}^\top$ is the Cholesky decomposition of the correlation matrix \mathbf{R} , so \mathbf{L} is lower diagonal. The
16 distribution LKJchol is the distribution induced on \mathbf{L} by applying the LKJ distribution to \mathbf{R} . We
17 then just have to sample \mathbf{z} from a standard normal, and scale the results. This typically simplifies
18 the inference problem considerably.

19 A method to automatically derive such reparameterizations is discussed in [GMH20].
20

21 12.7 Stochastic gradient MCMC

22 Consider an unnormalized target distribution of the following form:
23

24
$$\pi(\boldsymbol{\theta}) \propto p(\boldsymbol{\theta}, \mathcal{D}) = p_0(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathbf{x}_n | \boldsymbol{\theta})$$
 (12.141)
25

26 where $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$. Alternatively we can define the target distribution in terms of an energy
27 function (negative log joint) as follows:

28
$$p(\boldsymbol{\theta}, \mathcal{D}) \propto \exp(-\mathcal{E}(\boldsymbol{\theta}))$$
 (12.142)
29

30 The energy function can be decomposed over data samples:
31

32
$$\mathcal{E}(\boldsymbol{\theta}) = \sum_{n=1}^N \mathcal{E}_n(\boldsymbol{\theta})$$
 (12.143)
33

34
$$\mathcal{E}_n(\boldsymbol{\theta}) = -\log p(\mathbf{x}_n | \boldsymbol{\theta}) - \frac{1}{N} \log p_0(\boldsymbol{\theta})$$
 (12.144)
35

36 Evaluating the full energy (e.g., to compute an acceptance probability in the Metropolis Hastings
37 algorithm, or to compute the gradient in HMC) takes $O(N)$ time, which does not scale to large data.
38 In this section, we discuss some solutions to this problem.
39

40 12.7.1 Stochastic Gradient Langevin Dynamics (SGLD)

41 Recall from Equation (12.90) that the (overdamped) **Langevin diffusion** SDE has the following
42 form

43
$$d\boldsymbol{\theta}_t = -\nabla \mathcal{E}(\boldsymbol{\theta}_t) dt + \sqrt{2} d\mathbf{W}_t$$
 (12.145)
44

where $d\mathbf{W}_t$ is a Wiener noise (also called Brownian noise) process. In discrete time, we can use the following Euler approximation:

$$\theta_{t+1} \approx \theta_t - \eta_t \nabla \mathcal{E}(\theta_t) + \sqrt{2\eta_t} \epsilon_t \quad (12.146)$$

where $\epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and η_t is a step size.

Computing the gradient at each step takes $O(N)$ time. We can solve this by computing an unbiased minibatch approximation to the gradient term

$$\mathbf{g}(\theta_t, \xi_t) = \frac{1}{B} \sum_{n \in \xi_t} \nabla \mathcal{E}_n(\theta_t) = -\nabla \log p_0(\theta_t) - \frac{N}{B} \sum_{n \in \xi_t} \nabla \log p(\mathbf{x}_n | \theta_t) \quad (12.147)$$

This gives rise to the following update:

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{g}(\theta_t, \xi_t) + \sqrt{2\eta_t} \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (12.148)$$

This is called **stochastic gradient Langevin dynamics** or **SGLD** [Wel11]. The resulting update step is identical to SGD, except for the addition of a Gaussian noise term. (See [Neg+21] for some recent analysis of this method; they also suggest setting $\eta_t \propto N^{-2/3}$.)

12.7.2 Preconditionining

As in SGD, we can get better results (especially for models such as neural networks) if we use preconditioning to scale the gradient updates. In [PT13], they use the Fisher information matrix (FIM) as the preconditioner; this method is known as **Stochastic Gradient Riemannian Langevin Dynamics** or **SGRLD**.

Unfortunately, computing the FIM is often hard to compute. In [Li+16], they propose to use the same kind of diagonal approximation as used by RMSprop; this is called **preconditioned SGLD**, and has the following form:

$$\mathbf{v}_t = \alpha \mathbf{v}_{t-1} + (1 - \alpha) \mathbf{g}_t \odot \mathbf{g}_t \quad (12.149)$$

$$\mathbf{G}_t = \text{diag}(1/(\sqrt{\mathbf{v}_t} + \epsilon)) \quad (12.150)$$

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{G}_t \mathbf{g}_t + \sqrt{2\eta_t \mathbf{G}_t} \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (12.151)$$

The term $0 < \alpha < 1$ controls the memory of the process; in the paper, they set $\alpha = 0.99$. The term ϵ ensures the matrix is positive definite; they use $\epsilon = 10^{-5}$. See Section 20.4.3.1 for an example.

An alternative to an RMSprop-like preconditioner is to use an Adam-like preconditioner, as proposed in [KSL21]. This is called **SGLD-Adam**, and has the following form:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \quad (12.152)$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t \quad (12.153)$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \quad (12.154)$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \quad (12.155)$$

$$\mathbf{G}_t = \text{diag}(1/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)) \quad (12.156)$$

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{G}_t \hat{\mathbf{m}}_t + \sqrt{2\eta_t \mathbf{G}_t} \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (12.157)$$

In [CSN21] they set $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

1 **12.7.3 Reducing the variance of the gradient estimate**

3 The variance of the noise introduced by minibatching can be quite large, which can hurt the
4 performance of methods such as SGLD [BDM18]. In [Bak+17], they propose to reduce the variance
5 of this estimate by using a **control variate** estimator; this method is therefore called **SGLD-CV**.
6 Specifically they use the following gradient approximation:

8 $\hat{\nabla}_{cv}\mathcal{E}(\boldsymbol{\theta}_t) = \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}}))$ (12.158)

11 Here $\hat{\boldsymbol{\theta}}$ is any fixed value, but it is often taken to be a MAP estimate. The reason Equation (12.158)
12 is valid is because the terms we add and subtract are equal in expectation, and hence we get an
13 unbiased estimate:

15 $\mathbb{E} [\hat{\nabla}_{cv}\mathcal{E}(\boldsymbol{\theta}_t)] = \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \mathbb{E} \left[\frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}})) \right]$ (12.159)

18 $= \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \nabla\mathcal{E}(\boldsymbol{\theta}_t) - \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) = \nabla\mathcal{E}(\boldsymbol{\theta}_t)$ (12.160)

20 Note that the first term, $\nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) = \sum_{n=1}^N \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}})$, requires a single pass over the entire dataset, but
21 only has to be computed once (e.g., while estimating $\hat{\boldsymbol{\theta}}$).

23 One disadvantage of SGLD-CV is that the reference point $\hat{\boldsymbol{\theta}}$ has to be precomputed, and is then
24 fixed. An alternative is to update the reference point online, by performing periodic full batch
25 estimates. This is called **SVRG-LD** [Dub+16; Cha+18], where SVRG stands for stochastic variance
26 reduced gradient, and LD stands for Langevin Dynamics. If we use $\tilde{\boldsymbol{\theta}}_t$ to denote the most recent
27 snapshot (reference point), the corresponding gradient estimate is given by

28 $\hat{\nabla}_{svrg}\mathcal{E}(\boldsymbol{\theta}_t) = \nabla\mathcal{E}(\tilde{\boldsymbol{\theta}}_t) + \frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\tilde{\boldsymbol{\theta}}_t))$ (12.161)

31 We recompute the snapshot every τ steps (known as the epoch length). See Algorithm 16 for the
32 pseudo-code.

34

35 **Algorithm 16:** SVRG Langevin Descent

37 1 Initialize $\boldsymbol{\theta}_0$;
38 2 **for** $t = 1 : T$ **do**
39 3 **if** $t \bmod \tau = 0$ **then**
40 4 $\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}_t$;
41 5 $\tilde{\mathbf{g}} = \sum_{n=1}^N \mathcal{E}_n(\tilde{\boldsymbol{\theta}})$;
42 6 Sample minibatch $\mathcal{S}_t \in \{1, \dots, N\}$;
43 7 $\mathbf{g}_t = \tilde{\mathbf{g}} + \frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\tilde{\boldsymbol{\theta}}))$;
44 8 $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{g}_t + \sqrt{2\eta_t} \mathcal{N}(\mathbf{0}, \mathbf{I})$;

46

47

The disadvantage of SVRG is that it needs to perform a full pass over the data every τ steps. An alternative approach, called **SAGA-LD** [Dub+16; Cha+18] (which stands for stochastic averaged gradient acceleration), avoids this by only performing a single full pass over the data at the start of the algorithm. However, the SAGA method needs to store N gradient vectors, one per data point, which is prohibitive in memory cost except for certain linear models.

12.7.4 SG-HMC

We discussed Hamiltonian Monte Carlo (HMC) in Section 12.5, which uses auxiliary momentum variables to improve performance over Langevin MC. In this section, we discuss a way to speed it up by approximating the gradients using minibatches. This is called called **SG-HMC** [CFG14a; ZG21], where SG stands for “stochastic gradient”.

Recall that the leapfrog updates have the following form:

$$\mathbf{v}_{t+1/2} = \mathbf{v}_t - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_t) \quad (12.162)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \mathbf{v}_{t+1/2} \quad (12.163)$$

$$\mathbf{v}_{t+1} = \mathbf{v}_{t+1/2} - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_{t+1}) \quad (12.164)$$

We can substitute $\mathbf{v}_{t+1/2}$ into the two following equations, and replace the full batch gradient with a stochastic approximation, to get

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \mathbf{v}_t - \frac{\eta^2}{2} \mathbf{g}(\boldsymbol{\theta}_t, \boldsymbol{\xi}_t) \quad (12.165)$$

$$\mathbf{v}_{t+1} = \mathbf{v}_t - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_t, \boldsymbol{\xi}_t) - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_{t+1}, \boldsymbol{\xi}_{t+1/2}) \quad (12.166)$$

where $\boldsymbol{\xi}_t$ and $\boldsymbol{\xi}_{t+1/2}$ are independent sources of randomness. In the minibatch setting the stochastic gradients are given by

$$\mathbf{g}(\boldsymbol{\theta}_t, \boldsymbol{\xi}_t) = \frac{1}{B} \sum_{n \in \boldsymbol{\xi}_t} \nabla \mathcal{E}_n(\boldsymbol{\theta}_t) = -\nabla \log p_0(\boldsymbol{\theta}_t) - \frac{N}{B} \sum_{n \in \boldsymbol{\xi}_t} \nabla \log p(\mathbf{x}_n | \boldsymbol{\theta}_t) \quad (12.167)$$

In [ZG21], they show that this algorithm (even without the MH rejection step) provides a good approximation to the posterior (in the sense of having small Wasserstein-2 distance) for the case where the energy function is strongly convex. Furthermore, performance can be considerably improved if we use the variance reduction methods discussed in Section 12.7.3.

12.7.5 Underdamped Langevin Dynamics

The **underdamped Langevin dynamics (ULD)** has the form of the following SDE [CDC15; LMS16; Che+18a; Che+18d]:

$$d\boldsymbol{\theta}_t = \mathbf{v}_t dt \quad (12.168)$$

$$d\mathbf{v}_t = -\mathbf{g}(\boldsymbol{\theta}_t) dt - \gamma \mathbf{v}_t dt + \sqrt{2\gamma} d\mathbf{W}_t$$

where $\mathbf{g}(\boldsymbol{\theta}_t) = \nabla \mathcal{E}(\boldsymbol{\theta}_t)$ is the gradient or **force** acting on the particle, $\gamma > 0$ is the **friction** parameter, and $d\mathbf{W}_t$ is a Wiener noise (also called Brownian noise) process.

Equation (12.168) is a version of the Langevin dynamics of Equation (12.90) with a momentum term \mathbf{v}_t . We can solve the dynamics using various integration methods, as we discuss below. It can be shown (see e.g., [LMS16]) that these methods are accurate to second order, whereas solving standard (overdamped) Langevin is only accurate to first order, and thus will require more sampling steps to achieve a given accuracy.

A common approach to discretizing the underdamped Langevin equations is to use a **splitting** approach, in which we represent the update to the $(\boldsymbol{\theta}, \mathbf{v})$ variables in terms of three pieces, denoted A, B and O:

$$d \begin{pmatrix} \boldsymbol{\theta} \\ \mathbf{v} \end{pmatrix} = \underbrace{\begin{pmatrix} \boldsymbol{\theta} dt \\ \mathbf{0} \end{pmatrix}}_A + \underbrace{\begin{pmatrix} \mathbf{0} \\ -\mathbf{g}(\boldsymbol{\theta}) dt \end{pmatrix}}_B + \underbrace{\begin{pmatrix} \mathbf{0} \\ -\gamma \mathbf{v} dt + \sqrt{2\gamma} d\mathbf{W} \end{pmatrix}}_O \quad (12.169)$$

The A term is also called the **drift** term, the B term is also called the **kick** term, and the O term is called the **fluctuation** term, and corresponds to an Ornstein-Uhlenbeck process. Given this decomposition, we can define the following operators:

$$A_\eta(\boldsymbol{\theta}, \mathbf{v}) = (\boldsymbol{\theta} + \eta \mathbf{v}, \mathbf{v}) \quad (12.170)$$

$$B_\eta(\boldsymbol{\theta}, \mathbf{v}) = (\boldsymbol{\theta}, \mathbf{v} - \eta \mathbf{g}(\boldsymbol{\theta})) \quad (12.171)$$

$$O_{\mathbf{M}, \mathbf{r}}(\boldsymbol{\theta}, \mathbf{v}) = (\boldsymbol{\theta}, \boldsymbol{\Gamma} \mathbf{v} + \sqrt{\mathbf{I} - \boldsymbol{\Gamma}^2} \mathbf{r}) \quad (12.172)$$

where $\mathbf{r} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is a noise term, and $\boldsymbol{\Gamma} \in \mathbb{R}^{D \times D}$ is a symmetric matrix, often taken to be $\boldsymbol{\Gamma} = e^{-\eta\gamma} \mathbf{I}$, where η is the step size and γ is the friction term.

Using this notation, we can define the **BAOAB** method to be the combined set of operators

$$B_\eta \circ A_{\eta/2} \circ O_{e^{-\gamma\eta} \mathbf{I}, \mathbf{r}_t} \circ A_{\eta/2} \circ B_{\eta/2} \quad (12.173)$$

This corresponds to these updates:

$$\mathbf{v}_{t+1/2} = \mathbf{v}_t - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_t) \quad (12.174)$$

$$\boldsymbol{\theta}_{t+1/2} = \boldsymbol{\theta}_t + \frac{\eta}{2} \mathbf{v}_{t+1/2} \quad (12.175)$$

$$\tilde{\mathbf{v}}_{t+1/2} = \alpha \mathbf{v}_{t+1/2} + \sqrt{(1 - \alpha^2)} \mathbf{r}_t \quad (12.176)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_{t+1/2} + \frac{\eta}{2} \tilde{\mathbf{v}}_{t+1/2} \quad (12.177)$$

$$\mathbf{v}_{t+1} = \tilde{\mathbf{v}}_{t+1/2} - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_{t+1}) \quad (12.178)$$

where $\alpha = e^{-\gamma\eta}$. Another popular scheme is the **ABOBA** scheme. It can be shown [LMS16] that any symmetric (or **palindromic**) string of these three operators will result in a second order approximation, with error at most $O(\eta^2)$. In [MW18] they propose an approximate version of the ABOBA scheme, in which stochastic gradients are used in the B steps.

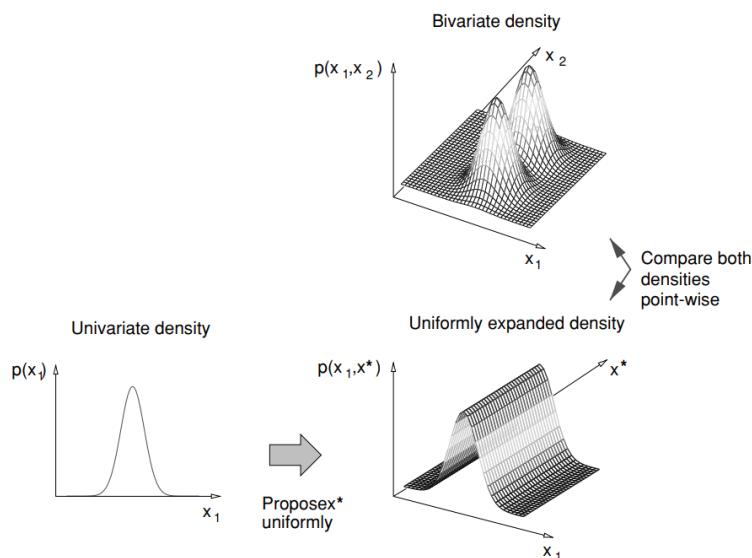


Figure 12.24: To compare a 1d model against a 2d model, we first have to map the 1d model to 2d space so the two have a common measure. Note that we assume the ridge has finite support, so it is integrable. From Figure 17 of [And+03]. Used with kind permission of Nando de Freitas.

12.8 Reversible jump (trans-dimensional) MCMC

Suppose we have a set of models with different numbers of parameters, e.g., mixture models in which the number of mixture components is unknown. Let the model be denoted by m , and let its unknowns (e.g., parameters) be denoted by $\boldsymbol{x}_m \in \mathcal{X}_m$ (e.g., $\mathcal{X}_m = \mathbb{R}^{n_m}$, where n_m is the dimensionality of model m). Sampling in spaces of differing dimensionality is called **trans-dimensional MCMC**. We could sample the model indicator $m \in \{1, \dots, M\}$ and sample all the parameters from the product space $\prod_{m=1}^M \mathcal{X}_m$, but this is very inefficient. It is more parsimonious to sample in the union space $\mathcal{X} = \bigcup_{m=1}^M \{m\} \times \mathcal{X}_m$, where we only worry about parameters for the currently active model.

The difficulty with this approach arises when we move between models of different dimensionality. The trouble is that when we compute the MH acceptance ratio, we are comparing densities defined in different dimensionality spaces, which is meaningless. For example, comparing densities on two points of a sphere makes sense, but comparing a density on a sphere to a density on a circle does not, as there is a dimensional mismatch in the two concepts. The solution, proposed by [Gre98] and known as **reversible jump MCMC** or **RJMCMC**, is to augment the low dimensional space with extra random variables so that the two spaces have a common measure. This is illustrated in Figure 12.24.

We give a sketch of the algorithm below. For more details, see e.g., [Gre03; HG12].

1 **12.8.1 Basic idea**

3 To explain the method in more detail, we follow the presentation of [And+03]. To ensure a common
4 measure, we need to define a way to extend each pair of subspaces \mathcal{X}_m and \mathcal{X}_n to $\mathcal{X}_{m,n} = \mathcal{X}_m \times \mathcal{U}_{m,n}$
5 and $\mathcal{X}_{n,m} = \mathcal{X}_n \times \mathcal{U}_{n,m}$. We also need to define a deterministic, differentiable and invertible mapping
6

7 $(\mathbf{x}_m, \mathbf{u}_{m,n}) = f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m}) = (f_{n \rightarrow m}^x(\mathbf{x}_n, \mathbf{u}_{n,m}), f_{n \rightarrow m}^u(\mathbf{x}_n, \mathbf{u}_{n,m}))$ (12.179)

9 Invertibility means that

11 $f_{m \rightarrow n}(f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m})) = (\mathbf{x}_n, \mathbf{u}_{n,m})$ (12.180)

14 Finally, we need to define proposals $q_{n \rightarrow m}(\mathbf{u}_{n,m}|n, \mathbf{x}_n)$ and $q_{m \rightarrow n}(\mathbf{u}_{m,n}|m, \mathbf{x}_m)$. We can sample these
15 “noise” terms, and then “feed them” into the deterministic mappings in order to generate samples
16 from a different-sized space. We will give an example of this below.

17 Now suppose we are in state (n, \mathbf{x}_n) . We move to (m, \mathbf{x}_m) by generating $\mathbf{u}_{n,m} \sim q_{n \rightarrow m}(\cdot|n, \mathbf{x}_n)$,
18 ensuring that we have reversibility $(\mathbf{x}_m, \mathbf{u}_{m,n}) = f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m})$. We then accept the move with
19 probability

21 $A_{n \rightarrow m} = \min \left\{ 1, \frac{p(m, \mathbf{x}_m^*)}{p(n, \mathbf{x}_n)} \times \frac{q(n|m)}{q(m|n)} \times \frac{q_{m \rightarrow n}(\mathbf{u}_{m,n}|m, \mathbf{x}_m^*)}{q_{n \rightarrow m}(\mathbf{u}_{n,m}|n, \mathbf{x}_n)} \times |\det \mathbf{J}_{f_{m \rightarrow n}}| \right\}$ (12.181)

24 where $\mathbf{x}_m^* = f_{n \rightarrow m}^x(\mathbf{x}_n, \mathbf{u}_{n,m})$, $\mathbf{J}_{f_{m \rightarrow n}}$ is the Jacobian of the transformation

27 $J_{f_{m \rightarrow n}} = \frac{\partial f_{n \rightarrow m}(\mathbf{x}_m, \mathbf{u}_{m,n})}{\partial (\mathbf{x}_m, \mathbf{u}_{m,n})}$ (12.182)

30 and $|\det \mathbf{J}|$ is the absolute value of the determinant of the Jacobian.

32 **Algorithm 17:** Generic reversible jump MCMC (single step)

- 34 1 Sample $u \sim U(0, 1)$;
35 2 If $u \leq b_k$;
36 3 then birth move;
37 4 else if $u \leq (b_k + d_k)$ then death move;
38 5 else if $u \leq (b_k + d_k + s_k)$ then split move;
39 6 else if $u \leq (b_k + d_k + s_k + m_k)$ then merge move;
40 7 else update parameters;
-

43 Some pseudo-code for the algorithm is given in Algorithm 17, where b_k is the probability of a birth
44 move if the current model is k , d_k is the probability of a death move, s_k is the probability of a split
45 move, and m_k is the probability of a merge move. If we don’t make a dimension-changing move, we
46 can just update the parameters in the usual way.

47

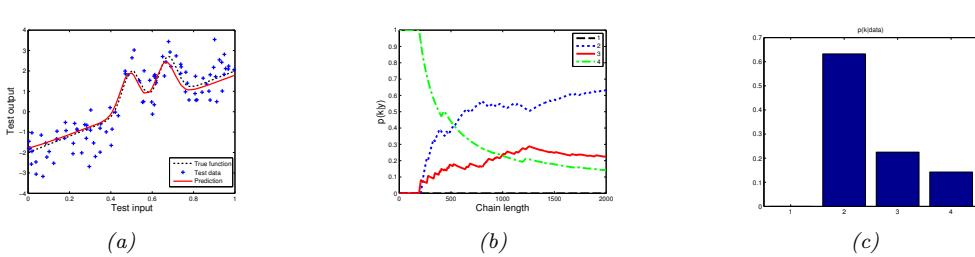


Figure 12.25: Fitting an RBF network to some 1d data using RJMCMC. (a) Prediction on test set. (b) Plot of $p(k|\mathcal{D})$ vs iteration. (c) Final posterior $p(k|\mathcal{D})$. Adapted from Figure 4 of [AFD01].

12.8.2 Example

Let us consider an example from [AFD01]. They consider an RBF network for nonlinear regression of the form

$$f(\mathbf{x}) = \sum_{j=1}^k a_j \mathcal{K}(\|\mathbf{x} - \boldsymbol{\mu}_j\|) + \boldsymbol{\beta}^\top \mathbf{x} + \beta_0 + \epsilon \quad (12.183)$$

where $\mathcal{K}()$ is some kernel function (e.g., a Gaussian), k is the number of such basis functions, and ϵ is a Gaussian noise term. If $k = 0$, the model corresponds to linear regression.

They fit this model to some training data. The prediction on test data is shown in Figure 12.25(a), and does not exhibit any overfitting, despite the high degree of noise. Estimates of $p(k|\mathcal{D})$, the distribution over the number of basis functions, are shown in Figure 12.25(b) as a function of the iteration number; the posterior at the final iteration is shown in Figure 12.25(c). There is clearly the most posterior support for $k = 2$, which makes sense given the two ‘‘bumps’’ in the data.

To generate these results, we consider several kinds of proposal. One of them is to split a current basis function μ into two new ones using

$$\mu_1 = \mu - u_{n,n+1}\alpha, \quad \mu_2 = \mu + u_{n,n+1}\alpha \quad (12.184)$$

where α is a parameter of the proposal, and $u_{n,m}$ is sampled from some distribution (e.g., uniform). To ensure reversibility, they define a corresponding merge move

$$\mu = \frac{\mu_1 + \mu_2}{2} \quad (12.185)$$

where μ_1 is chosen at random, and μ_2 is its nearest neighbor. To ensure these moves are reversible, we require $\|\mu_1 - \mu_2\| < 2\beta$.

The acceptance ratio for the split move is given by

$$A_{\text{split}} = \min \left\{ 1, \frac{p(k+1, \mu_{k+1})}{p(k, \mu_{k+1})} \times \frac{1/(k+1)}{1/k} \times \frac{1}{p(u_{n,m})} \times |\det \mathbf{J}_{\text{split}}| \right\} \quad (12.186)$$

where $1/k$ is the probability of choosing one of the k bases uniformly at random. The Jacobian is

$$\mathbf{J}_{\text{split}} = \frac{\partial(\mu_1, \mu_2)}{\partial(\mu, u_{n,m})} = \det \begin{pmatrix} 1 & 1 \\ -\beta & \beta \end{pmatrix} \quad (12.187)$$

1 so $|\det \mathbf{J}_{split}| = 2\beta$. The acceptance ratio for the merge move is given by
2

3
4
$$A_{merge} = \min \left\{ 1, \frac{p(k-1, \mu_{k-1})}{p(k, \mu_k)} \times \frac{1/(k-1)}{1/k} \times |\det \mathbf{J}_{merge}| \right\} \quad (12.188)$$

5

6 where $|\det \mathbf{J}_{merge}| = 1/(2\beta)$.
7

8 **12.8.3 Discussion**
9

10 RJMCMC algorithms can be quite tricky to implement. If, however, the continuous parameters can
11 be integrated out (resulting in a method called collapsed RJMCMC), much of the difficulty goes
12 away, since we are just left with a discrete state space, where there is no need to worry about change
13 of measure. For example, if we fix the centers μ_j in Equation (12.183) (e.g., using samples from the
14 data, or using K-means clustering), we are left with a linear model, where we can integrate out the
15 parameters. All that is left to do is sample which of these fixed basis functions to include in the
16 model, which is a discrete variable selection problem. See e.g., [Den+02] for details.
17

18 In Chapter 33, we discuss **Bayesian nonparametric models**, which allow for an infinite number
19 of different models. Surprisingly, such models are often easier to deal with computationally (as well
20 as more realistic, statistically) than working with a finite set of different models.

21 **12.9 Annealing methods**
22

23 Many distributions are multimodal and hence hard to sample from. However, by analogy to the way
24 metals are heated up and then cooled down in order to make the molecules align, we can imagine
25 using a computational temperature parameter to smooth out a distribution, gradually cooling it to
26 recover the original “bumpy” distribution. We first explain this idea in more detail in the context of
27 an algorithm for MAP estimation. We then discuss extensions to the sampling case.
28

29 **12.9.1 Parallel tempering**
30

31 Another way to combine MCMC and annealing is to run multiple chains in parallel at different
32 temperatures, and allow one chain to sample from another chain at a neighboring temperature. In
33 this way, the high temperature chain can make long distance moves through the state space, and have
34 this influence lower temperature chains. This is known as **parallel tempering**. See e.g., [ED05;
35 Kat+06] for details.
36

37
38
39
40
41
42
43
44
45
46
47

13 Sequential Monte Carlo inference

13.1 Introduction

In this chapter, we discuss **sequential Monte Carlo** or **SMC** algorithms, which can be used to sample from a sequence of related probability distributions. SMC is most commonly used to solve filtering in state-space models (SSM, Chapter 31), but it can also be applied to other problems, such as sampling from a static (but possibly multi-modal) distribution, or for sampling rare events from some process. Our presentation is based on the excellent tutorial [NLS19], and differs from traditional presentations, such as [Aru+02], by emphasizing the fact that we are sampling sequences of related variables, not just computing the filtering distribution of an SSM; this more general perspective will let us tackle static estimation problems, as we will see. (This more general perspective is also used in the tutorial in [DJ11].) For a more formal (measure theoretic) treatment of SMC, using the **Feynman-Kac** formalism, see [CP20b].

13.1.1 Problem statement

In SMC, the goal is to sample from a sequence of related distributions of the form

$$\pi_t(\mathbf{z}_{1:t}) = \frac{1}{Z_t} \tilde{\gamma}_t(\mathbf{z}_{1:t}) \quad (13.1)$$

for $t = 1 : T$, where $\tilde{\gamma}_t$ is the unnormalized target distribution, π_t is the normalized version, and $\mathbf{z}_{1:t}$ are the random variables of interest. In some applications (e.g., filtering in an SSM), we care about each intermediate marginal distribution, $\pi_t(\mathbf{z}_t)$, for $t = 1 : T$; this is called **particle filtering**. (The word “particle” just means “sample”.) In other applications, we only care about the final distribution, $\pi_T(\mathbf{z}_T)$, and the intermediate steps are introduced just for computational reasons; this is called an **SMC sampler**. We briefly review both of these below, and go into more detail in later sections.

13.1.2 Particle filtering for state-space models

An important application of SMC is to sequential (online) inference (state estimation) in SSMs. As an example, consider a Markovian state-space model with the following joint distribution:

$$\pi_T(\mathbf{z}_{1:T}) \propto p(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{t=1}^T p(\mathbf{z}_t|\mathbf{z}_{t-1})p(\mathbf{y}_t|\mathbf{z}_t) \quad (13.2)$$

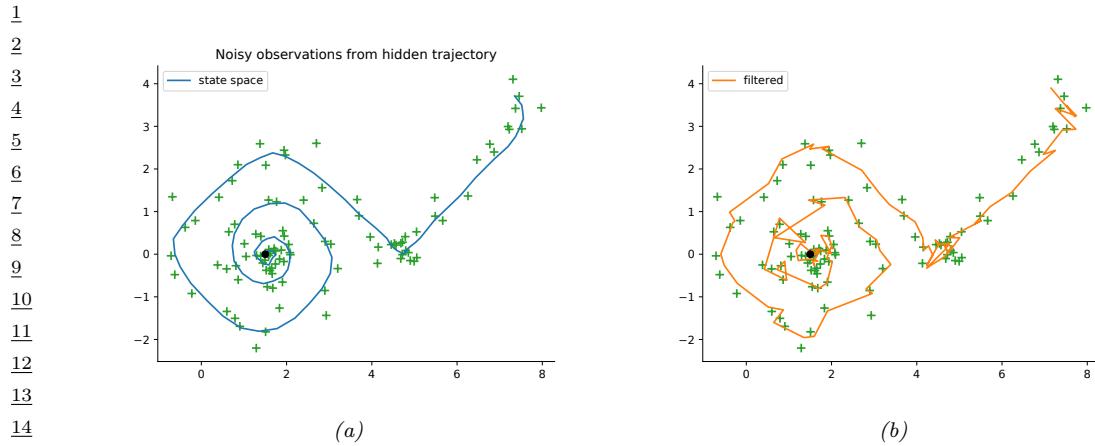


Figure 13.1: Illustration of particle filtering (using the dynamical prior as the proposal) applied to a 2d nonlinear dynamical system. (a) True underlying state and observed data. (b) PF estimate of the posterior mean. Generated by `bootstrap_filter.py`.

A common choice is to define the unnormalized target distribution at step t to be

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{s=1}^t p(\mathbf{z}_s|\mathbf{z}_{s-1})p(\mathbf{y}_s|\mathbf{z}_s) \quad (13.3)$$

Note that this is a distribution over an (ever growing) sequence of variables. However, we often only care about the most recent marginal of this distribution, in which case we just need to compute $\tilde{\gamma}_t(\mathbf{z}_t) = p(\mathbf{z}_t, \mathbf{y}_{1:t})$, which avoids having to store the full history.

For example, consider the following 2d nonlinear tracking problem:

$$\begin{aligned} p(\mathbf{z}_t|\mathbf{z}_{t-1}) &= \mathcal{N}(\mathbf{z}_t|f(\mathbf{z}_{t-1}), q\mathbf{I}) \\ p(\mathbf{y}_t|\mathbf{z}_t) &= \mathcal{N}(\mathbf{y}_t|\mathbf{z}_t, r\mathbf{I}) \\ f(\mathbf{z}) &= (z_1 + \Delta \sin(z_2), z_2 + \Delta \cos(z_1)) \end{aligned} \quad (13.4)$$

where Δ is the step size of the underlying continuous system, q is the variance of the system noise, and r is the variance of the observation noise. (We treat Δ , q and r as fixed constants; see Section 13.3.4 for a discussion of joint state and parameter estimation.) The true underlying state trajectory, and the corresponding noisy measurements, are shown in Figure 13.1a. The posterior mean estimate of the state, computed using 2000 samples in a simple form of SMC called the bootstrap filter (Section 13.2.3.1), is shown in Figure 13.1b.

Particle filtering can also be applied to **non-Markovian models**, where \mathbf{z}_t may depend on all the past hidden states, $\mathbf{z}_{1:t-1}$, and \mathbf{y}_t depends on the current \mathbf{z}_t and possibly also all the past hidden states, $\mathbf{z}_{1:t-1}$. In this case, the target distribution at step t is

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{s=1}^t p(\mathbf{z}_s|\mathbf{z}_{1:s-1})p(\mathbf{y}_s|\mathbf{z}_{1:s}) \quad (13.5)$$

For example, consider a 1d Gaussian sequence model where the dynamics are first-order Markov, but the observations depend on the entire past sequence:

$$\begin{aligned} p(z_t | z_{1:t-1}) &= \mathcal{N}(z_t | \phi z_{t-1}, q^2) \\ p(y_t | z_{1:t}) &= \mathcal{N}(y_t | \sum_{s=1}^t \beta^{t-s} z_s, r^2) \end{aligned} \quad (13.6)$$

If we set $\beta = 0$, we obtain a linear-Gaussian SSM. As β gets larger, the dependence on the past increases, making the inference problem harder. (We will revisit this example below.)

13.1.3 SMC samplers for static parameter estimation

Now consider the problem of parameter estimation from a fixed dataset, $\mathcal{D} = \{\mathbf{y}_n : n = 1 : N\}$. We suppose the observations are conditionally iid, so the posterior has the form $p(\mathbf{z}|\mathcal{D}) \propto p(\mathbf{z}) \prod_{n=1}^N p(\mathbf{y}_n|\mathbf{z})$, where \mathbf{z} is the unknown parameter. It is not immediately obvious how to approximate $p(\mathbf{z}|\mathcal{D})$ using SMC, since we just have one distribution. However, we can convert this into a sequential inference problem in several different ways. One approach, known as **data tempering**, defines the (marginal) target distribution at step t as $\tilde{\gamma}_t(\mathbf{z}_t) = p(\mathbf{z}_t)p(\mathbf{y}_{1:t}|\mathbf{z}_t)$. In this case, the number of time steps T is the same as the number of data samples, N . Another approach, known as **likelihood tempering**, defines the (marginal) target distribution at step t as $\tilde{\gamma}_t(\mathbf{z}_t) = p(\mathbf{z}_t)p(\mathcal{D}|\mathbf{z}_t)^{\tau_t}$, where $0 = \tau_t < \dots < \tau_T = 1$ is a temperature parameter. In this case, the number of steps T depends on how quickly we anneal the distribution from the initial prior $p(\mathbf{z}_1)$ to the final target $p(\mathbf{z}_T)p(\mathcal{D}|\mathbf{z}_T)$.

Once we have defined the marginal target distributions $\tilde{\gamma}_t(\mathbf{z}_t)$, we need a way to expand this to a joint target distribution over a *sequence* of variables, $\tilde{\gamma}_t(\mathbf{z}_{1:t})$, so the distributions become connected to each other. We explain how to do this in Section 13.6. We can then apply particle filtering to the problem in the usual way. At the end, we extract the final joint target distribution, $\tilde{\gamma}_T(\mathbf{z}_{1:T}) = p(\mathbf{z}_{1:T})p(\mathcal{D}|\mathbf{z}_T)$, from which we can compute the marginal target distribution $\tilde{\gamma}_T(\mathbf{z}_T) = p(\mathbf{z}_T, \mathcal{D})$, from which we can get the posterior $p(\mathbf{z}|\mathcal{D})$ by normalizing. We give the details in Section 13.6.

13.2 Basics of SMC

In this section, we cover the basics of SMC.

13.2.1 Importance sampling

Suppose we are interested in estimating the expectation of some function φ_t with respect to a target distribution π_t to compute

$$\pi_t(\varphi) \triangleq \mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] \quad (13.7)$$

One approach is to use self-normalized importance sampling (Section 11.5) with proposal $q_t(\mathbf{z}_{1:t})$. We then get the approximation

$$\mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] \approx \sum_{i=1}^{N_s} W_t^i \varphi_t(\mathbf{z}_{1:t}^i), \quad \mathbf{z}_{1:t}^i \stackrel{\text{iid}}{\sim} q_t \quad (13.8)$$

1 where the **normalized weights** are
2

$$\frac{3}{4} \quad W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j} \quad (13.9)$$

6 and the **unnormalized weights** are
7

$$\frac{8}{9} \quad \tilde{w}_t^i = \frac{\pi_t(\mathbf{z}_{1:t})}{q_t(\mathbf{z}_{1:t}^i)} \quad (13.10)$$

10 Alternatively, instead of computing the expectation of a specific target function, we can just
11 approximate the target distribution itself, using a sum of weighted samples:
12

$$\frac{13}{14} \quad \pi_t(\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i) \triangleq \hat{\pi}_t(\mathbf{z}_{1:t}) \quad (13.11)$$

16 Furthermore, from Equation (11.43), we know that we can approximate the normalization constant
17 using the following unbiased estimator:
18

$$\frac{19}{20} \quad Z_t \approx \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i \triangleq \hat{Z}_t \quad (13.12)$$

22 The problem with importance sampling when applied in the context of sequential models is that
23 the dimensionality of the state space is very large, and increases with t . This makes it very hard to
24 define a good proposal that covers the high probability regions, resulting in most samples getting
25 negligible weight. Indeed it is known that IS suffers from the curse of dimensionality. In the sections
26 below, we discuss better sampling strategies.
27

28 13.2.2 Sequential importance sampling

30 In this section, we discuss **sequential importance sampling** or **SIS**, in which the proposal has
31 the following autoregressive structure:
32

$$\frac{33}{34} \quad q_t(\mathbf{z}_{1:t}) = q_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.13)$$

35 We can obtain samples from $q_{t-1}(\mathbf{z}_{1:t-1})$ by reusing the $\mathbf{z}_{1:t-1}^i$ samples, which we then extend by
36 one step by sampling from the conditional $q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$. We can think of this as “growing” the chain
37 (sequence of states). The unnormalized weights can be computed recursively as follows:

$$\frac{38}{39} \quad \tilde{w}_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{q_t(\mathbf{z}_{1:t})} = \frac{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{q_{t-1}(\mathbf{z}_{1:t-1})} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})} \quad (13.14)$$

$$\frac{41}{42} \quad = \tilde{w}_{t-1}(\mathbf{z}_{1:t-1}) \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})} \quad (13.15)$$

43 The ratio factors are sometimes called the **incremental importance weights**:
44

$$\frac{45}{46} \quad \alpha_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})} \quad (13.16)$$

1
2 See Algorithm 18 for pseudocode for the resulting SIS algorithm. (In practice we compute the weights
3 in log-space, and convert back using the log-sum-exp trick.)

4 Note that, in the special case of state space models, the weight computation can be further
5 simplified. In particular, suppose we have

$$7 \quad \tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) = p(\mathbf{y}_t | \mathbf{z}_{1:t}) p(\mathbf{z}_t | \mathbf{z}_{1:t-1}) p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1}) = p(\mathbf{y}_t | \mathbf{z}_{1:t}) p(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) \\ 8 \quad (13.17) \\ 9$$

10 Then the incremental weight is given by

$$12 \quad \alpha_t(\mathbf{z}_{1:t}) = \frac{p(\mathbf{y}_t | \mathbf{z}_{1:t}) p(\mathbf{z}_t | \mathbf{z}_{1:t-1})}{q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})} \\ 13 \quad (13.18) \\ 14 \\ 15$$

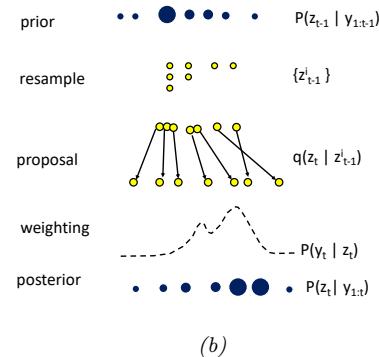
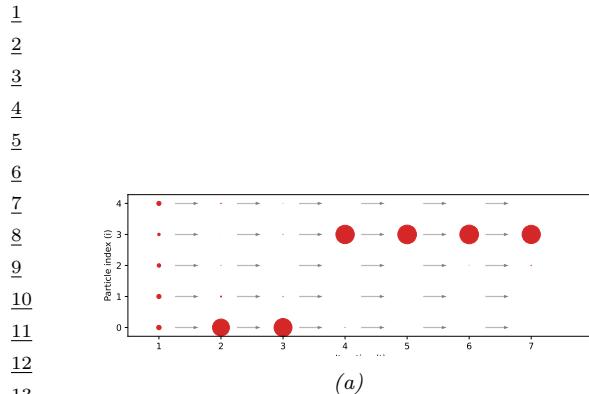
16 **Algorithm 18:** Sequential importance sampling

17 1 Initialization: $\mathbf{z}_1^i \sim q_1(\mathbf{z}_1)$, $\tilde{w}_1^i = \frac{\tilde{\gamma}_1(\mathbf{z}_1^i)}{q_1(\mathbf{z}_1^i)}$, $W_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$, $\hat{\pi}_1(\mathbf{z}_1) = \sum_{i=1}^{N_s} W_1^i \delta(\mathbf{z}_1 - \mathbf{z}_1^i)$
 18 2 **for** $t = 2 : T$ **do**
 19 3 **for** $i = 1 : N_s$ **do**
 20 4 Sample $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$
 21 5 Append $\mathbf{z}_{1:t}^i = (\mathbf{z}_{1:t-1}^i, \mathbf{z}_t^i)$
 22 6 Compute incremental weight $\alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$
 23 7 Compute unnormalized weight $\tilde{w}_t^i = \tilde{w}_{t-1}^i \alpha_t^i$
 24 8 Compute normalized weights $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$ for $i = 1 : N_s$
 25 9 Define $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$

30 Unfortunately SIS suffers from a problem known as **weight degeneracy** or **particle impoverishment**, in which most of the weights go to zero, so the posterior ends up being approximated by a single particle. This is illustrated in Figure 13.2a, where we apply SIS to the non-Markovian example in Equation (13.6) using $N_s = 5$ particles. The reason for degeneracy is that each particle has to “explain” (generate) the entire sequence of observations. Each sequence of guessed states becomes increasingly improbable over time, due to the product of likelihood terms, and the differences between the weights of each hypothesis will grow exponentially. Of course, there has to be a best sequence amongs the set of candidates, so when we normalize the weights, the best one will get weight 1 and the rest will get weight 0. We discuss a solution to this in Section 13.2.3.

41 **13.2.3 Sequential importance sampling with resampling**

43 In this section, we describe **sequential importance sampling with resampling (SISR)**. The
44 basic idea is this: instead of “growing” all of the old particle sequences by one step, we first select the
45 N_s “fittest” particles, by sampling from the old posterior, and then letting these survivors grow by
46 one step.



14 *Figure 13.2:* (a) Illustration of weight degeneracy for SIS applied to the model in Equation (13.6). with
15 parameters $(\phi, q, \beta, r) = (0.9, 10.0, 0.5, 1.0)$. We use $T = 6$ steps ad $N_s = 5$ samples. We see that as t
16 increases, almost all the probability mass concentrates on particle 3. Generated by `sis_vs_smcmc.py`. Adapted
17 from Figure 2 of [NLS19]. (b) Illustration of the bootstrap particle filtering algorithm.

18

19 **Algorithm 19:** Sequential importance sampling with resampling

20 1 Initialization: $z_1^i \sim q_1(z_1)$, $\tilde{w}_1^i = \frac{\tilde{\gamma}_1(z_1^i)}{q_1(z_1^i)}$, $W_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$, $\hat{\pi}_1(z_1) = \sum_{i=1}^{N_s} W_1^i \delta(z_1 - z_1^i)$

21 2 **for** $t = 2 : T$ **do**

22 3 Resample $z_{t-1}^{1:N_s} = \text{resample}(\hat{\pi}_{t-1})$

23 4 **for** $i = 1 : N_s$ **do**

24 5 Move $z_t^i \sim q_t(z_t | z_{1:t-1}^i)$

25 6 Weight $\tilde{w}_t^i = \alpha_t^i = \frac{\tilde{\gamma}_t(z_{1:t}^i)}{\tilde{\gamma}_{t-1}(z_{1:t-1}^i) q_t(z_t^i | z_{1:t-1}^i)}$

26 7 Compute $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$ for $i = 1 : N_s$

27 8 Define $\hat{\pi}_t(z_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(z_{1:t} - z_{1:t}^i)$

32

33

34 In more detail, at step t , we sample from

35 $q_t^{\text{SISR}}(z_{1:t}) = \hat{\pi}_{t-1}(z_{1:t-1}) q_t(z_t | z_{1:t-1}) \quad (13.19)$

36 where $\hat{\pi}_{t-1}(z_{1:t-1})$ is the previous weighted posterior approximation. By contrast, in SIS, we sample
37 from

38 $q_t^{\text{SIS}}(z_{1:t}) = q_{t-1}^{\text{SIS}}(z_{1:t-1}) q_t(z_t | z_{1:t-1}) \quad (13.20)$

39 We can sample from Equation (13.19) in two steps. First we **resample** N_s samples from $\hat{\pi}_{t-1}(z_{1:t-1})$
40 to get a *uniformly weighted* set of new samples $z_{1:t-1}^i$. (See Section 13.2.4 for details on how to do
41 this.) Then we extend each sample using $z_t^i \sim q_t(z_t | z_{1:t-1}^i)$, and concatenate z_t^i to $z_{1:t-1}^i$,

42 After making a proposal, we compute the unnormalized weights. We use the usual expression
43 for target over proposal, except we “pretend” that the proposal is given by $\tilde{\gamma}_{t-1}(z_{1:t-1}^i) q_t(z_t^i | z_{1:t-1}^i)$

44

even though we used $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}^i)q_t(\mathbf{z}_t^i|\mathbf{z}_{1:t-1}^i)$. The intuitive reason why this is valid is because the previous weighted approximation, $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}^i)$, was an unbiased estimate of the previous target distribution, $\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})$. (See e.g., [CP20b] for more theoretical details.) The final expression for the unnormalized weights is given by the incremental weights, since the resampling step sets $\tilde{w}_{t-1}^i = 1$:

$$\tilde{w}_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i)q_t(\mathbf{z}_t^i|\mathbf{z}_{1:t-1}^i)} \quad (13.21)$$

We then normalize these weights and compute the new approximationg to the target posterior $\hat{\pi}_t(\mathbf{z}_{1:t})$. See Algorithm 19 for the pseudocode.

13.2.3.1 Bootstrap filter

We now consider a special case of SISR, in which the model is an SSM, and the proposal distribution is equal to the dynamical prior:

$$q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1}) = p(\mathbf{z}_t|\mathbf{z}_{1:t-1}) \quad (13.22)$$

In this case, the corresponding weight update simplifies to

$$\tilde{w}_t(\mathbf{z}_{1:t}) = \tilde{w}_{t-1}(\mathbf{z}_{1:t-1}) \frac{p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t})}{p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.23)$$

$$= \tilde{w}_{t-1}(\mathbf{z}_{1:t-1}) \frac{p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1})}{p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.24)$$

$$= \tilde{w}_{t-1}(\mathbf{z}_{1:t-1})p(\mathbf{y}_t|\mathbf{z}_{1:t}) \quad (13.25)$$

Thus we just multiply the old weights by their likelihood. (In the first-order Markov case, this simplifies to $\tilde{w}_t(\mathbf{z}_{1:t}) = \tilde{w}_{t-1}(\mathbf{z}_{1:t-1})p(\mathbf{y}_t|\mathbf{z}_t)$.) This special case is called the **bootstrap filter** [Gor93] or the **survival of the fittest** algorithm [KKR95]. (In the computer vision literature, it is called the **condensation** algorithm, which stands for “conditional density propagation” [IB98]. See Figure 13.2b for an illustration of how this algorithm works, and Figure 13.1b for some sample results on real data.

The bootstrap filter is be useful for models where we can sample from the dynamics, but cannot evaluate the transition model pointwise. This occurs in certain implicit dynamical models, such as those defined using differential equatons (see e.g., [IBK06]); such models are often used in epidemiology. However, in general it is much more efficient to use proposals that take the current evidence \mathbf{y}_t into account. We discuss ways to approximate such “locally optimal” proposals in Section 13.4.

13.2.3.2 Estimating the normalizing constant

To compute the normalization constant for the full sequence posterior, $p(\mathbf{z}_{1:T}|\mathbf{y}_{1:T}) = \pi_T(\mathbf{z}_{1:T})/Z_T$, where $Z_T = p(\mathbf{y}_{1:T}) = \prod_{t=1}^T p(\mathbf{y}_t|\mathbf{y}_{1:t-1})$, we can use the MC approximation to the normalization constant Equation (13.12) at each step to get

$$\hat{Z}_T = \prod_{t=1}^T \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i \quad (13.26)$$

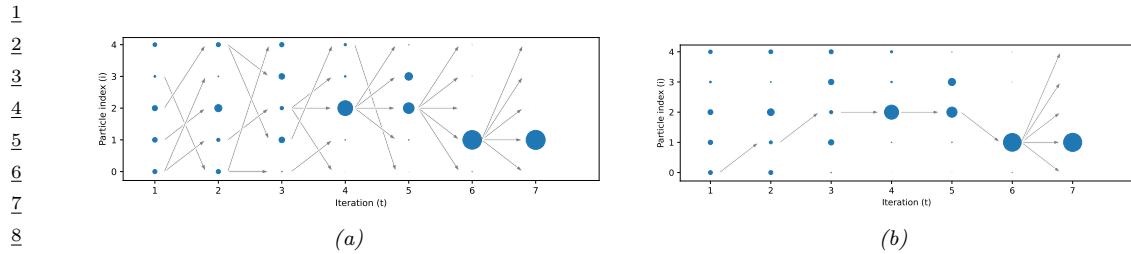


Figure 13.3: (a) Illustration of diversity of samples in SMC applied to the model in Equation (13.6). (b) Illustration of the path degeneracy problem. Generated by `sis_vs_smc.py`. Adapted from Figure 3 of [NLS19].

13.2.3.3 Path degeneracy problem

In Figure 13.3a we show how particle filtering can result in a much more diverse set of active particles, with more balanced weights when applied to the non-Markovian example in Equation (13.6).

While particle filtering does not suffer from weight degeneracy, it does suffer from another problem known as **path degeneracy**. This refers to the fact that the number of particles that survive for many steps may drop rapidly over time, resulting in a loss of diversity when we try to represent the distribution over the past. We illustrate this in Figure 13.3b, where we only include arrows for samples that have been resampled at each step up until the final step. We see that we have $N_s = 5$ identical copies of \mathbf{z}_1^1 in the final set of surviving sequences. (The time at which all the paths meet at a common ancestor, when tracing backwards in time, is known as the **coalescence** time.) We discuss some ways to ameliorate this issue in Section 13.2.4 and Section 13.2.5.

13.2.4 Resampling methods

In this section, we discuss various resampling methods, which can help reduce weight degeneracy and path degeneracy.

13.2.4.1 Multinomial resampling

The simplest approach to resampling is known as **multinomial resampling**. This works as follows. First we form the cumulative distribution from the weights $W_{t-1}^{1:N_s}$, as illustrated by the staircase in Figure 13.4. Then then we sample N_s uniform random variables, $u^i \sim \{0, 1\}$. Finally, we see which interval U^i lands in; if it falls in bin a , we assign the new sample \mathbf{z}_{t-1}^i to be the same as the old \mathbf{z}_{t-1}^a . We say that a is the **ancestor** of i . For precisely, we say a is the ancestor of sample i if

$$\sum_{j=1}^{a-1} W_{t-1}^j \leq u^i < \sum_{j=1}^a W_{t-1}^j \quad (13.27)$$

See ?? 13.1 for some Python code.

Listing 13.1: Multinomial resampling

```
def multinomial_resampling(w, x):
    N = w.shape[0]
    u = np.random.rand(N)
```

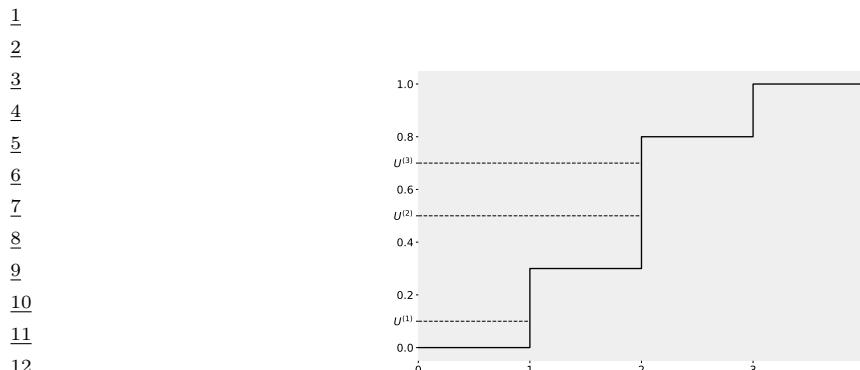


Figure 13.4: Illustration of how to sample from the empirical CDF $P(x) = \sum_{n=1}^N W_n \mathbb{I}(x \geq n)$ shown in black. The height of step n is W_n . If U^m picks step n , then we set the ancestor of m to be n , i.e., $A^m = n$. In this example, $A^{1:3} = (1, 2, 2)$. Adapted from Figure 9.3 of [CP20b].

```

bins = np.cumsum(w)
ancestors = np.digitize(u, bins)
return x[ancestors]

```

Although this is a simple method, it can introduce a lot of variance into the representation of the distribution. For example, suppose all the weights are equal, $W^n = 1/N$. Let $\mathcal{W}^n = \sum_{m=1}^N \mathbb{I}(A^m = n)$ be the number of “offspring” for particle n (i.e., the number of times this particle is chosen in the resampling step). We have $\mathcal{W}^n \sim \text{Bin}(N, 1/N)$, so $P(\mathcal{W}^n = 0) = (1 - 1/N)^N \approx e^{-1} \approx 0.37$. So there is a 37% chance that any given particle will disappear even though they all had the same initial weight. In the sections below, we discuss some **low variance resampling** methods, which can help reduce the path degeneracy problem.

13.2.4.2 Stratified resampling

A simple approach to improve on multinomial resampling is to use **stratified resampling**, in which we divide the unit interval into N_s strata, $(0, 1/N_s]$, $(1/N_s, 2/N_s]$, up to $(1 - 1/N_s, 1]$. We then generate $u^i \sim \text{Unif}((i - 1)/N_s, i/N_s)$ and derive the corresponding ancestor indexes using Equation (13.27). See ?? 13.2 for some Python code.

Listing 13.2: Stratified resampling

```

def stratified_resampling(w, x):
    N = w.shape[0]
    u = (np.arange(N) + np.random.rand(N))/N
    bins = np.cumsum(w)
    ancestors = np.digitize(u, bins)
    return x[ancestors]

```

1 **13.2.4.3 Systematic resampling**

3 We can further reduce the variance by forcing all the samples from $\hat{\pi}_{t-1}$ to be deterministically
4 generated from a shared random source, $u^i \sim \text{Unif}(0, 1)$, by computing
5

$$\underline{6} \quad u^i = \frac{i-1}{N_s} + \frac{u}{N_s} \quad (13.28)$$

8 We derive the corresponding ancestor indexes using Equation (13.27). See ?? 13.3 for some Python
9 code. (We see that this only differs from ?? 13.2 in a single line.)
10

11 *Listing 13.3: Systematic resampling*

```
12 def systematic_resampling(w, x):
13     N = w.shape[0]
14     u = (np.arange(N) + np.random.rand()) / N
15     bins = np.cumsum(w)
16     ancestors = np.digitize(u, bins)
17     return x[ancestors]
```

18 **13.2.4.4 Comparison**

19 It can be proved that all of the above methods are unbiased. It can also be proved that stratified
20 resampling is lower variance than multinomial resampling. Empirically it seems that systematic
21 resampling is lower variance than other methods [HSG06], although it can fail to converge depending
22 on the order of the input samples [GCW19]. A more complex resampling scheme, that is guaranteed
23 to converge and which is also low variance, is described in [GCW19].
24

25 **13.2.5 Adaptive resampling**

26 The resampling step can result in loss of diversity, since each ancestor may generate multiple children,
27 and some may generate no children, since the ancestor indices A_t^n are sampled independently; this
28 is called path degeneracy. On the other hand, if we never resample, we end up with SIS, which
29 suffers from weight degeneracy (particles with negligible weight). A compromise is to use **adaptive**
30 **resampling**, in which we resample whenever the **effective sample size** or **ESS** drops below some
31 minimum, such as $N/2$. A common way to define the ESS is as follows:¹
32

$$\underline{34} \quad \text{ESS}(W^{1:N}) = \frac{1}{\sum_{n=1}^N (W^n)^2} = \frac{\left(\sum_{n=1}^N \tilde{w}^n\right)^2}{\sum_{n=1}^N (\tilde{w}^n)^2} \quad (13.29)$$

35 Note that if we have k weights with $w^n = 1$ and $N - k$ weights with $w^n = 0$, then the ESS is k ; thus
36 ESS is between 1 and N .

37 The pseudocode for SISR with adaptive resampling is given in Algorithm 20. We see that the
38 expression for the weights of the particles are different when no resampling occurs. In particular,
39 instead of using the equally weighted samples after resampling, we use the weighted samples from the
40 previous step without resampling. We then need to add the term $\frac{W_{t-1}^i}{1/N_s}$ as an importance sampling
41

42 1. Note that the ESS used in SMC is different than the ESS used in MCMC (Section 12.6.2.3), which takes into
43 account auto-correlation of the samples.

correction, where the factor $1/N_s$ corresponds to the proposal distribution of not resampling [NLS19, p30]. Although these extra factors will cancel out when we compute the normalized weights, W_t^i , we need to keep track of them for the estimate of Z_T in Equation (13.26) to be valid. (An alternative approach is to compute Z_T in a different way, by keeping track of which steps involved resampling, as in [CP20b, p134].)

Algorithm 20: SISR with adaptive resampling

```

1 Initialization:  $\mathbf{z}_1^i \sim q_1(\mathbf{z}_1)$ ,  $\tilde{w}_1^i = \frac{\tilde{\gamma}_1(\mathbf{z}_1^i)}{q_1(\mathbf{z}_1^i)}$ ,  $W_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$ ,  $\hat{\pi}_1(\mathbf{z}_1) = \sum_{i=1}^{N_s} W_1^i \delta(\mathbf{z}_1 - \mathbf{z}_1^i)$ 
2 for  $t = 2 : T$  do
3   if  $ESS(W_{t-1}^{1:N_s}) < ESS_{\min}$  then
4     Resample  $\mathbf{z}_{t-1}^{1:N_s} = \text{resample}(\hat{\pi}_{t-1})$ 
5     for  $i = 1 : N_s$  do
6       Move  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
7       Unnormalized weights  $\tilde{w}_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
8     else
9       for  $i = 1 : N_s$  do
10      Move  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
11      Unnormalized weights  $\tilde{w}_t^i = \frac{W_{t-1}^i}{1/N_s} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
12    Normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
13    Define  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

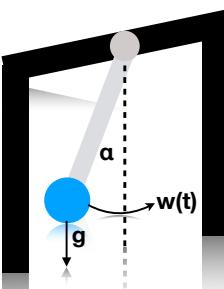
13.3 Some applications of particle filtering

In this section, we give some examples of particle filtering applied to some state estimation problems in different kinds of state-space models. We focus on using the simplest kind of SMC algorithm, namely the bootstrap filter (Section 13.2.3.1).

13.3.1 1d pendulum model with outliers

Consider a simple pendulum of unit mass and length swinging from a fixed attachment, as in Figure 13.5. Such an object is in principle entirely deterministic in its behavior. However, in the real world, there are often unknown forces at work (e.g., air turbulence, friction). We will model these by a continuous time random Gaussian noise process $w(t)$. This gives rise to the following differential equation:

$$\frac{d^2\alpha}{dt^2} = -g \sin(\alpha) + w(t) \quad (13.30)$$



11 *Figure 13.5: Illustration of a pendulum swinging. g is the force of gravity, $w(t)$ is a random external force,*
12 *and α is the angle wrt the vertical. Adapted from Figure 3.10 in [Sar13].*

13

14 We can write this as a nonlinear SSM by defining the state to be $z_1(t) = \alpha(t)$ and $z_2(t) = d\alpha(t)/dt$.
15 Thus

16
$$\frac{dz}{dt} = \begin{pmatrix} z_2 \\ -g \sin(z_1) \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} w(t) \quad (13.31)$$

17 If we discretize this step size Δ , we get the following formulation [Sar13, p74]:

18
$$\underbrace{\begin{pmatrix} z_{1,t} \\ z_{2,t} \end{pmatrix}}_{\mathbf{z}_t} = \underbrace{\begin{pmatrix} z_{1,t-1} + z_{2,t-1}\Delta \\ z_{2,t-1} - g \sin(z_{1,t-1})\Delta \end{pmatrix}}_{f(\mathbf{z}_{t-1})} + \mathbf{q}_{t-1} \quad (13.32)$$

19 where $\mathbf{q}_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ with

20
$$\mathbf{Q} = q^c \begin{pmatrix} \frac{\Delta^3}{3} & \frac{\Delta^2}{2} \\ \frac{\Delta^2}{2} & \Delta \end{pmatrix} \quad (13.33)$$

21 where q^c is the spectral density (continuous time variance) of the continuous-time noise process.

22 If we observe the angular position, we get the linear observation model

23
$$y_t = \alpha_t + r_t = h(\mathbf{z}_t) + r_t \quad (13.34)$$

24 where $h(\mathbf{z}_t) = z_{1,t}$ and r_t is the observation noise. If we only observe the horizontal position, we get
25 the nonlinear observation model

26
$$y_t = \sin(\alpha_t) + r_t = h(\mathbf{z}_t) + r_t \quad (13.35)$$

27 where $h(\mathbf{z}_t) = \sin(z_{1,t})$.

28 If the observation noise is Gaussian, $r_t \sim \mathcal{N}(0, R)$, then the EKF (Section 8.5.2), UKF (Section 8.6.2)
29 and bootstrap filter (Section 13.2.3.1) all give very similar results (not shown). However, suppose
30 that some fraction $p = 0.4$ of the observations are outliers, coming from a $\text{Unif}(-2, 2)$ distribution.
31 (These could represent a faulty sensor, for example.) In this case, the bootstrap filter is more robust,
32 as shown in Figure 13.6, since it can handle multimodal posteriors induced by uncertainty about
33 which observations are signal and which are noise. By contrast, EKF and UKF assume a unimodal
34 (Gaussian) posterior, which is very sensitive to outliers.

35

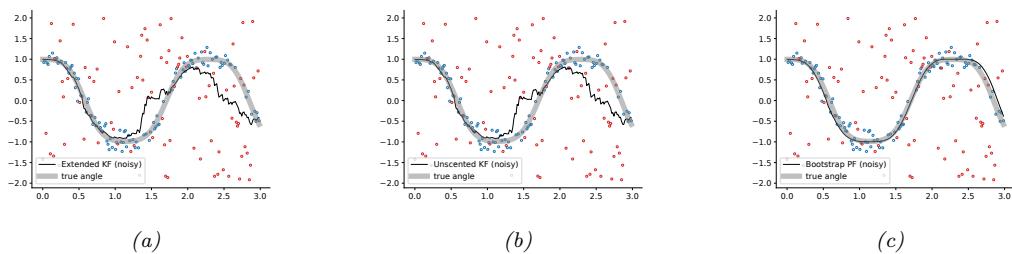


Figure 13.6: Filtering algorithms applied to the noisy pendulum model with 40% outliers (shown in red). (a) EKF. (b) UKF. (c) Bootstrap filter. Generated by [pendulum_1d.py](#).

13.3.2 Visual object tracking

In Section 13.1.2, we tracked an object given noisy measurements of its location, as estimated by some kind of distance sensor. A harder problem is to track an object just given a sequence of frames from a video camera. This is called **visual tracking**. In this section we consider an example where the object is a remote-controlled helicopter [NKMG03]. We will use a simple linear motion model for the centroid of the object, and a color histogram for the likelihood model, using **Bhattacharya distance** to compare histograms.

Figure 13.7 shows some example frames. The system uses $S = 250$ particles, with an effective sample size of $N_{\text{eff}} = 134$. (a) shows the belief state at frame 1. The system has had to resample 5 times to keep the effective sample size above the threshold of 150; (b) shows the belief state at frame 251; the red lines show the estimated location of the center of the object over the last 250 frames. (c) shows that the system can handle **visual clutter** (the hat of the human operator), as long as it does not have the same color as the target object; (d) shows that the system is confused between the grey of the helicopter and the grey of the building (the posterior is bimodal, but the green ellipse, representing the posterior mean and covariance, is in between the two modes); (e) shows that the probability mass has shifted to the wrong mode: i.e., the system has lost track; (f) shows the particles spread out over the gray building; recovery of the object is very unlikely from this state using this proposal.

The simplest way to improve performance of this method is to use more particles. A more efficient approach is to perform **tracking by detection**, by running an object detector over the image every few frames, and to use these as proposals (see Section 13.4). This provides a way to combine discriminative, bottom-up object detection (which can fail in the presence of occlusion) with generative, top-down tracking (which can fail if there are unpredictable motions, or new objects entering the scene). See e.g., [HF09; VG+09; GGO19; OTT19] for further details.

13.3.3 Robot localization

Consider a mobile robot wandering around an indoor environment. We will assume that it already has a map of the world, represented in the form of an **occupancy grid**, which just specifies whether each grid cell is empty space or occupied by something solid like a wall. The goal is for the robot to estimate its location. This can be solved optimally using an HMM filter (also called a **histogram**

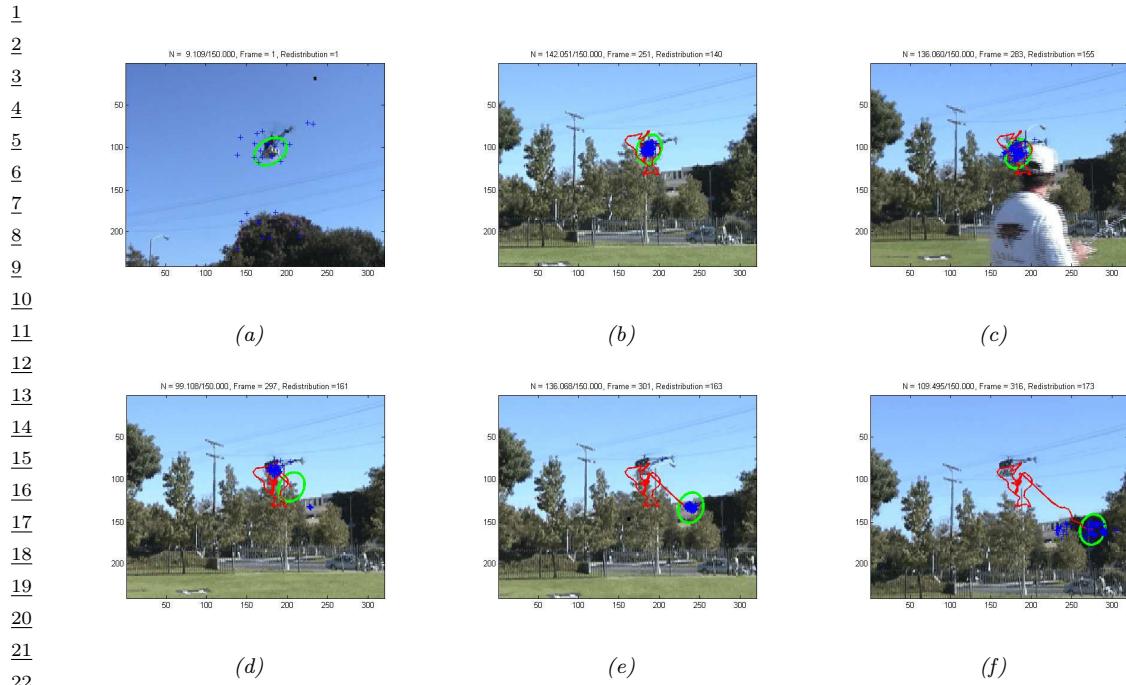


Figure 13.7: Example of particle filtering applied to visual object tracking, based on color histograms. Blue dots are posterior samples, green ellipse is Gaussian approximation to posterior. (a-c) Successful tracking. (d): Tracker gets distracted by an outlier gray patch in the background, and moves the posterior mean away from the object. (e-f): Losing track. See text for details. Used with kind permission of Sébastien Paris.

filter [JB16b]), since we are assuming the state space is discrete. However, since the number of states, K , is often very large, the $O(K^2)$ time complexity per update is prohibitive. We can use a particle filter as a sparse approximation to the belief state. This is known as **Monte Carlo localization** [TBF06].

Figure 13.8 gives an example of the method in action. The robot uses a sonar range finder, so it can only sense distance to obstacles. It starts out with a uniform prior, reflecting the fact that the owner of the robot may have turned it on in an arbitrary location. (Figuring out where you are, starting from a uniform prior, is called **global localization**.) After the first scan, which indicates two walls on either side, the belief state is shown in (b). The posterior is still fairly broad, since the robot could be in any location where the walls are fairly close by, such as a corridor or any of the narrow rooms. After moving to location 2, the robot is pretty sure it must be in a corridor and not a room, as shown in (c). After moving to location 3, the sensor is able to detect the end of the corridor. However, due to symmetry, it is not sure if it is in location I (the true location) or location II. (This is an example of **perceptual aliasing**, which refers to the fact that different things may look the same.) After moving to locations 4 and 5, it is finally able to figure out precisely where it is (not shown). The whole process is analogous to someone getting lost in an office building, and wandering the corridors until they see a sign they recognize.

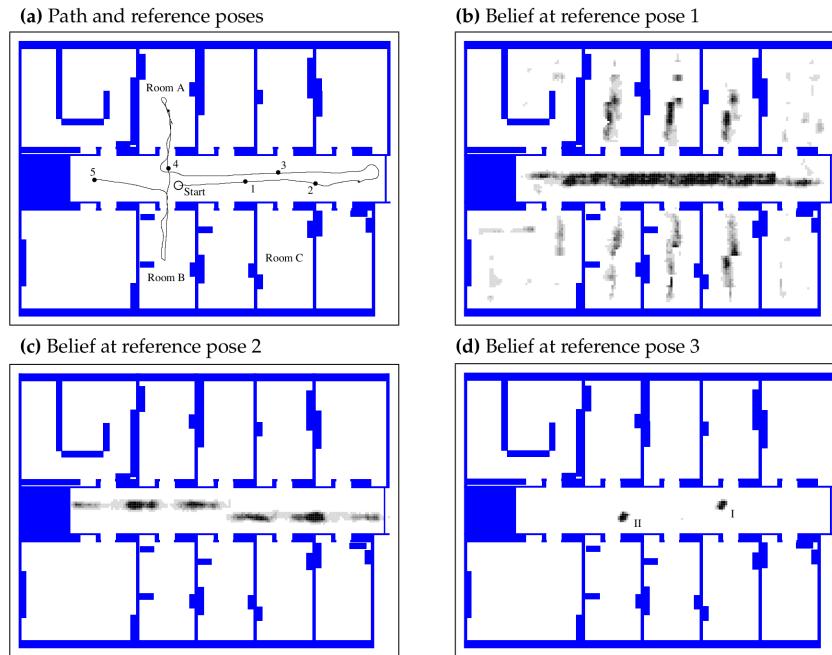


Figure 13.8: Illustration of Monte Carlo localization for a mobile robot in an office environment using a sonar sensor. From Figure 8.7 of [TBF06]. Used with kind permission of Sebastian Thrun.

13.3.4 Online parameter estimation

It is tempting to use particle filtering to perform online Bayesian inference for the parameters of a model $p(\mathbf{y}_t | \mathbf{x}_t, \boldsymbol{\theta})$, just as we did using the Kalman filter for linear regression (Section 8.4.2) and the EKF for MLPs (Section 17.6.1). However, this technique will not work. The reason is that static parameters correspond to a dynamical model with zero system noise, $p(\boldsymbol{\theta}_t | \boldsymbol{\theta}_{t-1}) = \delta(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1})$. However, such a deterministic model causes problems for particle filtering, because the particles can only be reweighted by the likelihood, but cannot be moved by the deterministic transition model. Thus the diversity in the trajectories rapidly goes to zero, and the posterior collapses [Kan+15].

It is possible to add **artificial process noise**, but this causes the influence of earlier observations to decay exponentially with time, and also “washes out” the initial prior. In Section 13.6.3, we present a solution to this problem based on SMC samplers, which generalize the particle filter by allowing static variables to be turned into a sequence by adding auxiliary random variables.

13.4 Proposal distributions

Choosing a good proposal distribution $q_t(\mathbf{z}_{1:t})$ is one of the most important factors in determining whether an SMC algorithm will give reliable estimates. We discuss various ways to choose a proposal in the sections below.

1 **13.4.1 Locally optimal proposal**

3 We define the (one-step) **locally optimal proposal distribution** $q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1})$ to be the one that
4 minimizes

6
$$J = D_{\text{KL}}(\pi_{t-1}(\mathbf{z}_{1:t-1}) q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_{1:t})) \quad (13.36)$$

7 This is given by

9
$$q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_t(\mathbf{z}_{1:t-1})} \quad (13.37)$$

12 where $\tilde{\gamma}_t(\mathbf{z}_{1:t-1}) = \int \tilde{\gamma}_t(\mathbf{z}_{1:t}) d\mathbf{z}_t$.

13 To see this, note that we have the following (where const refers to terms that are constant wrt the
14 proposal):

16
$$D_{\text{KL}}(\pi_{t-1}(\mathbf{z}_{1:t-1}) q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_{1:t})) \quad (13.38)$$

17 $= \mathbb{E}_{\pi_{t-1} q_t} [\log \{\pi_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})\} - \log \pi_t(\mathbf{z}_{1:t})] \quad (13.39)$

18 $= \mathbb{E}_{\pi_{t-1} q_t} [\log q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) - \log \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})] + \text{const} \quad (13.40)$

19 $= \mathbb{E}_{\pi_{t-1} q_t} [D_{\text{KL}}(q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}))] + \text{const} \quad (13.41)$

21 where the inner KL is minimized by choosing $q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})$.

22 Note that the subscript t specifies the t 'th distribution, so in the context of SSMs, we have
23 $\pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{1:t-1}, \mathbf{y}_{1:t})$. Thus we see that when proposing \mathbf{z}_t , we should condition on all
24 the data, including the most recent observation, \mathbf{y}_t ; this is called a **guided particle filter**, and will
25 will be better than the bootstrap filter, which proposes from the prior.

26 In general, it is intractable to compute the locally optimal proposal, but it can be done in some
27 cases. For example consider the non-Markov, but Gaussian, SSM from Equation (13.6), with target

29
$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = \tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) p(z_t | z_{t-1}) p(y_t | \mathbf{z}_{1:t}) \quad (13.42)$$

31 Since the joint distribution $p(\mathbf{z}_{1:t} | \mathbf{y}_{1:t})$ is multivariate Gaussian, we can compute the optimal proposal
32 as follows [NLS19, p35]:

33
$$q_t^*(z_t | \mathbf{z}_{1:t-1}) \propto \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_t(\mathbf{z}_{1:t-1})} \propto p(z_t | z_{t-1}) p(y_t | \mathbf{z}_{1:t}) \quad (13.43)$$

36
$$\propto \mathcal{N}\left(z_t \mid \frac{r\phi z_{t-1} + qy_t - q \sum_{s=1}^{t-1} \beta^{t-s} z_s}{q+r}, \frac{qr}{q+r}\right) \quad (13.44)$$

39 **13.4.2 Proposals based on the Laplace approximation**

41 One way to approximate the optimal proposal is to use the Laplace approximation (Section 7.4.3), as
42 suggested in [DGA00]. In particular, consider an SSM with linear-Gaussian latent dynamics and a
43 GLM likelihood. At each step, we compute the maximum $\mathbf{z}_t^* = \text{argmax} \log p(\mathbf{y}_t | \mathbf{z}_t)$ as step t (e.g.,
44 using Newton-Raphson), and then approximate the likelihood using

46
$$p(\mathbf{y}_t | \mathbf{z}_t) \approx \mathcal{N}(\mathbf{z}_t | \mathbf{z}_t^*, -\mathbf{H}_t^*) \quad (13.45)$$

47

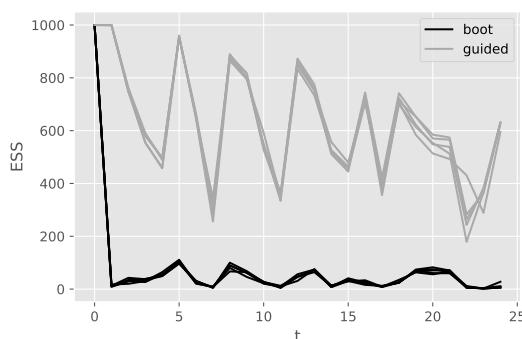


Figure 13.9: Effective sample size at each step for the bootstrap particle filter and a guided particle filter for a Gaussian SSM with Poisson likelihood. Adapted from Figure 10.4 of [CP20b]. Generated by `pf_guided_neural_decoding.ipynb`.

where \mathbf{H}_t^* is the Hessian of the log-likelihood at the mode. We now compute $p(\mathbf{z}_t | \mathbf{z}_{t-1}^i, \mathbf{y}_t)$ using the predict-update step of the Kalman filter. This combination is called the the **Laplace Gaussian filter** [Koy+10]. We give an example in Section 13.4.2.1.

13.4.2.1 Example: neural decoding

In this section, we give an example where we apply the Laplace approximation to an SSM with linear-Gaussian dynamics and a Poisson likelihood. The application arises from neuroscience. In particular, assume we record the **neural spike trains** as a monkey moves its hand around in space. Let $\mathbf{z}_t \in \mathbb{R}^6$ represent the 3d location and velocity of the hand. We model the dynamics of the hand using a simple Brownian random walk model [CP20b, p157]:

$$\begin{pmatrix} z_t(i) \\ z_t(i+3) \end{pmatrix} | \mathbf{z}_{t-1} \sim \mathcal{N}_2 \left(\begin{pmatrix} 1 & \delta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} z_{t-1}(i) \\ z_{t-1}(i+3) \end{pmatrix}, \sigma^2 \mathbf{Q} \right) \quad (13.46)$$

where the covariance of the noise is given by the following, assuming a discretization step of δ :

$$\mathbf{Q} = \begin{pmatrix} \delta^3/3 & \delta^2/2 \\ \delta^2/2 & \delta \end{pmatrix} \quad (13.47)$$

We assume the k 'th observation at time t is the number of spikes for neuron k in this sensing interval:

$$p(y_t(k) | \mathbf{z}_t) = \text{Poi}(\lambda_k(\mathbf{z}_t)) \quad (13.48)$$

$$\log \lambda_k(\mathbf{z}_t) = \alpha_k + \beta_k^\top \mathbf{z}_t \quad (13.49)$$

Our goal is to compute $p(\mathbf{z}_t | \mathbf{y}_{1:t})$, which lets us infer the position of the hand from the neural code. (Apart from basic science, this can be useful for applications such as helping disabled people control their arms using “mind control”.)

To illustrate this, we sample random parameters α and β , and then sample data from the model for 25 time steps. We then apply particle filtering to the problem, using either the bootstrap filter (i.e., proposal is the random walk prior) or the guided filter (i.e., proposal is the Laplace approximation mentioned above). In Figure 13.9, we see that the effective sample size of the guided filter is much higher than for the bootstrap filter.

13.4.3 Proposals based on the extended and unscented Kalman filter

An alternative way to create approximate proposal distributions is based on the extended Kalman filter (Section 8.5.2) and unscented Kalman filter (Section 13.4.3). This combination is called the **extended particle filter** [DGA00] and **unscented particle filter** [Mer+00]. To explain these methods, we follow the presentation of [NLS19, p36]. We assume the (non-linear and/or non-Gaussian) dynamical system can be written as follows:

$$\underline{z}_t = f(\underline{z}_{t-1}, \epsilon_t) \quad (13.50)$$

$$\underline{y}_t = h(\underline{z}_t, \eta_t) \quad (13.51)$$

where ϵ_t is the system noise and η_t is the observation noise. The EKF and UKF approximations assume that the two-slice joint distribution is Gaussian:

$$\underline{p}(\underline{z}_t, \underline{y}_t | \underline{z}_{1:t-1}) \approx \mathcal{N}\left(\begin{pmatrix} \underline{z}_t \\ \underline{y}_t \end{pmatrix} | \hat{\mu}, \hat{\Sigma}\right) \quad (13.52)$$

where

$$\hat{\mu} = \begin{pmatrix} \hat{\mu}_z \\ \hat{\mu}_y \end{pmatrix}, \hat{\Sigma} = \begin{pmatrix} \hat{\Sigma}_{zz} & \hat{\Sigma}_{zy} \\ \hat{\Sigma}_{yz} & \hat{\Sigma}_{yy} \end{pmatrix} \quad (13.53)$$

The EKF and UKF compute μ and Σ differently. In the EKF, we linearize f and h , and assume the noise terms are Gaussian. We then compute $p(\underline{z}_t, \underline{y}_t | \underline{z}_{1:t-1})$ exactly for this linearized model. In the UKF, we propagate sigma points through f and h , and approximate the resulting means and covariances (see Section 8.6.1). This avoids the need to compute the Jacobian term, and does not rely on a Gaussian assumption. For the details, see [NLS19, p56].

Once we have computed μ and Σ , we can use standard rules for Gaussian conditioning to compute the approximate proposal as follows:

$$\underline{p}(\underline{z}_t | \underline{z}_{1:t-1}, \underline{y}_t) \approx \mathcal{N}(\underline{z}_t | \mu_t, \Sigma_t) \quad (13.54)$$

$$\mu_t = \hat{\mu}_z + \hat{\Sigma}_{zy} \hat{\Sigma}_{yy}^{-1} (\underline{y}_t - \hat{\mu}_y) \quad (13.55)$$

$$\Sigma_t = \hat{\Sigma}_{zz} - \hat{\Sigma}_{zy} \hat{\Sigma}_{yy}^{-1} \hat{\Sigma}_{yz} \quad (13.56)$$

13.4.4 Proposals based on SMC

It is possible to use importance sampling, or SMC, to compute a proposal distribution, $q_t(\underline{z}_t | \underline{z}_{1:t-1}^i)$: we just run an SMC algorithm at each step t , and for each particle i , where the target distribution is the optimal proposal, $p(\underline{z}_t | \underline{z}_{1:t-1}^i, \underline{y}_{1:t})$. (Note that this is simpler than directly approximating the full joint distribution, $p(\underline{z}_{1:t} | \underline{y}_{1:t})$.) This is called **nested SMC** [NLS15; NLS19].

This method can approximate the locally optimal proposal arbitrarily well, since it does not make any limiting parametric assumptions. However, the method can be slow, since we need to create M inner samples for each of the N_s outer SMC samples. However, the nested method can sometimes give better results for the same amount of compute, even without using parallelization [NLS15; NLS19].

13.4.5 Neural adaptive SMC

Instead of manually designing proposals, it is possible to learn them. For example, suppose we represent the proposal using

$$q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}; \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{\boldsymbol{\lambda}}(\mathbf{z}_{1:t-1}), \boldsymbol{\Sigma}_{\boldsymbol{\lambda}}(\mathbf{z}_{1:t-1})) \quad (13.57)$$

where $\boldsymbol{\mu}_{\boldsymbol{\lambda}}$ and $\boldsymbol{\Sigma}_{\boldsymbol{\lambda}}$ is some kind of neural network parameterized by $\boldsymbol{\lambda}$. The key question is what objective to use to find the best $\boldsymbol{\lambda}$. We discuss some approaches in this and following sections.

In this section, we consider **adaptive proposal distributions**, in which we try to fit the proposal $q_t(\mathbf{z}_{1:t})$ to the target $\pi_t(\mathbf{z}_{1:t})$ for a specific sequence $\mathbf{y}_{1:T}$. We can either do this locally for each t , as in e.g., [CMO08], or globally, for the final time step T , as in e.g., [GGT15; PW16]. We focus on the latter. Furthermore, we assume $\pi_T(\mathbf{z}_{1:T}) = p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T})$.

Since we want the proposal to be broader than the target, we use the inclusive KL divergence, $D_{\text{KL}}(p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) \| q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda}))$, as the objective. We can rewrite this as

$$\mathcal{L} = D_{\text{KL}}(p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) \| q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda})) = - \int p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) \log q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda}) d\mathbf{z}_{1:T} + \text{const} \quad (13.58)$$

The gradient of this objective is given by

$$\nabla \mathcal{L} = -\mathbb{E}_{p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T})} \left[\sum_{t=1}^T \nabla_{\boldsymbol{\lambda}} \log q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}; \boldsymbol{\lambda}) \right] \quad (13.59)$$

Unfortunately, these expectations are wrt the intractable target distribution.

In [GGT15], they propose to use SMC to draw approximate samples from $p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T})$, using as proposals $q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}; \boldsymbol{\lambda}^{k-1})$, where $\boldsymbol{\lambda}^{k-1}$ are the parameters of the proposal from the previous step of the outer loop optimization. We then approximate

$$\nabla \mathcal{L}^k \approx \sum_{i=1}^{N_s} \sum_{t=1}^T W_T^i \nabla_{\boldsymbol{\lambda}} \log q_T(\mathbf{z}_{1:T}^i; \boldsymbol{\lambda})|_{\boldsymbol{\lambda}=\boldsymbol{\lambda}^k} \quad (13.60)$$

Although this is a biased approximation, it can work well in some cases [GGT15].

13.4.6 Amortized adaptive SMC

Another approach is to use amortized inference (Section 10.3.7) to approximate the expectations in Equation (13.59), instead of using SMC, as proposed in [PW16]. The idea is to learn a proposal that works well for any observed sequence $\mathbf{y}_{1:T}$, not just a particular sequence. We can do this by

1 optimizing the expected KL:
2

$$\begin{aligned} \underline{3} \quad \mathcal{L} &= \int p(\mathbf{y}) D_{\text{KL}}(p(\mathbf{z}_{1:T}|\mathbf{y}) \| q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda}_\phi(\mathbf{y}_{1:T}))) d\mathbf{y} \end{aligned} \quad (13.61)$$

$$\begin{aligned} \underline{4} \quad &= \int p(\mathbf{y}) \int p(\mathbf{z}|\mathbf{y}) \log \left[\frac{p(\mathbf{z}|\mathbf{y})}{q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda}_\phi(\mathbf{y}_{1:T}))} \right] d\mathbf{z} d\mathbf{y} \end{aligned} \quad (13.62)$$

$$\begin{aligned} \underline{5} \quad &= -\mathbb{E}_{p(\mathbf{z}_{1:T}, \mathbf{y}_{1:T})} [\log q_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda}_\phi(\mathbf{y}_{1:T}))] + \text{const} \end{aligned} \quad (13.63)$$

6 where $\boldsymbol{\lambda}_\phi(\mathbf{y}) = f_\phi^{\text{inf}}(\mathbf{y})$ are the parameters of the proposal computed by an inference network. We
7 can approximate the gradient of this using
8

$$\begin{aligned} \underline{9} \quad \nabla \mathcal{L}^k &= \nabla_{\boldsymbol{\lambda}} \log q_T(\tilde{\mathbf{z}}_{1:T}; \boldsymbol{\lambda}_\phi(\tilde{\mathbf{y}}_{1:T})) \end{aligned} \quad (13.64)$$

10 where we sample from the generative model, $(\tilde{\mathbf{z}}_{1:T}, \tilde{\mathbf{y}}_{1:T}) \sim p(\mathbf{z}_{1:T}, \mathbf{y}_{1:T})$. This is an unbiased
11 estimate, and does not need to use SMC in the inner loop. However, the inference network needs to
12 be trained offline on simulated data, so the method may not be optimal for any particular observed
13 sequence $\mathbf{y}_{1:T}$.

14 In many problems, the model has repeated structure, which lets us learn a single inference network
15 that can be reused multiple times.

16

17 13.4.7 Variational SMC

18 Instead of trying to learn good local proposals, we can instead try to learn proposals such that the
19 resulting joint distribution arising from the entire SMC process, $\hat{\pi}_T$, is close to the target, π_T . Let
20 $\mathbb{E}[\hat{\pi}_T]$ be the marginal distribution of a single sample of the empirical joint from SMC. One can show
21 [Nae+18] the KL divergence of this to the true distribution can be bounded as follows:

$$\begin{aligned} \underline{22} \quad D_{\text{KL}}(\mathbb{E}[\hat{\pi}_T(\mathbf{z}_{1:T}; \boldsymbol{\lambda})] \| \pi_T(\mathbf{z}_{1:T})) &\leq -\mathbb{E} \left[\log \frac{\hat{Z}_T}{Z_T} \right] \end{aligned} \quad (13.65)$$

23 where

$$\begin{aligned} \underline{24} \quad \mathbb{E} \left[\log \hat{Z}_T \right] &= \mathbb{E} \left[\sum_{t=1}^T \log \left(\frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i(\mathbf{z}_{1:t}^i; \boldsymbol{\lambda}) \right) \right] \end{aligned} \quad (13.66)$$

25 Since the KL divergence is non-negative, we have

$$\begin{aligned} \underline{26} \quad \mathbb{E} \left[\log \hat{Z}_T \right] &\leq \mathbb{E}[Z_T] \end{aligned} \quad (13.67)$$

27 Thus Equation (13.66) is an evidence lower bound (Section 10.1.2). Maximizing this results in a
28 technique called **variational SMC** [Nae+18; Le+18; Mad+17].

29 We can approximate \hat{Z}_T using SMC. If we assume the proposal distribution is reparameterizable
30 (Section 6.6.4), and if we ignore the gradient from the resampling step, we can approximate the
31 gradient of the ELBO using

$$\begin{aligned} \underline{32} \quad \nabla_{\boldsymbol{\lambda}} \mathbb{E} \left[\log \hat{Z}_T \right] &\approx \mathbb{E} \left[\sum_{t=1}^T \sum_{i=1}^N W_t^i \nabla_{\boldsymbol{\lambda}} \log \tilde{w}_t^i(\mathbf{z}_{1:t}^i; \boldsymbol{\lambda}) \right] \end{aligned} \quad (13.68)$$

13.5 Rao-Blackwellised particle filtering (RBPF)

In some models, we can partition the hidden variables into two kinds, \mathbf{c}_t and \mathbf{z}_t , such that we can analytically integrate out \mathbf{z}_t provided we know the values of $\mathbf{c}_{1:t}$. This means we only have to sample $\mathbf{c}_{1:t}$, and can represent $p(\mathbf{z}_t|\mathbf{c}_{1:t}, \mathbf{y}_{1:t})$ parametrically. These hybrid particles are sometimes called **distributional particles** or **collapsed particles** [KF09a, Sec 12.4]. This combines techniques from particle filtering (Section 13.2) with deterministic methods such as Kalman filtering (Section 8.4.1).

The advantage of this approach is that we reduce the dimensionality of the space in which we are sampling, which reduces the variance of our estimate. This technique is known as **Rao-Blackwellised particle filtering** or **RBPF** for short. (See Section 11.6.1 for more details on Rao-Blackwellisation.)

In Section 13.5.1 we give an example of RBPF for inference in a switching linear dynamical systems.

In Section 13.5.2 we illustrate RBPF for inference in the SLAM model for a mobile robot.

13.5.1 Mixture of Kalman filters

In this section, we consider the application of RBPF to the switching linear dynamical system (**SLDS**) model discussed in Section 8.8.3.1. For notational simplicity, we ignore the control inputs \mathbf{u}_t . Thus the model is given by

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}, c_t = k) = \mathcal{N}(\mathbf{z}_t|\mathbf{F}_k \mathbf{z}_{t-1}, \mathbf{Q}_k) \quad (13.69)$$

$$p(\mathbf{y}_t|\mathbf{z}_t, c_t = k) = \mathcal{N}(\mathbf{y}_t|\mathbf{H}_k \mathbf{z}_t, \mathbf{R}_k) \quad (13.70)$$

$$p(c_t = k|c_{t-1} = j) = A_{jk} \quad (13.71)$$

We let $\boldsymbol{\theta}_k = (\mathbf{F}_k, \mathbf{H}_k, \mathbf{Q}_k, \mathbf{R}_k, \mathbf{A}_{:,k})$ represent all the parameters for state k . This can be used to track a system that switches between discrete modes or operating regimes, represented by the discrete variable c_t . The key insight is that, conditional on knowing all these latent variables, $\mathbf{c}_{1:t}$, the remaining system is linear Gaussian; hence if we sample trajectories $\mathbf{c}_{1:t}^n$, we can apply a Kalman filter to each particle. This can be thought of as a **mixture of Kalman filters** [CL00]. The resulting belief state is represented by

$$p(\mathbf{z}_t, \mathbf{c}_t | \mathbf{y}_{1:t}) \approx \sum_{n=1}^N W_t^n \delta(\mathbf{c}_t - \mathbf{c}_t^n) \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_t^n, \boldsymbol{\Sigma}_t^n) \quad (13.72)$$

To derive the filtering algorithm, note that the full posterior at time t can be written as follows:

$$p(\mathbf{c}_{1:t}, \mathbf{z}_{1:t} | \mathbf{y}_{1:t}) = p(\mathbf{z}_{1:t} | \mathbf{c}_{1:t}, \mathbf{y}_{1:t}) p(\mathbf{c}_{1:t} | \mathbf{y}_{1:t}) \quad (13.73)$$

The second term is given by the following:

$$p(\mathbf{c}_{1:t} | \mathbf{y}_{1:t}) \propto p(\mathbf{y}_t | \mathbf{c}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{c}_{1:t} | \mathbf{y}_{1:t-1}) \quad (13.74)$$

$$= p(\mathbf{y}_t | \mathbf{c}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{c}_t | \mathbf{c}_{1:t-1}, \mathbf{y}_{1:t-1}) p(\mathbf{c}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.75)$$

$$= p(\mathbf{y}_t | \mathbf{c}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{c}_t | \mathbf{c}_{t-1}) p(\mathbf{c}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.76)$$

Note that, unlike the case of standard particle filtering, we cannot write $p(\mathbf{y}_t | \mathbf{c}_{1:t}, \mathbf{y}_{1:t-1}) = p(\mathbf{y}_t | \mathbf{c}_t)$, since \mathbf{c}_t does not d-separate the past observations from \mathbf{y}_t , as is evident from Figure 8.18a.

1 Suppose we form the importance distribution recursively, as follows:
 2

$$3 \quad q(\mathbf{c}_{1:t} | \mathbf{y}_{1:t}) = q(\mathbf{c}_t | \mathbf{c}_{1:t-1}, \mathbf{y}_{1:t}) q(\mathbf{c}_{1:t-1} | \mathbf{y}_{1:t}) \quad (13.77)$$

5 Then we get the unnormalized importance weights
 6

$$7 \quad \tilde{w}_t^n \propto \frac{p(\mathbf{y}_t | c_t^n, \mathbf{c}_{1:t-1}^n, \mathbf{y}_{1:t-1}) p(c_t^n | c_{t-1}^n)}{q(\mathbf{c}_t^n | \mathbf{c}_{1:t-1}^n, \mathbf{y}_{1:t})} \tilde{w}_{t-1}^n \quad (13.78)$$

9 If we propose from the prior, $q(c_t | \mathbf{c}_{t-1}^n, \mathbf{y}_{1:t}) = p(c_t | c_{t-1}^n)$, and we sample discrete state k , the
 10 weight update becomes
 11

$$12 \quad \tilde{w}_t^n \propto \tilde{w}_{t-1}^n p(\mathbf{y}_t | c_t^n = k, \mathbf{c}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \tilde{w}_{t-1}^n L_{tk}^n \quad (13.79)$$

14 where
 15

$$16 \quad L_{tk}^n = p(\mathbf{y}_t | c_t = k, \mathbf{c}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \int p(\mathbf{y}_t | c_t = k, \mathbf{z}_t) p(\mathbf{z}_t | c_t = k, \mathbf{y}_{1:t-1}, \mathbf{c}_{1:t-1}^n) d\mathbf{z}_t \quad (13.80)$$

18 The quantity L_{tk}^n is the predictive density for the new observation \mathbf{y}_t conditioned on $c_t = k$ and
 19 the history of previous latents, $\mathbf{c}_{1:t-1}^n$. In the case of SLDS models, this can be computed using
 20 the normalization constant of the Kalman filter, Equation (8.98). The resulting algorithm is shown
 21 in Algorithm 21. The step marked ‘‘KFupdate’’ refers to the Kalman filter update equations in
 22 Section 8.4.1, and is applied to each particle separately.
 23

24 **Algorithm 21:** One step of RBPF for SLDS using prior as proposal

```

26 1 for n = 1 : N do
27 2   k ~ p(c_t | c_{t-1}^n) ;
28 3   c_t^n := k;
29 4   ( $\mu_t^n, \Sigma_t^n, L_{tk}^n$ ) = KFupdate( $\mu_{t-1}^n, \Sigma_{t-1}^n, \mathbf{y}_t, \theta_k$ );
30 5    $\tilde{w}_t^n = \tilde{w}_{t-1}^n L_{tk}^n$ ;
31 6 Normalize weights:  $W_t^n = \frac{\tilde{w}_t^n}{\sum_{n'} \tilde{w}_t^{n'}}$  ;
32 7 Compute ESS =  $\frac{1}{\sum_{n=1}^N (W_t^n)^2}$ ;
33 8 if ESS < ESS_min then
34 9    $A_t^{1:N} = \text{Resample}(W_t^{1:N})$ 
35 10   $\mathbf{c}_t^{1:N} = \mathbf{c}_t^{\mathbf{A}_t}, \mu_t^{1:N} = \mu_t^{\mathbf{A}_t}, \Sigma_t^{1:N} = \Sigma_t^{\mathbf{A}_t}$  ;
36 11   $W_t^n = 1/N$  ;
37
38
39
```

40 An improved version of the algorothm can be developed based on the fact that we are sampling a
 41 discrete state space. At each step, we propagate each of the N old particles through all K possible
 42 transition models. We then compute the weight for all NK new particles, and sample from this to
 43 get the final set of N particles. This latter step can be done using the **optimal resampling** method
 44 of [FC03], which will stochastically select the particles with the largest weight, while also ensuring
 45 the result is an unbiased approximation. In addition, this approach ensures that we do not have
 46 duplicate particles, which is wasteful and unnecessary when the state space is discrete.

47

1 **13.5.1.1 Example: tracking a maneuvering object**

2 In this section we give an example of RBPF for an SLDS from [DGK01]. Our goal is to track an
3 object that has the following motion model (same as Equation (8.328)):

4 $p(\mathbf{z}_t | \mathbf{z}_{t-1}, c_t = k) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}\mathbf{z}_{t-1} + \mathbf{B}_k \mathbf{u}_t, \mathbf{Q})$ (13.81)
5

6 where $\mathbf{z}_t = (x_t, \dot{x}_t, y_t, \dot{y}_t)$ contains the 2d position and velocity, the dynamics matrix is define by
7

8
$$\mathbf{F} = \begin{pmatrix} 1 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
 (13.82)
9

10 where $\Delta = 0.1$, the noise covariance is $\mathbf{Q} = 0.2\mathbf{I}$, the input (control) signal is $\mathbf{u}_t = 1$, and the input
11 vectors are $\mathbf{b}_1 = (0, 0, 0, 0)$, $\mathbf{b}_2 = (-1.225, -0.35, 1.225, 0.35)$ and $\mathbf{b}_3 = (1.225, 0.35, -1.225, -0.35)$.
12 Thus the system will turn in different directions depending on the discrete state. We set the
13 observation model to $\mathbf{H} = \mathbf{I}$, and $\mathbf{R} = 10\text{diag}(2, 1, 2, 1)$. The discrete state transition matrix is given
14 by
15

16
$$\mathbf{A} = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$
 (13.83)
17

18 Figure 13.10a shows some observations, and the true state of the system, from a sample run, for
19 100 steps. The colors denote the discrete state, and the location of the symbol denotes the (x, y)
20 location. The small dots represent noisy observations. Figure 13.10b shows the estimate of the state
21 computed using RBPF with the optimal proposal with 1000 particles. In Figure 13.10c, we show the
22 analogous estimate using the bootstrap filter; we see that performance is worse.
23

24 In Figure 13.11a and Figure 13.11b, we show the posterior marginals of the (x, y) locations over
25 time. Figure 13.12a shows the true discrete state, and Figure 13.12b shows the approximate MAP
26 distribution over states; this has a classification error rate of 29%, but occasionally misclassifying
27 isolated time steps does not significantly hurt estimation of the continuous states, as we can see
28 from Figure 13.10b.
29

30 **13.5.2 FastSLAM**

31 We discussed the problem of simultaneous localization and mapping or SLAM in Section 31.3.2, where
32 we pointed out that the exact inference, even in the linear-Gaussian case, can be intractable for large
33 numbers of landmarks, due to the $K \times K$ covariance matrix. However, conditional on knowing the
34 robot's path, $\mathbf{r}_{1:t}$, the landmark locations are independent, i.e., $p(\mathbf{l}_t | \mathbf{r}_{1:t}, \mathbf{y}_{1:t}) = \prod_{k=1}^K p(\mathbf{l}_t^k | \mathbf{r}_{1:t}, \mathbf{y}_{1:t})$.
35 This can be seen by looking at the DGM in Figure 31.9. We can therefore sample the trajectory
36 using particle filtering, and apply Kalman filtering to each landmark independently.
37

38 In more detail, we sampling the trajectory, $\mathbf{r}_{1:t}$, assuming the map is known, as in Monte Carlo
39 localization described in Section 13.3.3. Given each sampled trajectory, we can apply the Kalman
40 filter to the remaining continuous latent variables, \mathbf{l}_t ; since these are conditionally independent given
41 $\mathbf{r}_{1:t}$, the joint posterior over the K landmarks factorizes into a product of K 2d Gaussians. Thus we
42

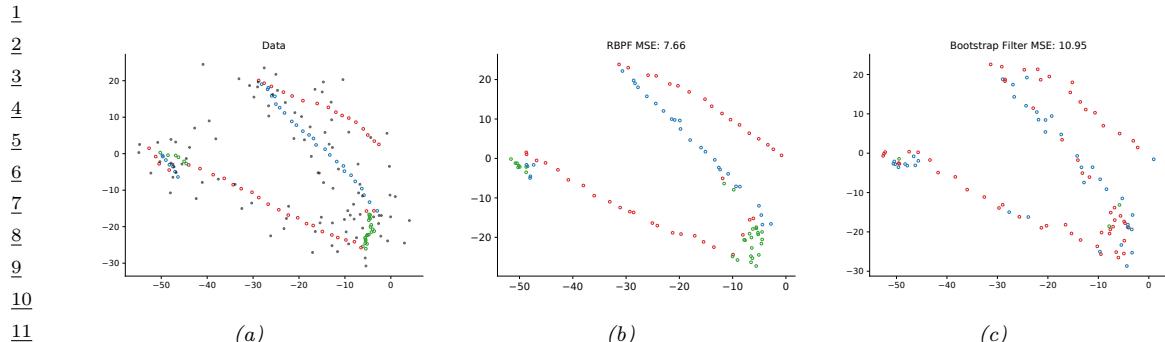


Figure 13.10: Illustration of state estimation for a switching linear model. (a) Black dots are observations, hollow circles are the true location, colors represent the discrete state. (b) Estimate from RBPF. Generated by `rpbpf_maneuver.py`. (c) Estimate from bootstrap filter. Generated by `bootstrap_filter_maneuver.py`.

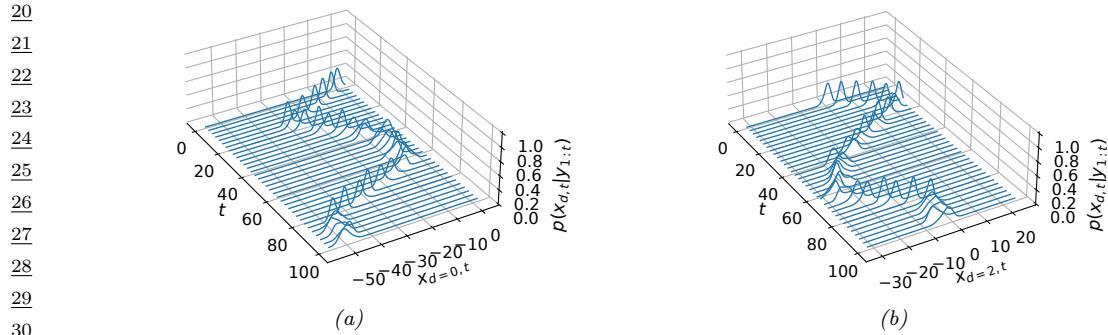


Figure 13.11: Posterior marginals of the location of the object over time, derived from the mixture of Gaussian representation. (a) x location (dimension 0). (b) y location (dimension 2). Generated by `rpbpf_maneuver_demo.py`.

run N KFs in parallel, where N is the number of samples (particles). This is feasible since each KF is fully factored, so inference takes $O(K)$ time per step.

The overall cost of this technique is $O(NK)$ per step. Fortunately, the number of particles needed for good performance is quite small, so the algorithm is essentially linear in the number of landmarks, making it quite scalable. This idea was first suggested in [Mur00], who applied it to grid-structured occupancy grids (and used the HMM filter for each particle). It was subsequently extended to landmark-based maps in [Thr+04], using the Kalman filter for each particle; they called the technique **FastSLAM**.

47

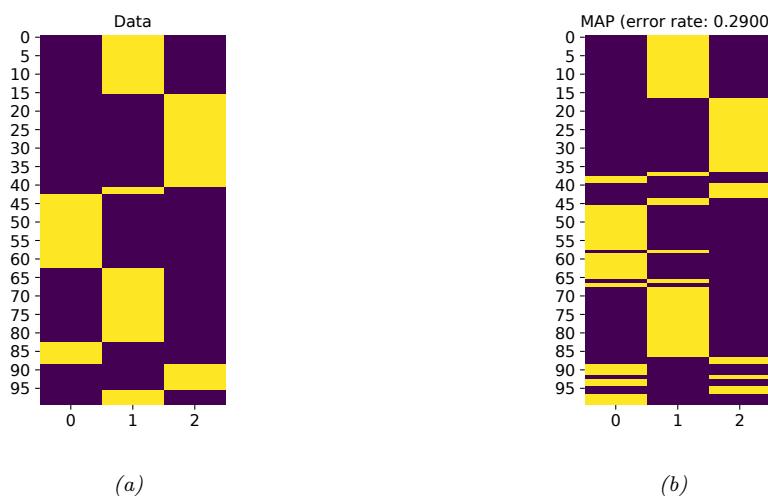


Figure 13.12: (a) Ground truth discrete state vs time, (b) Posterior distribution for the discrete state, derived from the particle representation. Generated by `rbspf_maneuver_demo.py`.

13.6 SMC samplers

In this section, we discuss **SMC samplers** (sequential Monte Carlo samplers), which are a combination of MCMC and importance sampling for sampling from a specific target distribution, $\pi(\mathbf{z}) = \tilde{\gamma}(\mathbf{z})/Z$.

The advantages of SMC samplers over standard MCMC are as follows: we can estimate the normalizing constant Z ; we can more easily develop adaptive versions that tune the transition kernel using the current set of samples; and the method is easier to parallelize (see e.g., [CCS22; Gre+22]).

The method works by defining a sequence of intermediate distributions, $\pi_t(\mathbf{z}_t)$, which we expand to a sequence of distributions over all the past variables, $\bar{\pi}_t(\mathbf{z}_{1:t})$. We then use the particle filtering algorithm to sample from each of these intermediate distributions. By marginalizing the final distribution, $\bar{\pi}_T(\mathbf{z}_{1:T})$, we recover samples from the target distribution, $\pi(\mathbf{z}) = \bar{\pi}_T(\mathbf{z}_T)$, as we explain below. (For more details, see e.g., [Dai+20a; CP20b].)

13.6.1 Ingredients of an SMC sampler

To define an SMC sampler, we need to specify several ingredients:

- A sequence of distributions defined on the same state space, $\pi_t(\mathbf{z}_t) = \tilde{\gamma}_t(\mathbf{z}_t)/Z_t$, for $t = 0 : T$;
- A **backwards kernel** $L_t(\mathbf{z}_t | \mathbf{z}_{t+1})$ (often written as $L(\mathbf{z}_t, \mathbf{z}_{t+1})$), which satisfies $\sum_{\mathbf{z}_t} L_t(\mathbf{z}_t | \mathbf{z}_{t+1}) = 1$. This allows us to create a joint distribution, $\bar{\pi}_T(\mathbf{z}_{1:T})$, from a marginal distribution, $\pi(\mathbf{z})$, by

1 working backwards from $\pi_T(\mathbf{z}_T)$, as follows:

2

$$\bar{\pi}_T(\mathbf{z}_{1:T}) = \pi_T(\mathbf{z}_T) \prod_{s=1}^{T-1} L_s(\mathbf{z}_s | \mathbf{z}_{s+1}) \quad (13.84)$$

3

4 This satisfies $\sum_{\mathbf{z}_{1:T-1}} \bar{\pi}_T(\mathbf{z}_{1:T}) = \pi_T(\mathbf{z}_T)$.

5

- 6
- 7 • A **forwards kernel** $M_t(\mathbf{z}_t | \mathbf{z}_{t-1})$ (often written as $M_t(\mathbf{z}_{t-1}, \mathbf{z}_t)$), which satisfies $\sum_{\mathbf{z}_t} M_t(\mathbf{z}_t | \mathbf{z}_{t-1}) = 1$. This can be used to propose new samples when we apply particle filtering to the above joint distribution.
- 8

9 If we define the backwards kernel to be the time reversal of the forwards kernel, things simplify
10 considerable. In particular, suppose we apply SMC to the following sequence of distributions:

11

12

$$\bar{\pi}_t(\mathbf{z}_{1:t}) = \pi_t(\mathbf{z}_t) \prod_{s=1}^{t-1} L_s(\mathbf{z}_s | \mathbf{z}_{s+1}) \quad (13.85)$$

13

14 If we use M_t as a proposal, the resulting unnormalized importance weight at step t is given by

15

16

$$\tilde{w}_t = \frac{\bar{\pi}_t(\mathbf{z}_{1:t})}{\bar{\pi}_{t-1}(\mathbf{z}_{1:t-1}) M_t(\mathbf{z}_t | \mathbf{z}_{t-1})} \propto \frac{\tilde{\gamma}_t(\mathbf{z}_t)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \frac{L_{t-1}(\mathbf{z}_{t-1} | \mathbf{z}_t)}{M_t(\mathbf{z}_t | \mathbf{z}_{t-1})} \quad (13.86)$$

17

18 The forwards kernel $M_t(\mathbf{z}_t | \mathbf{z}_{t-1})$ is usually chosen to be an MCMC kernel that leaves π_t invariant.
19 Unfortunately, it may be hard to evaluate $M_t(\mathbf{z}_t | \mathbf{z}_{t-1})$ pointwise, which makes it hard to evaluate
20 the importance weights. Fortunately, if we define the backwards kernel to be the **time reversal** of
21 the forwards kernel, the weights simplify. More precisely, suppose we define L_{t-1} so it satisfies

22

23

$$\pi_t(\mathbf{z}_t) L_{t-1}(\mathbf{z}_{t-1} | \mathbf{z}_t) = \pi_t(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t | \mathbf{z}_{t-1}) \quad (13.87)$$

24

25 In this case, the importance weight simplifies as follows:

26

27

$$\tilde{w}_t = \frac{Z_t \pi_t(\mathbf{z}_t) L_{t-1}(\mathbf{z}_{t-1} | \mathbf{z}_t)}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t | \mathbf{z}_{t-1})} \quad (13.88)$$

28

29

$$= \frac{Z_t \pi_t(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t | \mathbf{z}_{t-1})}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t | \mathbf{z}_{t-1})} \quad (13.89)$$

30

31

$$= \frac{\tilde{\gamma}_t(\mathbf{z}_{t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \quad (13.90)$$

32

33 We can use any kind of MCMC kernel for M_t . For example, if the parameters are real valued and
34 unconstrained, we can use a Markov kernel that corresponds to K steps of a random walk Metropolis-
35 Hastings sampler. We can set the covariance of the proposal to $\delta^2 \hat{\Sigma}_{t-1}$, where $\hat{\Sigma}_{t-1}$ is the empirical
36 covariance of the weighted samples from the previous step, $(W_{t-1}^{1:N}, \theta_{t-1}^{1:N})$, and $\delta = 2.38D^{-3/2}$ (which
37 is the optimal scaling parameter for RWMH). In high dimensional problems, we can use gradient
38 based Markov kernels, such as HMC [BCJ20] and NUTS [Dev+21]. For binary state spaces, we can
39 use the method of [SC13].

40

41

13.6.2 Likelihood tempering (geometric path)

There are many ways to specify the intermediate target distributions. In the **geometric path** method, we specify the intermediate distributions to be

$$(13.91)$$

where $0 = \lambda_0 < \lambda_1 < \dots < \lambda_T = 1$ are **inverse temperature** parameters, and \tilde{y}_0 is the initial proposal. If we apply particle filtering to this model, but “turn off” the resampling step, the method becomes equivalent to **annealed importance sampling** (Section 11.5.4).

In the context of Bayesian parameter inference, we often treat z as unknown parameter θ , and define $\tilde{\gamma}_0(\theta) \propto \pi_0(\theta)$ as the prior, and $\tilde{\gamma}(z) = \pi_0(\theta)p(\mathcal{D}|\theta)$ as the posterior. We can then redefine the intermediate distributions to be

$$\tilde{\gamma}_t(\boldsymbol{\theta}) = \pi_0(\boldsymbol{\theta}) \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.92)$$

where λ_t is the inverse temperature, and $\mathcal{E}(\theta)$ is the energy (potential) function. The importance weights are given by

$$\tilde{w}_t(\boldsymbol{\theta}) = \frac{\pi_0(\boldsymbol{\theta}) \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})]}{\pi_0(\boldsymbol{\theta}) \exp[-\lambda_{t-1} \mathcal{E}(\boldsymbol{\theta})]} = \exp[-\delta_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.93)$$

where $\lambda_t = \lambda_{t-1} + \delta_t$.

For this method to work well, it is important to use **adaptive tempering** to choose the λ_t so that the successive distributions are “equidistant”, for example as measured by χ^2 distance. In the case of a Gaussian prior and Gaussian energy, one can show [CP20b] that this can be achieved by picking $\lambda_t = (1 + \gamma)^{t+1} - 1$, where $\gamma > 0$ is some constant. Thus we should increase λ slowly at first, and then make bigger and bigger steps.

In practice we can estimate λ_t by setting $\lambda_t = \lambda_{t-1} + \delta_t$, where

$$\delta_t = \underset{\delta \in [0, 1 - \lambda_{t-1}]}{\operatorname{argmin}} (\text{ESSLW}(\{-\delta \mathcal{E}(\boldsymbol{\theta}_t^n)\}) - \text{ESS}_{\min}) \quad (13.94)$$

$$\text{ESSLW}(\{l_r\}) = \text{ESS}(\{e^{l_n}\}) \quad (13.95)$$

$$\text{ESS}(\{\tilde{w}_n\}) = \frac{\sum_n \tilde{w}_n^2}{(\sum_n \tilde{w}_n)^2} \quad (13.96)$$

where ESSLW computes the ESS from the log weights, $l_n = \log \tilde{w}^n$. This ensures the change in the ESS across steps is close to the desired minimum ESS, typically $0.5N$. (If there is no solution for δ in the interval, we set $\delta_t = 1 - \lambda_{t-1}$.) See Algorithm 22 for the overall algorithm.

13.6.2.1 Example: sampling from a 1d bimodal distribution

Consider the simple distribution

$$p(\theta) \propto \mathcal{N}(\theta | 0, I) \exp(-\mathcal{E}(\theta)) \quad (13.97)$$

where $\mathcal{E}(\theta) = c(||\theta||^2 - 1)^2$. We plot this in 1d in Figure 13.13a for $c = 5$; we see that it has a bimodal shape, since the low energy states correspond to parameter vectors whose norm is close to 1.

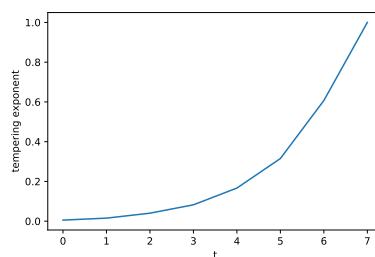
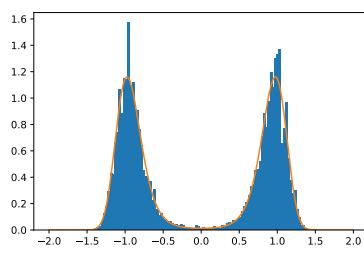
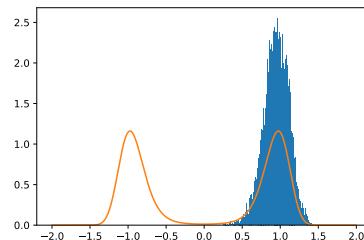
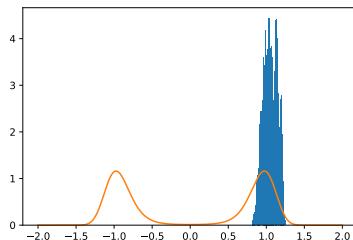
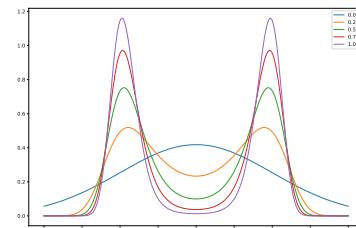
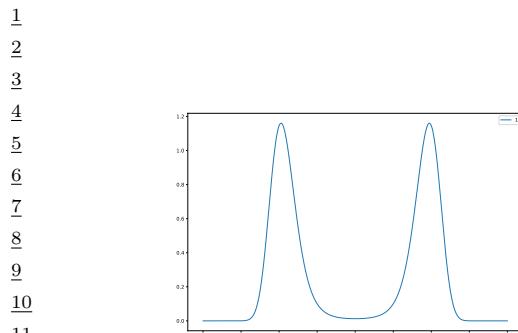


Figure 13.13: (a) Illustration of a bimodal target distribution. (b) Tempered versions of the target at different inverse temperatures, from $\lambda_T = 1$ down to $\lambda_1 = 0$. Generated by [smc_tempered_1d_bimodal.ipynb](#).

Algorithm 22: SMC with adaptive tempering

```

1    $\lambda_{-1} = 0, t = -1, W_{-1}^n = 1$ 
2   while  $\lambda_t < 1$  do
3        $t = t + 1$ 
4       if  $t = 0$  then
5            $\theta_0^n \sim \pi_0(\theta)$ 
6       else
7            $A_t^{1:N} = \text{Resample}(W_{t-1}^{1:N})$ 
8            $\theta_t^n \sim M_{\lambda_{t-1}}(\theta_{t-1}^{A_t^n}, \cdot)$ 
9       Compute  $\delta_t$  using Equation (13.94)
10       $\lambda_t = \lambda_{t-1} + \delta_t$ 
11       $w_t^n = \exp[-\delta \mathcal{E}(\theta_t^n)]$ 
12       $W_t^n = w_t^n / (\sum_{m=1}^N w_t^m)$ 
13
14
15
16

```

SMC is particularly useful for sampling from multimodal distributions, which can be provably hard to efficiently sample from using other methods, including HMC [MPS18], since gradients only provide local information about the curvature. As an example, in Figure 13.14a and Figure 13.14b we show the result of applying HMC (Section 12.5) and NUTS (Section 12.5.4.1) to this problem. We see that both algorithms get stuck near the initial state of $\theta_0 = 1$.

In Figure 13.13b, we show tempered versions of the target distribution at 5 different temperatures, chosen uniformly in the interval $[0, 1]$. We see that at $\lambda_1 = 0$, the tempered target is equal to the Gaussian prior (blue line), which is easy to sample from. Each subsequent distribution is close to the previous one, so (adaptive) SMC can track the change until it ends up at the target distribution with $\lambda_T = 1$, as shown in Figure 13.14c.

These SMC results were obtained using the adaptive tempering scheme described above. In Figure 13.14d we see that initially the temperature is small, and then it increases exponentially. The algorithm takes 8 steps until $\lambda_T \geq 1$.

13.6.3 Data tempering

If we have a set of iid observations, we can define the t 'th target to be

$$\tilde{\gamma}_t(\theta) = p(\theta)p(\mathbf{y}_{1:t}|\theta) \tag{13.98}$$

We can now apply SMC to this model. If we use a Markov kernel, the importance weight become

$$\tilde{w}_t(\theta) = \frac{p(\theta)p(\mathbf{y}_{1:t}|\theta)}{p(\theta)p(\mathbf{y}_{1:t-1}|\theta)} = p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \theta) \tag{13.99}$$

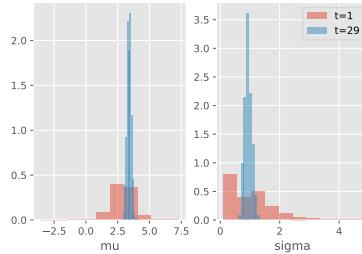
At each step, an MCMC kernel that leaves π_t invariant will typically take $O(t)$ time, since it has to evaluate $O(t)$ likelihood terms. Hence the total cost is $O(T^2)$ if there are T observations (or T minibatches). To reduce this, we can only apply the MCMC step at times t when the ESS drops below a certain level. This technique was proposed in [Cho02], who called it the **iterated batch importance sampling** or **IBIS** algorithm. See the pseudo code in Algorithm 23.

Algorithm 23: IBIS (SMC for static parameters)

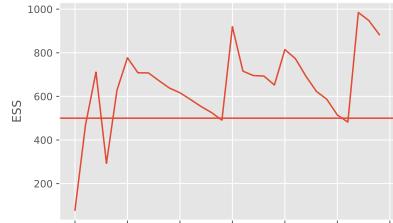
```

1    $\theta_0^n \sim \pi_0(\theta_0)$ 
2    $w_0^n = p(y_0|\theta_0^n)$ 
3    $W_0^n = w_0^n / \sum_{m=1}^N w_0^m$ 
4   for  $t = 1 : T$  do
5     if  $ESS(W_{t-1}^{1:N}) < ESS_{\min}$  then
6        $A_t^{1:N} \sim \text{resample}(W_{t-1}^{1:N})$ 
7        $\hat{w}_{t-1}^n = 1$ 
8        $\theta_t^n \sim M_t(\theta_{t-1}^{A_t^n}, \cdot)$ 
9     else
10     $A_t^n = n$ 
11     $\hat{w}_{t-1}^n = w_{t-1}^n$ 
12     $\theta_t^n = \theta_{t-1}^n$ 
13   $w_t^n = \hat{w}_{t-1}^n p(y_t|y_{0:t-1}, \theta_t^n)$ 
14   $W_t^n = w_t^n / \sum_{m=1}^N w_t^m$ 

```

192021

(a)



(b)

222324252627282930313233343536

In IBIS, once the resampling is triggered, the algorithm incurs $O(t)$ cost, to evaluate the likelihood of all the past data inside the Markov kernel M_t . We can upper bound this cost to $O(M)$ by keeping a fixed memory of at most M past examples. We can use Bayesian **core sets** to adaptively choose the best example to remember. The resulting method is called **SCMC**, which stands for sequential core-set Monte Carlo [Ber+21b].

373839404142434445

13.6.3.1 Example: IBIS for a 1d Gaussian

In this section, we give a simple example of IBIS applied to data from a 1d Gaussian, $y_t \sim \mathcal{N}(\mu = 3.14, \sigma = 1)$ for $t = 1 : 30$. The unknowns are $\theta = (\mu, \sigma)$. The prior is $p(\theta) = \mathcal{N}(\mu|0, 1)\text{Ga}(\sigma|a =$

$1, b = 1$). We use IBIS with an adaptive RWMH kernel. We use the “waste free” version of SMC [DC20], which collects all the MCMC samples for each particle for each SMC step. We use $N = 20$ particles, each updated for $K = 50$ steps, so we collect 1000 samples per time step.

Figure 13.15a shows the approximate posterior after $t = 1$ and $t = 29$ time steps. We see that the posterior concentrates on the true values of $\mu = 3.14$ and $\sigma = 1$.

Figure 13.15b plots the ESS vs time. The number of particles is 1000, and resampling (and MCMC moves) is triggered whenever this drops below 500. We see that we only need to invoke MCMC updates 7 times, and that these updates become increasingly infrequent as the posterior concentrates.

13.6.4 Sampling rare events and extrema

Suppose we want to sample values from $\pi_0(\boldsymbol{\theta})$ conditioned on the event that $S(\boldsymbol{\theta}) > \lambda^*$, where S is some score or “fitness” function. This corresponds to sampling a **rare event**, which can be hard. So it is natural to use SMC to sample from a sequence of distributions with gradually increasing thresholds:

$$\pi_t(\boldsymbol{\theta}) = \frac{1}{Z_t} \mathbb{I}(S(\boldsymbol{\theta}) \geq \lambda_t) \pi_0(\boldsymbol{\theta}) \quad (13.100)$$

with $\lambda_0 < \dots < \lambda_T = \lambda^*$. We can tackle this using likelihood tempering, where the “likelihood” is the function

$$G_t(\boldsymbol{\theta}_t) = \mathbb{I}(S(\boldsymbol{\theta}_t) \geq \lambda_t) \quad (13.101)$$

We can use SMC to generate samples from the final distribution π_T . We may also be interested in estimating

$$Z_T = p(S(\boldsymbol{\theta}) \geq \lambda_T) \quad (13.102)$$

where the probability is taken wrt $\pi_0(\boldsymbol{\theta})$.

We can adaptively set the thresholds λ_t as follows: at each step, sort the samples by their fitness, and set λ_t to the α 'th quantile. For example, if we set $\alpha = 0.5$, we keep the top 50% fittest particles. This ensures the ESS equals the minimum threshold at each step. For details, see [Cér+12].

Note that this method is very similar to the **cross-entropy method** (see supplementary material). The difference is that CEM fits a parametric distribution (e.g., a Gaussian) to the particles at each step and samples from that, rather than using a Markov kernel.

13.6.5 SMC-ABC and likelihood-free inference

The term **likelihood-free inference** refers to estimating the parameters $\boldsymbol{\theta}$ of a blackbox from which we can sample data, $\mathbf{y} \sim p(\mathbf{y}|\boldsymbol{\theta})$, but where we cannot compute the probability of that sample. Such models are called simulators, so this approach to inference is also called **simulation-based inference** (see e.g., [Nea+08; CBL20; Gou+96]). These models are also called implicit models (see Section 27.1).

If we want to approximate the posterior of a model with no known likelihood, we can use **Approximate Bayesian Computation** or **ABC** (see e.g., [Bea19; SFB18; Gut+14; Pes+21]). In this setting, we sample both parameters $\boldsymbol{\theta}$ and synthetic data \mathbf{y} such that the synthetic data

(generated from $\boldsymbol{\theta}$) is sufficiently close to the observed data \mathbf{y}^* , as judged by some distance score, $d(\mathbf{y}, \mathbf{y}^*) < \epsilon$. (For high dimensional problems, we typically require $d(\mathbf{s}(\mathbf{y}), \mathbf{s}(\mathbf{y}^*)) < \epsilon$, where $\mathbf{s}(\mathbf{y})$ is a low-dimensional summary statistic of the data.)

In SMC-ABC, we gradually decrease the discrepancy ϵ to get a series of distributions as follows:

$$\pi_t(\boldsymbol{\theta}, \mathbf{y}) = \frac{1}{Z_t} \pi_0(\boldsymbol{\theta}) p(\mathbf{y}|\boldsymbol{\theta}) \mathbb{I}(d(\mathbf{y}, \mathbf{y}^*) < \epsilon_t) \quad (13.103)$$

where $\epsilon_0 > \epsilon_1 > \dots$. This is similar to the rare event SMC samplers in Section 13.6.4, except that we can't directly evaluate the quality of a candidate $\boldsymbol{\theta}$, instead we must first convert it to data space and make the comparison there. For details, see [DMDJ12].

Although SMC-ABC is popular in some fields, such as genetics and epidemiology, this method is quite slow and does not scale to high dimensional problems. In such settings, a more efficient approach is to train a generative model to **emulate** the simulator; if this model is parametric with a tractable likelihood (e.g., a flow model), we can use the usual methods for posterior inference of its parameters (including gradient based methods like HMC). See e.g., [Bre+20a] for details.

13.6.6 SMC²

We have seen how SMC can be a useful alternative to MCMC. However it requires that we can efficiently evaluate the likelihood ratio terms $\frac{\gamma_t(\boldsymbol{\theta}_t)}{\gamma_{t-1}(\boldsymbol{\theta}_t)}$. In cases where this is not possible (e.g., for latent variable models), we can use SMC as a subroutine to approximate these likelihoods. This is called **SMC²**. For example, we can construct a “pseudo-marginal” version of IBIS as follows: the outer PF uses particles $\boldsymbol{\theta}_t^{1:N_\theta}$, and we use an inner PF on the m 'th such particle to estimate $p(\mathbf{y}_{0:t}|\boldsymbol{\theta}_t^m)$. For details, see [CP20b, Ch. 18].

Unfortunately, SMC² is not a recursive algorithm, so it cannot be used for online parameter estimation. An online extension of this method, called **recursive nested particle filter**, was proposed in [CM18].

13.7 Particle MCMC methods

In this section, we discuss some other sampling techniques that leverage the fact that SMC can give an unbiased estimate of the normalization constant Z for the target distribution. This can be useful for sampling with models where the exact likelihood is intractable. These are called **pseudo-marginal methods** [AR09].

To be more precise, note that the SMC algorithm can be seen as mapping a stream of random numbers \mathbf{u} into a set of samples, $\mathbf{z}_{1:T}^{1:N_s}$. We need random numbers $\mathbf{u}_{z,1:T}^{1:N_s}$ to specify the hidden states that are sampled at each step (using the inverse CDF of the proposal), and random numbers $\mathbf{u}_{a,1:T-1}^{1:N_s}$ to control the ancestor indices that are chosen (using the resampling algorithm), where each $u_{z,t}^i, u_{a,t}^i \sim \text{Unif}(0, 1)$. The normalization constant is also a function of these random numbers, so we denote it $\hat{Z}_t(\mathbf{u})$, where

$$\hat{Z}_t(\mathbf{u}) = \prod_{s=1}^t \frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_t^n(\mathbf{u}) \quad (13.104)$$

1 One can show (see e.g., [NLS19, p80]) that
 2

$$3 \quad \mathbb{E} [\hat{Z}_t(\mathbf{u})] = Z_t \quad (13.105)$$

4 where the expectation is wrt the distribution of \mathbf{u} , denoted $\tau(\mathbf{u})$. (Note that \mathbf{u} can be represented
 5 by a random seed.) This allows us to plug SMC inside other MCMC algorithms, as we show below.

6 Such methods are often used by **probabilistic programming systems** (see e.g., [Zho+20]),
 7 since PPLs often define models with many latent variable models defined implicitly (via sampling
 8 statements), as discussed in Section 4.5.4.
 9

10 13.7.1 Particle Marginal Metropolis Hastings

11 Suppose we want to compute the parameter posterior $p(\boldsymbol{\theta}|\mathbf{y}) = p(\mathbf{y}|\boldsymbol{\theta})p(\boldsymbol{\theta})/p(\mathbf{y})$ for a latent variable
 12 model with prior $p(\boldsymbol{\theta})$ and likelihood $p(\mathbf{y}|\boldsymbol{\theta}) = \int p(\mathbf{y}, \mathbf{h}|\boldsymbol{\theta})d\mathbf{h}$, where \mathbf{h} are latent variables (e.g., from
 13 a SSM). We can use Metropolis Hastings (Section 12.2) to avoid having to compute the partition
 14 function $p(\mathbf{y})$. However, in many cases it is intractable to compute the likelihood $p(\mathbf{y}|\boldsymbol{\theta})$ itself, due
 15 to the need to integrate over \mathbf{h} . This makes it hard to compute the MH acceptance probability
 16

$$17 \quad A = \min \left(1, \frac{p(\mathbf{y}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')q(\boldsymbol{\theta}^{j-1}|\boldsymbol{\theta}')}{p(\mathbf{y}|\boldsymbol{\theta}^{j-1})p(\boldsymbol{\theta}^{j-1})q(\boldsymbol{\theta}'|\boldsymbol{\theta}^{j-1})} \right) \quad (13.106)$$

18 where $\boldsymbol{\theta}^{j-1}$ is the parameter vector at iteration $j - 1$, and we are proposing $\boldsymbol{\theta}'$ from $q(\boldsymbol{\theta}'|\boldsymbol{\theta}^{j-1})$.
 19 However, we can use SMC to compute $\hat{Z}(\boldsymbol{\theta})$ as an unbiased approximation to $p(\mathbf{y}|\boldsymbol{\theta})$, which can be
 20 used to evaluate the MH acceptance ratio:
 21

$$22 \quad A = \min \left(1, \frac{\hat{Z}(\mathbf{u}', \boldsymbol{\theta}')p(\boldsymbol{\theta}')q(\boldsymbol{\theta}^{j-1}|\boldsymbol{\theta}')}{\hat{Z}(\mathbf{u}^{j-1}, \boldsymbol{\theta}^{j-1})p(\boldsymbol{\theta}^{j-1})q(\boldsymbol{\theta}'|\boldsymbol{\theta}^{j-1})} \right) \quad (13.107)$$

23 More precisely, we apply MH to an extended space, where we sample both the parameters $\boldsymbol{\theta}$ and the
 24 randomness \mathbf{u} for SMC.
 25

26 We can generalize the above to return samples of the latent states as well as the latent parameters,
 27 by sampling a single trajectory from
 28

$$29 \quad p(\mathbf{h}_{1:T}|\boldsymbol{\theta}, \mathbf{y}) \approx \hat{p}(\mathbf{h}|\boldsymbol{\theta}, \mathbf{y}, \mathbf{u}) = \sum_{i=1}^{N_s} W_T^i \delta(\mathbf{h}_{1:T} - \mathbf{h}_{1:T}^i) \quad (13.108)$$

30 by using the internal samples generated by SMC. Thus we can sample $\boldsymbol{\theta}$ and \mathbf{h} jointly. This is called
 31 the **particle marginal Metropolis Hastings (PMMH)** algorithm [ADH10]. See Algorithm 24
 32 for the pseudocode. See e.g. [DS15] for more practical details.
 33

34 13.7.2 Particle Independent Metropolis Hastings

35 Now suppose we just want to sample the latent states \mathbf{h} , with the parameters $\boldsymbol{\theta}$ being fixed. In
 36 this case we can simplify PMMH algorithm by not sampling $\boldsymbol{\theta}$. Since the latent states \mathbf{h} are now
 37 sampled independently of the state of the Markov chain, this is called the **particle independent**
 38

1
2 **Algorithm 24:** Particle Marginal Metropolis-Hastings
3 1 **for** $j = 1 : J$ **do**
4 2 Sample $\theta' \sim q(\theta' | \theta^{j-1})$, $\mathbf{u}' \sim \tau(\mathbf{u}')$, $\mathbf{h}' \sim \hat{p}(\mathbf{h}' | \theta', \mathbf{y}, \mathbf{u}')$
5 3 Compute $\hat{Z}(\mathbf{u}', \theta')$ using SMC
6 4 Compute A using Equation (13.107)
7 5 Sample $u \sim \text{Unif}(0, 1)$
8 6 **if** $u < A$ **then**
9 7 | Set $\theta^j = \theta'$, $\mathbf{u}^j = \mathbf{u}'$, $\mathbf{h}^j = \mathbf{h}'$
10 8 **else**
11 9 | Set $\theta^j = \theta^{j-1}$, $\mathbf{u}^j = \mathbf{u}^{j-1}$, $\mathbf{h}^j = \mathbf{h}^{j-1}$

13
14 **Algorithm 25:** Particle Independent Metropolis-Hastings
15
16 1 **for** $j = 1 : J$ **do**
17 2 Sample $\mathbf{u}' \sim \tau(\mathbf{u}')$, $\mathbf{h}' \sim \hat{p}(\mathbf{h}' | \theta, \mathbf{y}, \mathbf{u}')$
18 3 Compute $\hat{Z}(\mathbf{u}', \theta)$ using SMC
19 4 Compute $A = \min\left(1, \frac{\hat{Z}(\mathbf{u}', \theta)}{\hat{Z}(\mathbf{u}^{j-1}, \theta)}\right)$
20 5 Sample $u \sim \text{Unif}(0, 1)$
21 6 **if** $u < A$ **then**
22 7 | Set $\mathbf{u}^j = \mathbf{u}'$, $\mathbf{h}^j = \mathbf{h}'$
23 8 **else**
24 9 | Set $\mathbf{u}^j = \mathbf{u}^{j-1}$, $\mathbf{h}^j = \mathbf{h}^{j-1}$

27
28
29 **MH** algorithm. The acceptance ratio term also simplifies, since we can drop all terms involving θ .
30 See Algorithm 25 for the pseudocode.

31 One might wonder what the advantage of PIMH is over just using SMC. The answer is that PIMH
32 can return unbiased estimates of smoothing expectations, such as

33
34
$$\pi(\varphi) = \int \varphi(\mathbf{h}_{1:T}) \pi(\mathbf{h}_{1:T} | \theta, \mathbf{y}) d\mathbf{h}_{1:T} \quad (13.109)$$

35

36 whereas estimating this directly with SMC results in a consistent but biased estimate (in contrast to
37 the estimate of Z , which is unbiased). For details, see [Mid+19].

38
39 **13.7.3 Particle Gibbs**

40
41 In PMMH, we define a transition kernel that, given $(\theta^{(j-1)}, \mathbf{h}^{(j-1)})$, generates a sample $(\theta^{(j)}, \mathbf{h}^{(j)})$,
42 while leaving the target distribution invariant. Another way to perform this task is to use **particle**
43 **Gibbs sampling**, which avoids needing to specify any proposal distributions. In this approach, we
44 first sample $N - 1$ trajectories $\mathbf{h}_{1:T}^{1:N-1} \sim p(\mathbf{h} | \theta^{(j-1)}, \mathbf{y})$ using **conditional SMC**, keeping the N 'th
45 trajectory fixed at the retained particle $\mathbf{h}_{1:T}^N = \mathbf{h}^{(j-1)}$. We then sample a new value for $\mathbf{h}^{(j)}$ from
46 the empirical distribution $\hat{\pi}_T(\mathbf{h}_{1:T}^{1:N})$. Finally we sample $\theta^{(j)} \sim p(\theta | \mathbf{h}^{(j)})$. For details, see [ADH10].

47

1 Another variant, known as **particle Gibbs with ancestor sampling**, is discussed in [LJS14]; it
2 is particularly well-suited to state-space models.
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

PART III

Prediction

14 Predictive models: an overview

14.1 Introduction

The vast majority of machine learning is concerned with tackling a single problem, namely learning to predict outputs \mathbf{y} from inputs \mathbf{x} using some function f that is estimated from a labeled training set $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n) : n = 1 : N\}$, for $\mathbf{x}_n \in \mathcal{X} \subseteq \mathbb{R}^D$ and $\mathbf{y}_n \in \mathcal{Y} \subseteq \mathbb{R}^C$. We can model our uncertainty about the correct output for a given input using a conditional probability model of the form $p(\mathbf{y}|f(\mathbf{x}))$. When \mathcal{Y} is a discrete set of labels, this is called (in the ML literature) a **discriminative model**, since it lets us discriminate (distinguish) between the different possible values of \mathbf{y} . If the output is real-valued, $\mathcal{Y} = \mathbb{R}$, this is called a **regression model**. (In the statistics literature, the term “regression model” is used in both cases, even if \mathcal{Y} is a discrete set.) We will use the more generic term **“predictive model”** to refer to such models.

A predictive model can be considered as a special case of a conditional generative model (discussed in Chapter 21). In a predictive model, the output is usually low dimensional, and there is a single best answer that we want to predict. However, in most generative models, the output is usually high dimensional, such as images or sentences, and there may be many correct outputs for any given input. We will discuss a variety of types of predictive model in Section 14.1.1, but we defer the details to subsequent chapters. The rest of this chapter then discusses issues that are relevant to all types of predictive model, regardless of the specific form, such as evaluation.

14.1.1 Types of model

There are many different kinds of predictive model $p(\mathbf{y}|\mathbf{x})$. The biggest distinction is between **parametric models**, that have a fixed number of parameters independent of the size of the training set, and **non-parametric models** that have a variable number of parameters that grows with the size of the training set. Non-parametric models are usually more flexible, but can be slower to use for prediction. Parametric models are usually less flexible, but are faster to use for prediction.

Most non-parametric models are based on comparing a test input \mathbf{x} to some or all of the stored training examples $\{\mathbf{x}_n, n = 1 : N\}$, using some form of similarity, $s_n = \mathcal{K}(\mathbf{x}, \mathbf{x}_n) \geq 0$, and then predicting the output using some weighted combination of the training labels, such as $\hat{\mathbf{y}} = \sum_{n=1}^N s_n \mathbf{y}_n$. A typical example is a Gaussian process, which we discuss in Chapter 18. Other examples, such as K -nearest neighbor models, are discussed in the prequel to this book, [Mur22].

Most parametric models have the form $p(\mathbf{y}|\mathbf{x}) = p(\mathbf{y}|f(\mathbf{x}; \boldsymbol{\theta}))$, where f is some kind of function that predicts the parameters (e.g., the mean, or logits) of the output distribution (e.g., Gaussian or categorical). There are many kinds of function we can use. If f is a linear function of $\boldsymbol{\theta}$ (i.e.,

$f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \phi(\mathbf{x})$ for some *fixed* feature transformation ϕ), then the model is called a generalized linear model or GLM, which we discuss in Chapter 15. If f is a non-linear, but differentiable, function of $\boldsymbol{\theta}$ (e.g., $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}_2^\top \phi(\mathbf{x}; \boldsymbol{\theta}_1)$ for some learnable function $\phi(\mathbf{x}; \boldsymbol{\theta}_1)$), then it is common to represent f using a neural network (Chapter 16). Other types of predictive model, such as decision trees and random forests, are discussed in the prequel to this book, [Mur22].

14.1.2 Model fitting using ERM, MLE and MAP

In this section, we briefly discuss some methods used for fitting (parametric) models. The most common approach is to use **maximum likelihood estimation** or **MLE**, which amounts to solving the following optimization problem:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} p(\mathcal{D}|\boldsymbol{\theta}) = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} \log p(\mathcal{D}|\boldsymbol{\theta}) \quad (14.1)$$

If the dataset is N iid data samples, the likelihood decomposes into a product of terms, $p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^N p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$. Thus we can instead minimize the following (scaled) **negative log likelihood**:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^N [-\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})] \quad (14.2)$$

We can generalize this by replacing the **log loss** $\ell_n(\boldsymbol{\theta}) = -\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$ with a more general loss function to get

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} r(\boldsymbol{\theta}) \quad (14.3)$$

where $r(\boldsymbol{\theta})$ is the **empirical risk**

$$r(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \ell_n(\boldsymbol{\theta}) \quad (14.4)$$

This approach is called **empirical risk minimization** or **ERM**.

ERM can easily result in **overfitting**, so it is common to add a penalty or regularizer term to get

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} r(\boldsymbol{\theta}) + \lambda C(\boldsymbol{\theta}) \quad (14.5)$$

where $\lambda \geq 0$ controls the degree of regularization, and $C(\boldsymbol{\theta})$ is some complexity measure. If we use log loss, and we define $C(\boldsymbol{\theta}) = -\log \pi_0(\boldsymbol{\theta})$, where $\pi_0(\boldsymbol{\theta})$ is some prior distribution, and we use $\lambda = 1$, we recover the **MAP estimate**

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} \log p(\mathcal{D}|\boldsymbol{\theta}) + \log \pi_0(\boldsymbol{\theta}) \quad (14.6)$$

This can be solved using standard optimization methods (see Chapter 6).

14.1.3 Model fitting using Bayes, VI and generalized Bayes

Another way to prevent overfitting is to estimate a *probability distribution over parameters*, $q(\boldsymbol{\theta})$, instead of a point estimate. That is, we can try to estimate the ERM in expectation:

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{P}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] \quad (14.7)$$

If $\mathcal{P}(\Theta)$ is the space of all probability distributions over parameters, then the solution will converge to a delta function that puts all its probability on the MLE. Thus this approach, on its own, will not prevent overfitting. However, we can regularize the problem by preventing the distribution from moving too far from the prior. If we measure the divergence between q and the prior using KL divergence, we get

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{P}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] + \frac{1}{\lambda} D_{\text{KL}}(q \| \pi_0) \quad (14.8)$$

The solution to this problem is known as the **Gibbs posterior**, and is given by the following:

$$\hat{q}(\boldsymbol{\theta}) = \frac{e^{-\lambda r(\boldsymbol{\theta})} \pi_0(\boldsymbol{\theta})}{\int e^{-\lambda r(\boldsymbol{\theta}')} \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.9)$$

This is widely used in the **PAC-Bayes** community (see e.g., [Alq21]).

Now suppose we use log loss, and set $\lambda = N$, to get

$$\hat{q}(\boldsymbol{\theta}) = \frac{e^{-\sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta})} \pi_0(\boldsymbol{\theta})}{\int e^{-\sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}')} \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.10)$$

Then the resulting distribution is equivalent to the Bayes posterior:

$$\hat{q}(\boldsymbol{\theta}) = \frac{p(\mathcal{D} | \boldsymbol{\theta}) \pi_0(\boldsymbol{\theta})}{\int p(\mathcal{D} | \boldsymbol{\theta}') \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.11)$$

Often computing the Bayes posterior is intractable. We can simplify the problem by restricting attention to a limited family of distributions, $\mathcal{Q}(\Theta) \subset \mathcal{P}(\Theta)$. This gives rise to the following objective:

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{Q}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [-\log p(\mathcal{D} | \boldsymbol{\theta})] + D_{\text{KL}}(q \| \pi_0) \quad (14.12)$$

This is known as **variational inference**; see Chapter 10 for details. (See also Section 6.8, where we discuss the Bayesian learning rule.)

We can generalize this by replacing the negative log likelihood with a general risk, $r(\boldsymbol{\theta})$. Furthermore, we can replace the KL with a general divergence, $D(q || \pi_0)$, which we can weight using a general λ . This gives rise to the following objective:

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{Q}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] + \lambda D(q || \pi_0) \quad (14.13)$$

This is called **generalized Bayesian inference** [BHW16; KJD19; KJD21].

1 **14.2 Evaluating predictive models**

3 In this section we discuss how to evaluate the quality of a trained discriminative model.

5 **14.2.1 Proper scoring rules**

7 It is common to measure performance of a predictive model using a **proper scoring rule** [GR07a],
8 which is defined as follows. Let $S(p_{\theta}, (y, \mathbf{x}))$ be the score for predictive distribution $p_{\theta}(y|\mathbf{x})$ when
9 given an event $y|\mathbf{x} \sim p^*(y|\mathbf{x})$, where p^* is the true conditional distribution. (If we want to evaluate
10 a Bayesian model, where we marginalize out θ rather than condition on it, we just replace $p_{\theta}(y|\mathbf{x})$
11 with $p(y|\mathbf{x}) = \int p_{\theta}(y|\mathbf{x})p(\theta|\mathcal{D})d\theta$.) The expected score is defined by

13
$$S(p_{\theta}, p^*) = \int p^*(\mathbf{x})p^*(y|\mathbf{x})S(p_{\theta}, (y, \mathbf{x}))dyd\mathbf{x} \quad (14.14)$$

16 A proper scoring rule is one where $S(p_{\theta}, p^*) \leq S(p^*, p^*)$, with equality iff $p_{\theta}(y|\mathbf{x}) = p^*(y|\mathbf{x})$. Thus
17 maximizing such a proper scoring rule will force the model to match the true probabilities.

18 The log-likelihood, $S(p_{\theta}, (y, \mathbf{x})) = \log p_{\theta}(y|\mathbf{x})$, is a proper scoring rule. This follows from Gibbs
19 inequality:

21
$$S(p_{\theta}, p^*) = \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p_{\theta}(y|\mathbf{x})] \leq \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p^*(y|\mathbf{x})] \quad (14.15)$$

23 Therefore minimizing the NLL (aka log loss) should result in well-calibrated probabilities. However,
24 in practice, log-loss can over-emphasize tail probabilities [QC+06].

25 A common alternative is to use the **Brier score** [Bri50], which is defined as follows:

27
$$S(p_{\theta}, (y, \mathbf{x})) \triangleq \frac{1}{C} \sum_{c=1}^C (p_{\theta}(y=c|\mathbf{x}) - \mathbb{I}(y=c))^2 \quad (14.16)$$

30 This is just the squared error of the predictive distribution $\mathbf{p} = p(1:C|\mathbf{x})$ compared to the one-hot
31 label distribution \mathbf{y} . Since it based on squared error, the Brier score is less sensitive to extremely
32 rare or extremely common classes. The Brier score is also a proper scoring rule.

34 **14.2.2 Calibration**

36 A model whose predicted probabilities match the empirical frequencies is said to be **calibrated**
37 [Daw82; NMC05; Guo+17]. For example, if a classifier predicts $p(y=c|\mathbf{x}) = 0.9$, then we expect this
38 to be the true label about 90% of the time. A well-calibrated model is useful to avoid making the
39 wrong decision when the outcome is too uncertain (see e.g., ??). In the sections below, we discuss
40 some ways to measure and improve calibration.

41

42 **14.2.2.1 Expected calibration error**

44 To assess calibration, we divide the predicted probabilities into a finite set of bins or buckets, and then
45 assess the discrepancy between the empirical probability and the predicted probability by counting.
46 More precisely, suppose we have B bins. Let \mathcal{B}_b be the set of indices of samples whose prediction

47

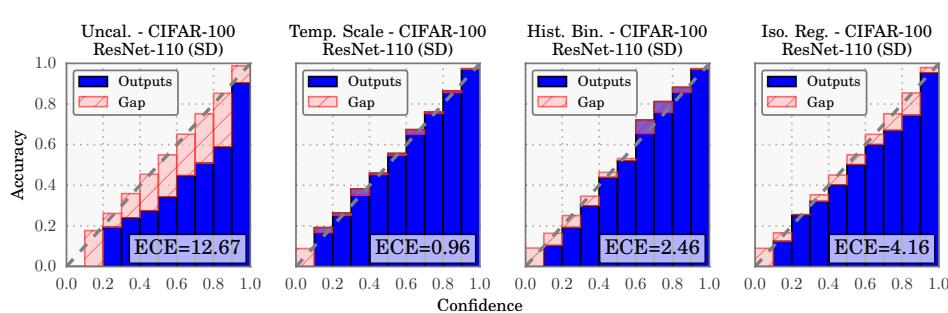


Figure 14.1: Reliability diagrams for the ResNet CNN image classifier [He+16b] applied to CIFAR-100 dataset. ECE is the expected calibration error, and measures the size of the red gap. Methods from left to right: original probabilities; after temperature scaling; after histogram binning; after isotonic regression. From Figure 4 of [Guo+17]. Used with kind permission of Chuan Guo.

confidence falls into the interval $I_b = (\frac{b-1}{B}, \frac{b}{B}]$. Here we use uniform bin widths, but we could also define the bins so that we can get an equal number of samples in each one.

Let $f(\mathbf{x})_c = p(y = c|\mathbf{x})$, $\hat{y}_n = \text{argmax}_{c \in \{1, \dots, C\}} f(\mathbf{x}_n)_c$, and $\hat{p}_n = \max_{c \in \{1, \dots, C\}} f(\mathbf{x}_n)_c$. The accuracy within bin b is defined as

$$\text{acc}(\mathcal{B}_b) = \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \mathbb{I}(\hat{y}_n = y_n) \quad (14.17)$$

The average confidence within this bin is defined as

$$\text{conf}(\mathcal{B}_b) = \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \hat{p}_n \quad (14.18)$$

If we plot accuracy vs confidence, we get a **reliability diagram**, as shown in Figure 14.1. The gap between the accuracy and confidence is shown in the red bars. We can measure this using the **expected calibration error (ECE)** [NCH15]:

$$\text{ECE}(f) = \sum_{b=1}^B \frac{|\mathcal{B}_b|}{B} |\text{acc}(\mathcal{B}_b) - \text{conf}(\mathcal{B}_b)| \quad (14.19)$$

In the multiclass case, the ECE only looks at the error of the MAP (top label) prediction. We can extend the metric to look at all the classes using the **marginal calibration error**, proposed in [KLM19]:

$$\text{MCE} = \sum_{c=1}^C w_c \mathbb{E} [(p(Y = c|f(\mathbf{x})_c) - f(\mathbf{x})_c)^2] \quad (14.20)$$

$$= \sum_{c=1}^C w_c \sum_{b=1}^B \frac{|\mathcal{B}_{b,c}|}{B} (\text{acc}(\mathcal{B}_{b,c}) - \text{conf}(\mathcal{B}_{b,c}))^2 \quad (14.21)$$

1 where $\mathcal{B}_{b,c}$ is the b 'th bin for class c , and $w_c \in [0, 1]$ denotes the importance of class c . (We can set
 2 $w_c = 1/C$ if all classes are equally important.) In [Nix+19], they call this metric **static calibration**
 3 **error**; they show that certain methods that have good ECE may have poor MCE. Other multi-class
 4 calibration metrics are discussed in [WLZ19].
 5

6

7 14.2.2.2 Improving calibration

8 In principle, training a classifier so it optimizes a proper scoring rule (such as NLL) should auto-
 9 matically result in a well-calibrated classifier. In practice, however, unbalanced datasets can result
 10 in poorly calibrated predictions. Below we discuss various ways for improving the calibration of
 11 probabilistic classifiers, following [Guo+17].
 12

13

14 14.2.2.3 Platt scaling

15 Let z be the log-odds, or logit, and $p = \sigma(z)$, produced by a probabilistic binary classifier. We wish
 16 to convert this to a more calibrated value q . The simplest way to do this is known as **Platt scaling**,
 17 and was proposed in [Pla00]. The idea is to compute $q = \sigma(az + b)$, where a and b are estimated via
 18 maximum likelihood on a validation set.

19 In the multiclass case, we can extend Platt scaling by using matrix scaling: $q = \sigma(\mathbf{W}z + \mathbf{b})$, where
 20 we estimate \mathbf{W} and \mathbf{b} via maximum likelihood on a validation set. Since \mathbf{W} has $K \times K$ parameters,
 21 where K is the number of classes, this method can easily overfit, so in practice we restrict \mathbf{W} to be
 22 diagonal.
 23

24

25 14.2.2.4 Nonparametric (histogram) methods

26 Platt scaling makes a strong assumption about how the shape of the calibration curve. A more
 27 flexible, nonparametric, method is to partition the predicted probabilities into bins, p_m , and to
 28 estimate an empirical probability q_m for each such bin; we then replace p_m with q_m ; this is known
 29 as **histogram binning** [ZE01a]. We can regularize this method by requiring that $q = f(p)$ be a
 30 piecewise constant, monotonically non-decreasing function; this is known as **isotonic regression**
 31 [ZE01a]. An alternative approach, known as the **scaling-binning calibrator**, is to apply a scaling
 32 method (such as Platt scaling), and then to apply histogram binning to that. This has the advantage
 33 of using the average of the scaled probabilities in each bin instead of the average of the observed
 34 binary labels (see Figure 14.2). In [KLM19], they prove that this results in better calibration, due to
 35 the lower variance of the estimator.

36 In the multiclass case, z is the vector of logits, and $p = \sigma(z)$ is the vector of probabilities. We
 37 wish to convert this to a better calibrated version, q . [ZE01b] propose to extend histogram binning
 38 and isotonic regression to this case by applying the above binary method to each of the K one-vs-rest
 39 problems, where K is the number of classes. However, this requires K separate calibration models,
 40 and results in an unnormalized probability distribution.
 41

42

43 14.2.2.5 Temperature scaling

44 In [Guo+17], they noticed empirically that the diagonal version of Platt scaling, when applied to
 45 a variety of DNNs, often ended learning a vector of the form $\mathbf{w} = (c, c, \dots, c)$, for some constant c .
 46 This suggests a simpler form of scaling, which they call **temperature scaling**: $q = \sigma(z/T)$, where
 47

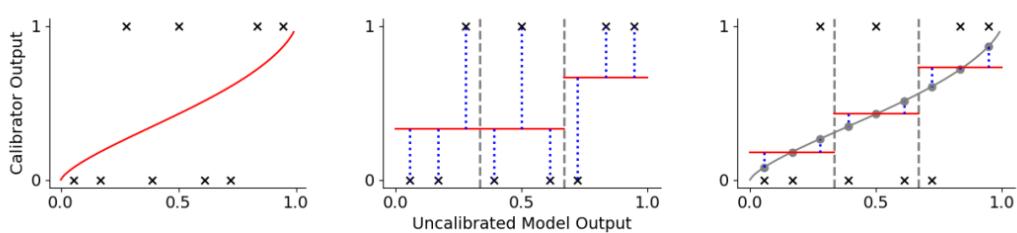


Figure 14.2: Visualization of 3 different approaches to calibrating a binary probabilistic classifier. Black crosses are the observed binary labels, red lines are the calibrated outputs. (a) Platt scaling. (b) Histogram binning with 3 bins. The output in each bin is the average of the binary labels in each bin. (c) The scaling-binning calibrator. This first applies Platt scaling, and then computes the average of the scaled points (gray circles) in each bin. From Figure 1 of [KLM19]. Used with kind permission of Ananya Kumar.

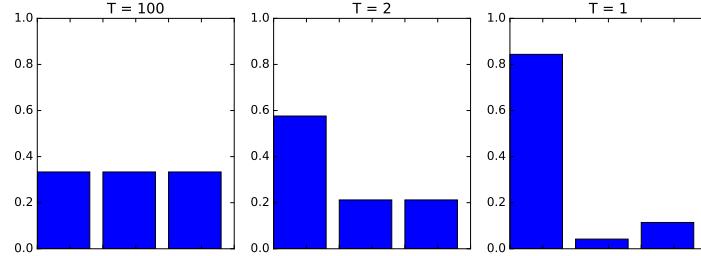


Figure 14.3: Softmax distribution $\sigma(\mathbf{a}/T)$, where $\mathbf{a} = (3, 0, 1)$, at temperatures of $T = 100$, $T = 2$ and $T = 1$. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is “spiky”, with most of its mass on the largest element. Generated by `softmax_plot.py`.

$T > 0$ is a temperature parameter, which can be estimated by maximum likelihood on the validation set. The effect of this temperature parameter is to make the distribution less peaky, as shown in Figure 14.3. [Guo+17] show empirically that this method produces the lowest ECE on a variety of DNN classification problems (see Figure 14.1 for a visualization). Furthermore, it is much simpler and faster than the other methods.

Note that Platt scaling and temperature scaling do not affect the identity of the most probable class label, so these methods have no impact on classification accuracy. However, they do improve calibration performance. A more recent multi-class calibration method is discussed in [Kul+19].

14.2.2.6 Label smoothing

When training classifiers, we usually represent the true target label as a one-hot vector, say $\mathbf{y} = (0, 1, 0)$ to represent class 2 out of 3. We can improve results if we “spread” some of the probability mass across all the bins. For example we may use $\mathbf{y} = (0.1, 0.8, 0.1)$. This is called **label smoothing** and often results in better-calibrated models [MKH19].

1
2 **14.2.2.7 Bayesian methods**

3 Bayesian approaches to fitting classifiers often result in more calibrated predictions, since they
4 represent uncertainty in the parameters. See Section 17.4.7 for an example. However, [Ova+19]
5 shows that well-calibrated models (even Bayesian ones) often become mis-calibrated when applied to
6 inputs that come from a different distribution (see Section 20.2 for details).
7

8
9 **14.2.3 Beyond evaluating marginal probabilities**

10 Calibration (Section 14.2.2) focuses on assessing properties of the marginal predictive distribution
11 $p(y|\mathbf{x})$. But this can sometimes be insufficient to distinguish between a good and bad model, especially
12 in the context of online learning and sequential decision making, as pointed out in [Lu+22; Osb+21;
13 WSG21]. For example, consider two learning agents who observe a sequence of coin tosses. Let the
14 outcome at time t be $Y_t \sim \text{Ber}(\theta)$, where θ is the unknown parameter. Agent 1 believes $\theta = 2/3$,
15 whereas agent 2 believes either $\theta = 0$ or $\theta = 1$, but is not sure which, and puts probabilities 1/3 and
16 2/3 on these events. Thus both agents, despite having different models, make identical predictions
17 for the next outcome: $p(Y_1^i = 0) = 1/3$ for agents $i = 1, 2$. However, the predictions of the two
18 agents about a *sequence* of τ future outcomes is very different: In particular, agent 1 predicts each
19 individual coin toss is a random Bernoulli event, where the probability is due to irreducible noise or
20 **aleatoric uncertainty**:

21
22
$$p(Y_1^1 = 0, \dots, Y_\tau^1 = 0) = \frac{1}{3^\tau} \quad (14.22)$$

23

24 By contrast, agent 2 predicts that the sequence will either be all heads or all tails, where the
25 probability is induced by **epistemic uncertainty** about the true parameters:

26
27
28
$$p(Y_1^2 = y_1, \dots, Y_\tau^2 = y_\tau) = \begin{cases} 1/3 & \text{if } y_1 = \dots = y_\tau = 0 \\ 2/3 & \text{if } y_1 = \dots = y_\tau = 1 \\ 0 & \text{otherwise} \end{cases} \quad (14.23)$$

29
30

31 The difference in beliefs between these agents will impact their behavior. For example, in a casino,
32 agent 1 incurs little risk on repeatedly betting on heads in the long run, but for agent 2, this would
33 be a very unwise strategy, and some initial information gathering (exploration) would be worthwhile.
34 Based on the above, we see that it is useful to evaluate *joint* predictive distributions when assessing
35 predictive models. In [Lu+22; Osb+21] they propose to evaluate the posterior predictive distributions
36 over τ outcomes $\mathbf{y} = Y_{T+1:T+\tau}$, given a set of τ inputs $\mathbf{x} = X_{T:T+\tau-1}$, and the past T data samples,
37 $\mathcal{D}_T = \{(X_t, Y_{t+1}) : t = 0, 1, \dots, T-1\}$. The Bayes optimal predictive distribution is
38

39
$$P_T^B = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.24)$$

40 This is usually intractable to compute. Instead the agent will use an approximate distribution, known
41 as a **belief state**, which we denote by
42

43
$$Q_T = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.25)$$

44

45 (We give some examples of this in Chapter 8.) The natural performance metric is the KL between
46 these distributions. Since this depend on the inputs \mathbf{x} and $\mathcal{D}_T = (X_{0:T-1}, Y_{1:T})$, we will averaged
47

the KL over these values, which are drawn iid from the true data generating distribution, which we denote by

$$P(X, Y, \mathcal{E}) = P(X|\mathcal{E})P(Y|X, \mathcal{E})P(\mathcal{E}) \quad (14.26)$$

where \mathcal{E} is the true but unknown environment. Thus we define our metric as

$$d_{B,Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T)} [D_{\text{KL}}(P^B(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.27)$$

where

$$P(\mathbf{x}, \mathcal{D}_T, \mathcal{E}) = P(\mathcal{E}) \underbrace{\left[\prod_{t=0}^{T-1} P(X_t|\mathcal{E})P(Y_{t+1}|X_t, \mathcal{E}) \right]}_{P(\mathcal{D}_T|\mathcal{E})} \underbrace{\left[\prod_{t=T}^{T+\tau-1} P(x_t|\mathcal{E}) \right]}_{P(\mathbf{x}|\mathcal{E})} \quad (14.28)$$

and $P(\mathbf{x}, \mathcal{D}_T)$ marginalizes this over environments.

Unfortunately, it is usually intractable to compute the exact Bayes posterior, P_T^B , so we cannot evaluate $d_{B,Q}^{KL}$. However, in Section 14.2.3.1, we show that

$$d_{B,Q}^{KL} = d_{\mathcal{E},Q}^{KL} - \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.29)$$

where the second term is a constant wrt the agent, and the first term is given by

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{E}) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.30)$$

$$= \mathbb{E}_{P(\mathbf{y}|\mathbf{x}, \mathcal{E})P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.31)$$

Hence if we rank agents in terms of $d_{\mathcal{E},Q}^{KL}$, it will give the same results as ranking them by $d_{B,Q}^{KL}$.

To compute $d_{\mathcal{E},Q}^{KL}$ in practice, we can use a Monte Carlo approximation: we just have to sample J environments, $\mathcal{E}^j \sim P(\mathcal{E})$, sample a training set \mathcal{D}_T from each environment, $\mathcal{D}_T^j \sim P(\mathcal{D}_T|\mathcal{E}^j)$, and then sample N data vectors of length τ , $(\mathbf{x}_n^j, \mathbf{y}_n^j) \sim P(X_{T:T+\tau-1}, Y_{T+1:T+\tau}|\mathcal{E}^j)$. We can then compute

$$\hat{d}_{\mathcal{E},Q}^{KL} = \frac{1}{JN} \sum_{j=1}^J \sum_{n=1}^N \left[\log P(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{E}^j) - \log Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{D}_T^j) \right] \quad (14.32)$$

where

$$p_{jn} = P(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{E}^j) = \prod_{t=T}^{T+\tau-1} P(Y_{n,t+1}^j|X_{n,t}^j, \mathcal{E}^j) \quad (14.33)$$

$$q_{jn} = Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{D}_T^j) = \int Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \boldsymbol{\theta})Q(\boldsymbol{\theta}|\mathcal{D}_T^j)d\boldsymbol{\theta} \quad (14.34)$$

$$\approx \frac{1}{M} \sum_{m=1}^M \prod_{t=T}^{T+\tau-1} Q(Y_{n,t+1}^j|X_{n,t}^j, \boldsymbol{\theta}_m^j) \quad (14.35)$$

1 where $\theta_m^j \sim Q(\theta|\mathcal{D}_T^j)$ is a sample from the agent's posterior over the environment.
 2

3 The above assumes that $P(Y|X)$ is known; this will be the case if we use a synthetic data generator,
 4 as in the the “neural testbed” in [Osb+21]. If we just have an J empirical distributions for $P^j(X, Y)$,
 5 we can replace the KL with the cross entropy, which only differs by an additive constant:

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} [D_{KL}(P(\mathbf{y}|\mathbf{x}, \mathcal{E}) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.36)$$

$$= \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{E})} [\log P(\mathbf{y}|\mathbf{x}, \mathcal{E})]}_{\text{const}} - \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T | \mathcal{E}) P(\mathcal{E})} [\log Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)]}_{d_{\mathcal{E},Q}^{CE}} \quad (14.37)$$

11 where the latter term is just the empirical negative log likelihood (NLL) of the agent on samples
 12 from the environment. Hence if we rank agents in terms of their NLL or cross entropy $d_{\mathcal{E},Q}^{CE}$ we will
 13 get the same results as ranking them by $d_{\mathcal{E},Q}^{KL}$, which will in turn give the same results as ranking
 14 them by $d_{B,Q}^{KL}$.

15 In practice we can approximate the cross entropy as follows:

$$\hat{d}_{\mathcal{E},Q}^{CE} = -\frac{1}{JN} \sum_{j=1}^J \sum_{n=1}^N \log Q(\mathbf{y}_n^j | \mathbf{x}_n^j, \mathcal{D}_T^j) \quad (14.38)$$

20 where $\mathcal{D}_T^j \sim P^j$, and $(\mathbf{x}_n^j, \mathbf{y}_n^j) \sim P^j$.

21 An alternative to estimating the KL or NLL is to evaluate the joint predictive accuracy by using it
 22 in a downstream task. In [Osb+21], they show that good predictive accuracy (for $\tau > 1$) correlates
 23 with good performance on a bandit problem (see Section 36.4). In [WSG21] they show that good
 24 predictive accuracy (for $\tau > 1$) results in good performance on a transductive active learning task.
 25

26 14.2.3.1 Proof of claim

27 We now prove Equation (14.29), based on [Lu+21]. First note that

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E}) P(\mathbf{y}|\mathbf{x}, \mathcal{E})} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.39)$$

$$= \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] + \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.40)$$

34 For the first term in Equation (14.40) we have

$$\mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] = \sum P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.41)$$

$$= \sum P(\mathbf{x}, \mathcal{D}_T) \sum P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.42)$$

$$= \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T)} [D_{KL}(P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] = d_{B,Q}^{KL} \quad (14.43)$$

42 We now show that the second term in Equation (14.40) reduces to the mutual information. We
 43 exploit the fact that

$$P(\mathbf{y}|\mathbf{x}, \mathcal{E}) = P(\mathbf{y}|\mathcal{D}_T, \mathbf{x}, \mathcal{E}) = \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})}{P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})} \quad (14.44)$$



Figure 14.4: Prediction set examples on Imagenet. We show three progressively more difficult examples of the class fox squirrel and the prediction sets generated by conformal prediction. From Figure 1 of [AB21]. Used with kind permission of Anastasios Angelopoulos.

since \mathcal{D}_T has no new information in beyond \mathcal{E} . From this we get

$$\mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] = \mathbb{E} \left[\log \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})/P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})}{P(\mathbf{y}|\mathcal{D}, \mathcal{D}_T)} \right] \quad (14.45)$$

$$= \sum P(\mathcal{D}_T, \mathbf{x}) \sum P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \log \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})}{P(\mathbf{y}|\mathcal{D}_T, \mathbf{x})P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})} \quad (14.46)$$

$$= \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.47)$$

Hence

$$d_{\mathcal{E}, Q}^{KL} = d_{B, Q}^{KL} + \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.48)$$

as claimed.

14.3 Conformal prediction

In this section, we briefly discuss **conformal prediction** [VGS05; SV08; ZFV20; AB21; KSB21]. This is a simple but effective way to create prediction intervals or sets with guaranteed frequentist coverage probability from any predictive method $p(y|\mathbf{x})$. This can be seen as a form of **distribution free uncertainty quantification**, since it works without making assumptions (beyond exchangeability of the data) about the true data generating process or the form of the model.¹ Our presentation is based on the excellent tutorial of [AB21].²

In conformal prediction, we start with some heuristic notion of uncertainty — such as the softmax score for a classification problem, or the variance for a regression problem — and we use it to define a **conformal score** $s(\mathbf{x}, y) \in \mathbb{R}$, which measures how badly the output y “conforms” to \mathbf{x} . (Large

1. The exchangeability assumption rules out time series data, which is serially correlated. However, extensions to conformal prediction have been developed for the time series case, see e.g., [Zaf+22]. The exchangeability assumption also rules out distribution shift, although this has also been partially addressed, as we briefly discuss in Section 20.3.1.1.

2. See also the easy-to-use **MAPIE** Python library at <https://mapie.readthedocs.io/en/latest/index.html>, and the list of papers at <https://github.com/valeman/awesome-conformal-prediction>.

values of the score are less likely, so it is better to think of it as a non-conformity score.) Next we apply this score to a **calibration** set of n labeled examples, that was not used to train f , to get $\mathcal{S} = \{s_i = s(\mathbf{x}_i, y_i) : i = 1 : n\}$. (In Section 14.3.4, we discuss what to do when we don't have a calibration set.) The user specifies a desired confidence threshold α , say 0.1, and we then compute the $(1 - \alpha)$ quantile \hat{q} of \mathcal{S} . (In fact, we should replace $1 - \alpha$ with $\frac{\lceil(n+1)(1-\alpha)\rceil}{n}$, to account for the finite size of \mathcal{S} .) Finally, given a new test input, \mathbf{x}_{n+1} , we compute the prediction set to be

$$\mathcal{T}(\mathbf{x}_{n+1}) = \{y : s(\mathbf{x}_{n+1}, y) \leq \hat{q}\} \quad (14.49)$$

Intuitively, we include all the outputs y that are plausible given the input. See Figure 14.4 for an illustration.

Remarkably, one can show the following general result

$$1 - \alpha \leq P^*(y_{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})) \leq 1 - \alpha + \frac{1}{n+1} \quad (14.50)$$

where the probability is wrt the true distribution $P^*(\mathbf{x}_{n+1}, y_{n+1})$. We say that the prediction set has a **coverage** level of $1 - \alpha$. This holds for any value of $n \geq 1$ and $\alpha \in [0, 1]$. The only assumption is that the values (\mathbf{x}_i, y_i) are exchangeable, and hence the calibration scores s_i are also exchangeable. (We also assume the calibration set is drawn from P^* , although in Section 20.3.1.1, we discuss how to handle covariate shift.)

To see why this is true, let us sort the scores so $s_1 < \dots < s_n$, so $\hat{q} = s_i$, where $i = \frac{\lceil(n+1)(1-\alpha)\rceil}{n}$. (We assume the scores are distinct, for simplicity.) The score s_{n+1} is equally likely to fall in anywhere between the calibration points s_1, \dots, s_n , since the points are exchangeable. Hence

$$P^*(s_{n+1} \leq s_k) = \frac{k}{n+1} \quad (14.51)$$

for any $k \in \{1, \dots, n+1\}$. The event $\{y_{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})\}$ is equivalent to $\{s_{n+1} \leq \hat{q}\}$. Hence

$$P^*(y_{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})) = P^*(s_{n+1} \leq \hat{q}) = \frac{\lceil(n+1)(1-\alpha)\rceil}{n+1} \geq 1 - \alpha \quad (14.52)$$

For the proof of the upper bound, see [Lei+18].

Although this result may seem like a “free lunch”, it is worth noting that we can always achieve a desired coverage level by defining the prediction set to be all possible labels. In this case, the prediction set will be independent of the input, but it will cover the true label $1 - \alpha$ of the time. To rule out some degenerate cases, we seek prediction sets that are as small as possible (although we allow for the set to be larger to harder examples), while meeting the coverage requirement. Achieving this goal requires that we define suitable conformal scores. Below we give some examples of how to compute conformal scores $s(\mathbf{x}, y)$ for different kinds of problem.³

14.3.1 Conformalizing classification

The simplest way to apply conformal prediction to multiclass classification is to derive the conformal score from the softmax score assigned to the label using $s(\mathbf{x}, y) = 1 - f(\mathbf{x})_y$, so large values are

³ It is also possible to learn conformal scores in an end-to-end way, jointly with the predictive model, as discussed in [Stu+22].

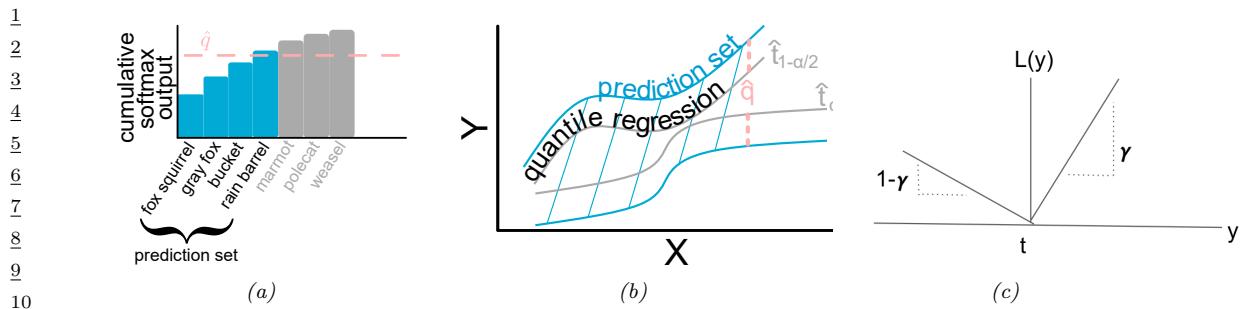


Figure 14.5: (a) Illustration of adaptive prediction set. From Figure 5 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (b) Illustrate of conformalized quantile regression. From Figure 6 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (c) Illustration of pinball loss function.

considered less likely than small values. We compute the threshold \hat{q} as described above, and then we define the prediction set to be $\mathcal{T}(\mathbf{x}) = \{y : f(\mathbf{x})_y \geq 1 - \hat{q}\}$, which matches Equation (14.49). That is, we take the set of all class labels above the specified threshold, as illustrated in Figure 14.4.

Although the above approach produces prediction sets with the smallest average size (as proved in [SLW19]), the size of the set tends to be too large for easy examples and too small for hard examples. We now present an improved method, known as **adaptive prediction sets**, due to [RSC20], which solves this problem. The idea is simple: we sort all the softmax scores, $f(\mathbf{x})_c$ for $c = 1 : C$, to get permutation $\pi_{1:C}$, and then we define $s(\mathbf{x}, y)$ to be the cumulative sum of the scores up until we reach label y : $s(\mathbf{x}, y) = \sum_{c=1}^k f(\mathbf{x})_{\pi_c}$, where $k = \pi_y$. We now compute \hat{q} as before, and define the prediction set $\mathcal{T}(\mathbf{x})$ to be the set of all labels, sorted in order of decreasing probability, until we cover \hat{q} of the probability mass. See Figure 14.5a for an illustration. This uses all the softmax scores output by the model, rather than just the top score, which accounts for its improved performance.

14.3.2 Conformalizing regression

In this section, we consider conformalized regression problems. Since now $y \in \mathbb{R}$, computing the prediction set in Equation (14.49) is expensive, so instead we will compute a prediction interval, specified by a lower and upper bound.

14.3.2.1 Conformalizing quantile regression

In this section, we use **quantile regression** to compute the lower and upper bounds. We first fit a function of the form $t_\gamma(\mathbf{x})$, which predicts the γ quantile of the pdf $P(Y|\mathbf{x})$. For example, if we set $\gamma = 0.5$, we get the median. If we use $\gamma = 0.05$ and $\gamma = 0.95$, we can get an approximate 90% prediction interval using $[t_{0.05}(\mathbf{x}), t_{0.95}(\mathbf{x})]$, as illustrated by the gray lines in Figure 14.5b. To fit the quantile regression model, we just replace squared loss with the **quantile loss**, also called the **pinball loss**, which is defined as

$$\ell_\gamma(y, \hat{t}) = (y - \hat{t})\gamma\mathbb{I}(y > \hat{t}) + (\hat{t} - y)(1 - \gamma)\mathbb{I}(y < \hat{t}) \quad (14.53)$$

where y is the true output and \hat{t} is the predicted value at quantile γ . See Figure 14.5c for an illustration.

The regression quantiles are only approximately a 90% interval because the model may be mismatched to the true distribution. However we can use conformal prediction to fix this. In particular, let us define the conformal score to be

$$s(\mathbf{x}, y) = \max(\hat{t}_{\alpha/2}(\mathbf{x}) - y, y - \hat{t}_{\alpha/2}(\mathbf{x})) \quad (14.54)$$

In other words, $s(\mathbf{x}, y)$ is a positive measure of how far the value y is outside the prediction interval, or is a negative measure if y is inside the prediction interval. We compute \hat{q} as before, and define the conformal prediction interval to be

$$\mathcal{T}(\mathbf{x}) = [\hat{t}_{\alpha/2}(\mathbf{x}) - \hat{q}, \hat{t}_{\alpha/2}(\mathbf{x}) + \hat{q}] \quad (14.55)$$

This makes the quantile regression interval wider if \hat{q} is positive (if the base method was overconfident), and narrower if \hat{q} is negative (if the base method was underconfident). See Figure 14.5b for an illustration. This approach is called **conformalized quantile regression** or **CQR** [RPC19].

14.3.2.2 Conformalizing predicted variances

There are many ways to define uncertainty scores $u(\mathbf{x})$, such as the predicted standard deviation, from which we can derive a prediction interval using

$$\mathcal{T}(\mathbf{x}) = [f(\mathbf{x}) - u(\mathbf{x})\hat{q}, f(\mathbf{x}) + u(\mathbf{x})\hat{q}] \quad (14.56)$$

Here \hat{q} is derived from the quantiles of the following conformal scores

$$s(\mathbf{x}, y) = \frac{|y - f(\mathbf{x})|}{u(\mathbf{x})} \quad (14.57)$$

The interval produced by this method tends to be wider than the one computed by CQR, since it extends an equal amount above and below the predicted value $f(\mathbf{x})$. In addition, the uncertainty measure $u(\mathbf{x})$ may not scale properly with α . Nevertheless, this is a simple post-hoc method that can be applied to many regression methods without needing to retrain them.

14.3.3 Conformalizing Bayes

Suppose we can compute the posterior predictive distribution $f(\mathbf{x})_y = p(y|\mathbf{x})$. If this is a perfect model, then the following prediction set would be optimal:

$$\mathcal{S}(\mathbf{x}) = \{y : f(\mathbf{x})_y > t\}, \text{ where } t \text{ is chosen so } \int_{y \in \mathcal{S}(\mathbf{x})} f(\mathbf{x})_y dy = 1 - \alpha \quad (14.58)$$

This set will not have the desired coverage if our modeling assumptions are wrong. However, we can conformalize it by defining $s(\mathbf{x}, y) = -f(\mathbf{x})_y$ and $\mathcal{T}(\mathbf{x}) = \{y : f(\mathbf{x})_y > -\hat{q}\}$. That is, we include all outputs above the chosen threshold. In [Hof21] they prove that this procedure has the smallest average size (Bayes risk) of any conformal procedure with $1 - \alpha$ coverage. Thus it is optimal in both the Bayesian and frequentist sense.

1
2 **14.3.4 What do we do if we don't have a calibration set?**

3 So far we have assumed access to a separate calibration set, which makes things simple. This is
4 called **split conformal prediction**. If we don't have enough data to adopt this splitting approach,
5 we can use **full conformal prediction** [VGS05], which requires fitting the model n times using
6 a leave-one-out type procedure. Alternatively we can use the more efficient Bayesian **add-one-in**
7 importance sampling procedure of [FH21], or the **jackknife+** procedure of [Bar+19].
8

9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

15 Generalized linear models

15.1 Introduction

A **generalized linear model** or **GLM** [MN89] is a conditional version of an exponential family distribution (Section 2.5). More precisely, the model has the following form:

$$p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \exp \left[\frac{y_n \eta_n - A(\eta_n)}{\sigma^2} + \log h(y_n, \sigma^2) \right] \quad (15.1)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$ is the natural parameter for the distribution, $A(\eta_n)$ is the log normalizer, $\mathcal{T}(y) = y$ is the sufficient statistic, and σ^2 is the dispersion term. Based on the results in Section 2.5.3, we can show that the mean and variance of the response variable are as follows:

$$\mu_n \triangleq \mathbb{E}[y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2] = A'(\eta_n) \triangleq \ell^{-1}(\eta_n) \quad (15.2)$$

$$\mathbb{V}[y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2] = A''(\eta_n) \sigma^2 \quad (15.3)$$

We will denote the mapping from the linear inputs to the mean of the output using $\mu_n = \ell^{-1}(\eta_n)$, where the function ℓ is known as the **link function**, and ℓ^{-1} is known as the **mean function**. This relationship is usually written as follows:

$$\ell(\mu_n) = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.4)$$

15.1.1 Examples

In this section, we give some examples of widely used GLMs.

15.1.1.1 Linear regression

Recall that linear regression has the form

$$p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right) \quad (15.5)$$

Hence

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2}(y_n - \eta_n)^2 - \frac{1}{2}\log(2\pi\sigma^2) \quad (15.6)$$

1 where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. We can write this in GLM form as follows:

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \frac{y_n \eta_n - \frac{\eta_n^2}{2}}{\sigma^2} - \frac{1}{2} \left(\frac{y_n^2}{\sigma^2} + \log(2\pi\sigma^2) \right) \quad (15.7)$$

6 We see that $A(\eta_n) = \eta_n^2/2$ and hence

$$\mathbb{E}[y_n] = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.8)$$

$$\mathbb{V}[y_n] = \sigma^2 \quad (15.9)$$

10 See Section 15.2 for details on linear regression.

13 15.1.1.2 Binomial regression

14 If the response variable is the number of successes in N_n trials, $y_n \in \{0, \dots, N_n\}$, we can use
15 **binomial regression**, which is defined by

$$p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = \text{Bin}(y_n | \boldsymbol{\sigma}(\mathbf{w}^\top \mathbf{x}_n), N_n) \quad (15.10)$$

18 We see that binary logistic regression is the special case when $N_n = 1$.

19 The log pdf is given by

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \log \mu_n + (N_n - y_n) \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.11)$$

$$= y_n \log\left(\frac{\mu_n}{1 - \mu_n}\right) + N_n \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.12)$$

26 where $\mu_n = \boldsymbol{\sigma}(\eta_n)$. To rewrite this in GLM form, let us define

$$\eta_n \triangleq \log \left[\frac{\mu_n}{(1 - \mu_n)} \right] = \log \left[\frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}} \frac{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}}{e^{-\mathbf{w}^\top \mathbf{x}_n}} \right] = \log \frac{1}{e^{-\mathbf{w}^\top \mathbf{x}_n}} = \mathbf{w}^\top \mathbf{x}_n \quad (15.13)$$

31 Hence we can write binomial regression in GLM form as follows

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \eta_n - A(\eta_n) + h(y_n) \quad (15.14)$$

34 where $h(y_n) = \log \binom{N_n}{y_n}$ and

$$A(\eta_n) = -N_n \log(1 - \mu_n) = N_n \log(1 + e^{\eta_n}) \quad (15.15)$$

38 Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = \frac{N_n e^{\eta_n}}{1 + e^{\eta_n}} = \frac{N_n}{1 + e^{-\eta_n}} = N_n \mu_n \quad (15.16)$$

42 and

$$\mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = N_n \mu_n (1 - \mu_n) \quad (15.17)$$

46 See the supplementary material for an example of binomial regression.

47

15.1.1.3 Poisson regression

If the response variable is an integer count, $y_n \in \{0, 1, \dots\}$, we can use **Poisson regression**, which is defined by

$$p(y_n | \mathbf{x}_n, \mathbf{w}) = \text{Poi}(y_n | \exp(\mathbf{w}^\top \mathbf{x}_n)) \quad (15.18)$$

where

$$\text{Poi}(y | \mu) = e^{-\mu} \frac{\mu^y}{y!} \quad (15.19)$$

is the Poisson distribution. Poisson regression is widely used in bio-statistical applications, where y_n might represent the number of diseases of a given person or place, or the number of reads at a genomic location in a high-throughput sequencing context (see e.g., [Kua+09]).

The log pdf is given by

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}) = y_n \log \mu_n - \mu_n - \log(y_n!) \quad (15.20)$$

where $\mu_n = \exp(\mathbf{w}^\top \mathbf{x}_n)$. Hence in GLM form we have

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}) = y_n \eta_n - A(\eta_n) + h(y_n) \quad (15.21)$$

where $\eta_n = \log(\mu_n) = \mathbf{w}^\top \mathbf{x}_n$, $A(\eta_n) = \mu_n = e^{\eta_n}$, and $h(y_n) = -\log(y_n!)$. Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = e^{\eta_n} = \mu_n \quad (15.22)$$

and

$$\mathbb{V}[y_n] = \frac{d^2A}{d\eta_n^2} = e^{2\eta_n} = \mu_n \quad (15.23)$$

15.1.1.4 Zero-inflated Poisson regression

In many forms of count data, the number of observed 0s is larger than what a model might expect, even after taking into account the predictors. Intuitively, this is because there may be many ways to produce no outcome. For example, consider predicting sales data for a product. If the sales are 0, does it mean the product is unpopular (so the demand is very low), or was it simply sold out (implying the demand is high, but exceed supply)? Similar problems arise in genomics, epidemiology, etc.

To handle such situations, it is common to use a **zero-inflated Poisson** or **ZIP** model. The likelihood for this model is a mixture of two distributions: a spike at 0, and a standard Poisson. Formally, we define

$$\text{ZIP}(y | \rho, \lambda) = \begin{cases} \rho + (1 - \rho) \exp(-\lambda) & \text{if } y = 0 \\ (1 - \rho) \frac{\lambda^y \exp(-\lambda)}{y!} & \text{if } y > 0 \end{cases} \quad (15.24)$$

Here ρ is the prior probability of picking the spike, and λ is the rate of the Poisson. We see that there are two “mechanisms” for generating a 0: either (with probability ρ) we chose the spike, or (with probability $1 - \rho$) we simply generate a zero count just because the rate of the Poisson is so low. (This latter event has probability $\lambda^0 e^{-\lambda} / 0! = e^{-\lambda}$.)

1 **15.1.2 GLMs with non-canonical link functions**

3 We have seen how the mean parameters of the output distribution are given by $\mu = \ell^{-1}(\eta)$, where the
4 function ℓ is the link function. There are several choices for this function, as we now discuss.

5 The **canonical link function** ℓ satisfies the property that $\theta = \ell(\mu)$, where θ are the canonical
6 (natural) parameters. Hence

8
$$\theta = \ell(\mu) = \ell(\ell^{-1}(\eta)) = \eta \quad (15.25)$$

10 This is what we have assumed so far. For example, for the Bernoulli distribution, the canonical
11 parameter is the log-odds $\eta = \log(\mu/(1 - \mu))$, which is given by the logit transform

12
$$\eta = \ell(\mu) = \text{logit}(\mu) = \log\left(\frac{\mu}{1 - \mu}\right) \quad (15.26)$$

15 The inverse of this is the sigmoid or logistic function

17
$$\mu = \ell^{-1}(\eta) = \sigma(\eta) = 1/(1 + e^{-\eta}) \quad (15.27)$$

19 However, we are free to use other kinds of link function. For example, in Section 15.4 we use

21
$$\eta = \ell(\mu) = \Phi^{-1}(\mu) \quad (15.28)$$

22
$$\mu = \ell^{-1}(\eta) = \Phi(\eta) \quad (15.29)$$

24 This is known as the **probit link function**.

25 Another link function that is sometimes used for binary responses is the **complementary log-log**
26 function

27
$$\eta = \ell(\mu) = \log(-\log(1 - \mu)) \quad (15.30)$$

29 This is used in applications where we either observe 0 events (denoted by $y = 0$) or one or more
30 (denoted by $y = 1$), where events are assumed to be governed by a Poisson distribution with rate λ .
31 Let E be the number of events. The Poisson assumption means $p(E = 0) = \exp(-\lambda)$ and hence

33
$$p(y = 0) = (1 - \mu) = p(E = 0) = \exp(-\lambda) \quad (15.31)$$

35 Thus $\lambda = -\log(1 - \mu)$. When λ is a function of covariates, we need to ensure it is positive, so we use
36 $\lambda = e^\eta$, and hence

38
$$\eta = \log(\lambda) = \log(-\log(1 - \mu)) \quad (15.32)$$

40 **15.1.3 Maximum likelihood estimation**

41 GLMs can be fit using similar methods to those that we used to fit logistic regression. In particular,
42 the negative log-likelihood has the following form (ignoring constant terms):

44
$$\text{NLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) = -\frac{1}{\sigma^2} \sum_{n=1}^N \ell_n \quad (15.33)$$

1 where

2

$$\ell_n \triangleq \eta_n y_n - A(\eta_n) \quad (15.34)$$

3 where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. For notational simplicity, we will assume $\sigma^2 = 1$.

4 We can compute the gradient for a single term as follows:

5

$$\mathbf{g}_n \triangleq \frac{\partial \ell_n}{\partial \mathbf{w}} = \frac{\partial \ell_n}{\partial \eta_n} \frac{\partial \eta_n}{\partial \mathbf{w}} = (y_n - A'(\eta_n)) \mathbf{x}_n = (y_n - \mu_n) \mathbf{x}_n \quad (15.35)$$

6 where $\mu_n = f(\mathbf{w}^\top \mathbf{x}_n)$, and f is the inverse link function that maps from canonical parameters to
7 mean parameters. (For example, in the case of logistic regression, we have $\mu_n = \sigma(\mathbf{w}^\top \mathbf{x})$.) This
8 gradient expression can be used inside SGD, or some other gradient method, in the obvious way.

9 The Hessian is given by

10

$$\mathbf{H} = \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}^\top} \text{NLL}(\mathbf{w}) = - \sum_{n=1}^N \frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} \quad (15.36)$$

11 where

12

$$\frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} = \frac{\partial \mathbf{g}_n}{\partial \mu_n} \frac{\partial \mu_n}{\partial \mathbf{w}^\top} = -\mathbf{x}_n f'(\mathbf{w}^\top \mathbf{x}_n) \mathbf{x}_n^\top \quad (15.37)$$

13 Hence

14

$$\mathbf{H} = \sum_{n=1}^N f'(\eta_n) \mathbf{x}_n \mathbf{x}_n^\top \quad (15.38)$$

15 For example, in the case of logistic regression, $f(\eta_n) = \sigma(\eta_n) = \mu_n$, and $f'(\eta_n) = \mu_n(1 - \mu_n)$. In
16 general, we see that the Hessian is positive definite, since $f'(\eta_n) > 0$; hence the negative log likelihood
17 is convex, so the MLE for a GLM is unique (assuming $f(\eta_n) > 0$ for all n).

18 15.1.4 Bayesian inference

19 To perform Bayesian inference of the parameters, we first need to specify a prior. Choosing a suitable
20 prior depends on the form of link function. For example, a “flat” or “uninformative” prior on the
21 offset term $\alpha \in \mathbb{R}$ will not translate to an uninformative prior on the probability scale if we pass α
22 through a sigmoid, as we discuss in Section 15.3.3.

23 Once we have chosen the prior, we can compute the posterior using a variety of approximate
24 inference methods. For small sample sizes, HMC (Section 12.5) is the easiest to use, since you just
25 need to write down the log likelihood and log prior, and use autograd to compute derivatives and pass
26 them to the HMC engine.¹ For large datasets, stochastic variational inference (Section 10.3.2) is often
27 more scalable. Of course, many other inference methods are possible, such as Laplace approximation
28 (Section 7.4.3), SMC (Section 13.6), etc.

29 1. For some examples of HMC applied to simple GLMs, see e.g., <https://austinrochford.com/posts/intro-prob-prog-pymc.html>. For a book-length treatment, see [GHV20].

1 **15.2 Linear regression**

3 **Linear regression** is the simplest case of a GLM. We gave a detailed introduction to this model in
4 the prequel to this book, [Mur22]. In this section, we discuss this model from a Bayesian perspective.
5

6 **15.2.1 Conjugate priors**

8 We first consider the case where just \mathbf{w} is unknown (so the observation noise variance parameter σ^2
9 is fixed), and then we consider the general case, where both σ^2 and \mathbf{w} are unknown.
10

11 **15.2.1.1 Noise variance is known**

13 The conjugate prior for linear regression has the following form:
14

$$\underline{15} \quad p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \tilde{\Sigma}) \quad (15.39)$$

17 We often use $\tilde{\mathbf{w}} = \mathbf{0}$ as the prior mean and $\tilde{\Sigma} = \tau^2 \mathbf{I}_D$ as the prior covariance. (We assume the bias
18 term is included in the weight vector, but often use a much weaker prior for it, since we typically do
19 not want to regularize the overall mean level of the output.)

20 To derive the posterior, let us first rewrite the likelihood in terms of an MVN as follows:
21

$$\underline{22} \quad \ell(\mathbf{w}) = p(\mathcal{D} | \mathbf{w}, \sigma^2) = \prod_{n=1}^N p(y_n | \mathbf{w}^\top \mathbf{x}, \sigma^2) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \quad (15.40)$$

25 where \mathbf{I}_N is the $N \times N$ identity matrix. We can then use Bayes rule for Gaussians (Equation (2.59))
26 to derive the posterior, which is as follows:
27

$$\underline{28} \quad p(\mathbf{w} | \mathbf{X}, \mathbf{y}, \sigma^2) \propto \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \tilde{\Sigma}) \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) = \mathcal{N}(\mathbf{w} | \hat{\mathbf{w}}, \hat{\Sigma}) \quad (15.41)$$

$$\underline{30} \quad \hat{\mathbf{w}} \triangleq \hat{\Sigma}^{-1} (\tilde{\Sigma}^{-1} \tilde{\mathbf{w}} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{y}) \quad (15.42)$$

$$\underline{32} \quad \hat{\Sigma} \triangleq (\tilde{\Sigma}^{-1} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.43)$$

34 where $\hat{\mathbf{w}}$ is the posterior mean, and $\hat{\Sigma}$ is the posterior covariance.
35

36 **Online inference**

38 In Section 8.4.2, we discuss the recursive least squares algorithm, which is a way to compute the
39 above posterior in an online (sequential) fashion.
40

41

42 **Connection to ridge regression**

43 Suppose $\tilde{\mathbf{w}} = \mathbf{0}$ and $\tilde{\Sigma} = \tau^2 \mathbf{I}$. In this case, the posterior mean becomes
44

$$\underline{45} \quad \hat{\mathbf{w}} = \frac{1}{\sigma^2} \hat{\Sigma} \mathbf{X}^\top \mathbf{y} = \left(\frac{\sigma^2}{\tau^2} \mathbf{I} + \mathbf{X}^\top \mathbf{X} \right)^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.44)$$

47

If we define $\lambda = \frac{\sigma^2}{\tau^2}$, we see this is equivalent to **ridge regression**, which optimizes

$$\mathcal{L}(\mathbf{w}) = \text{RSS}(\mathbf{w}) + \lambda \|\mathbf{w}\|^2 \quad (15.45)$$

where RSS is the residual sum of squares:

$$\text{RSS}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 = \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \frac{1}{2} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \quad (15.46)$$

15.2.1.2 Noise variance is unknown

In this section, we assume \mathbf{w} and σ^2 are both unknown. The likelihood is given by

$$\ell(\mathbf{w}, \sigma^2) = p(\mathcal{D} | \mathbf{w}, \sigma^2) \propto (\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right) \quad (15.47)$$

Since the regression weights now depend on σ^2 in the likelihood, the conjugate prior for \mathbf{w} has the form

$$p(\mathbf{w} | \sigma^2) = \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \sigma^2 \breve{\Sigma}) \quad (15.48)$$

For the noise variance σ^2 , the conjugate prior is based on the inverse Gamma distribution, which has the form

$$\text{IG}(\sigma^2 | \breve{a}, \breve{b}) = \frac{\breve{b}^{\breve{a}}}{\Gamma(\breve{a})} (\sigma^2)^{-(\breve{a}+1)} \exp(-\frac{\breve{b}}{\sigma^2}) \quad (15.49)$$

(See Section 2.2.2.8 for more details.) Putting these two together, we find that the joint conjugate prior is the **normal inverse Gamma** distribution:

$$\text{NIG}(\mathbf{w}, \sigma^2 | \tilde{\mathbf{w}}, \breve{\Sigma}, \breve{a}, \breve{b}) \triangleq \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \sigma^2 \breve{\Sigma}) \text{IG}(\sigma^2 | \breve{a}, \breve{b}) \quad (15.50)$$

$$\begin{aligned} &= \frac{\breve{b}^{\breve{a}}}{(2\pi)^{D/2} |\breve{\Sigma}|^{1/2} \Gamma(\breve{a})} (\sigma^2)^{-(\breve{a}+(D/2)+1)} \\ &\times \exp\left[-\frac{(\mathbf{w}-\tilde{\mathbf{w}})^\top \breve{\Sigma}^{-1} (\mathbf{w}-\tilde{\mathbf{w}}) + 2\breve{b}}{2\sigma^2}\right] \end{aligned} \quad (15.51)$$

This results in the following posterior:

$$p(\mathbf{w}, \sigma^2 | \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.52)$$

$$\hat{\mathbf{w}} = \hat{\Sigma} (\breve{\Sigma}^{-1} \tilde{\mathbf{w}} + \mathbf{X}^\top \mathbf{y}) \quad (15.53)$$

$$\hat{\Sigma} = (\breve{\Sigma}^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.54)$$

$$\hat{a} = \breve{a} + N/2 \quad (15.55)$$

$$\hat{b} = \breve{b} + \frac{1}{2} (\tilde{\mathbf{w}}^\top \breve{\Sigma}^{-1} \tilde{\mathbf{w}} + \mathbf{y}^\top \mathbf{y} - \hat{\mathbf{w}}^\top \hat{\Sigma}^{-1} \hat{\mathbf{w}}) \quad (15.56)$$

1 The expressions for $\hat{\mathbf{w}}$ and $\hat{\Sigma}$ are similar to the case where σ^2 is known. The expression for \hat{a} is also
2 intuitive, since it just updates the counts. The expression for \hat{b} can be interpreted as follows: it is
3 the prior sum of squares, \check{b} , plus the empirical sum of squares, $\mathbf{y}^\top \mathbf{y}$, plus a term due to the error in
4 the prior on \mathbf{w} .
5

6 The posterior marginals are as follows. For the variance, we have

$$\frac{8}{9} p(\sigma^2 | \mathcal{D}) = \int p(\mathbf{w} | \sigma^2, \mathcal{D}) p(\sigma^2 | \mathcal{D}) d\mathbf{w} = \text{IG}(\sigma^2 | \hat{a}, \hat{b}) \quad (15.57)$$

10 For the regression weights, it can be shown that

$$\frac{12}{13} p(\mathbf{w} | \mathcal{D}) = \int p(\mathbf{w} | \sigma^2, \mathcal{D}) p(\sigma^2 | \mathcal{D}) d\sigma^2 = \mathcal{T}(\mathbf{w} | \hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}} \hat{\Sigma}, 2 \hat{a}) \quad (15.58)$$

15 **15.2.1.3 Posterior predictive distribution**

16 In machine learning we usually care more about uncertainty (and accuracy) of our predictions, not
17 our parameter estimates. Fortunately, one can derive the posterior predictive distribution in closed
18 form. In particular, one can show that, given N' new test inputs $\tilde{\mathbf{X}}$, we have

$$\frac{21}{22} p(\tilde{\mathbf{y}} | \tilde{\mathbf{X}}, \mathcal{D}) = \int \int p(\tilde{\mathbf{y}} | \tilde{\mathbf{X}}, \mathbf{w}, \sigma^2) p(\mathbf{w}, \sigma^2 | \mathcal{D}) d\mathbf{w} d\sigma^2 \quad (15.59)$$

$$\frac{23}{24} = \int \int \mathcal{N}(\tilde{\mathbf{y}} | \tilde{\mathbf{X}}\mathbf{w}, \sigma^2 \mathbf{I}_{N'}) \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) d\mathbf{w} d\sigma^2 \quad (15.60)$$

$$\frac{25}{26} = \mathcal{T}(\tilde{\mathbf{y}} | \tilde{\mathbf{X}} \hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}} (\mathbf{I}_{N'} + \tilde{\mathbf{X}} \hat{\Sigma} \tilde{\mathbf{X}}^\top), 2 \hat{a}) \quad (15.61)$$

27 The posterior predictive mean is equivalent to “normal” linear regression, but where we plug in
28 $\hat{\mathbf{w}} = \mathbb{E}[\mathbf{w} | \mathcal{D}]$ instead of the MLE. The posterior predictive variance has two components: $\hat{b}/\hat{a}\mathbf{I}_{N'}$
29 due to the measurement noise, and $\hat{b}/\hat{a}\tilde{\mathbf{X}} \hat{\Sigma} \tilde{\mathbf{X}}^\top$ due to the uncertainty in \mathbf{w} . This latter term varies
30 depending on how close the test inputs are to the training data. The results are similar to using a
31 Gaussian prior (with fixed $\hat{\sigma}^2$), except the predictive distribution is even wider, since we are taking
32 into account uncertainty about σ^2 .
33

34 **15.2.2 Uninformative priors**

35 A common criticism of Bayesian inference is the need to use a prior. This is sometimes thought to
36 “pollute” the inferences one makes from the data. We can minimize the effect of the prior by using an
37 uninformative prior, as we discussed in Section 3.4. Below we discuss various uninformative priors
38 for linear regression.

41

42 **15.2.2.1 Jeffreys prior**

43 From Section 3.4.3.1, we know that the Jeffreys prior for the location parameter has the form
44 $p(\mathbf{w}) \propto 1$, and from Section 3.4.3.2, we know that the Jeffreys prior for the scale factor has the
45 form $p(\sigma) \propto \sigma^{-1}$. We can emulate these priors using an improper NIG prior with $\check{\mathbf{w}} = \mathbf{0}$, $\check{\Sigma} = \infty \mathbf{I}$,
46

$\check{a} = -D/2$ and $\check{b} = 0$. The corresponding posterior is given by

$$p(\mathbf{w}, \sigma^2 | \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.62)$$

$$\hat{\mathbf{w}} = \hat{\mathbf{w}}_{\text{mle}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.63)$$

$$\hat{\Sigma} = (\mathbf{X}^\top \mathbf{X})^{-1} \triangleq \mathbf{C} \quad (15.64)$$

$$\hat{a} = \frac{\nu}{2} \quad (15.65)$$

$$\hat{b} = \frac{s^2 \nu}{2} \quad (15.66)$$

$$s^2 \triangleq \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|^2}{\nu} \quad (15.67)$$

$$\nu = N - D \quad (15.68)$$

Hence the posterior distribution of the weights is given by

$$p(\mathbf{w} | \mathcal{D}) = \mathcal{T}(\mathbf{w} | \hat{\mathbf{w}}, s^2 \mathbf{C}, \nu) \quad (15.69)$$

where $\hat{\mathbf{w}}$ is the MLE. The marginals for each weight therefore have the form

$$p(w_d | \mathcal{D}) = \mathcal{T}(w_d | \hat{w}_d, s^2 C_{dd}, \nu) \quad (15.70)$$

15.2.2.2 Connection to frequentist statistics

Interestingly, the posterior when using Jeffrey's prior is formally equivalent to the **frequentist sampling distribution** of the MLE, which has the form

$$p(\hat{w}_d | \mathcal{D}^*) = \mathcal{T}(\hat{w}_d | w_d, s^2 C_{dd}, \nu) \quad (15.71)$$

where $\mathcal{D}^* = (\mathbf{X}, \mathbf{y}^*)$ is hypothetical data generated from the true model given the fixed inputs \mathbf{X} . In books on frequentist statistics, this is more commonly written in the following equivalent way (see e.g., [Ric95, p542]):

$$\frac{\hat{w}_d - w_d}{s\sqrt{C_{dd}}} \sim t_{N-D} \quad (15.72)$$

The sampling distribution is numerically the same as the posterior distribution in Equation (15.70) because $\mathcal{T}(w | \mu, \sigma^2, \nu) = \mathcal{T}(\mu | w, \sigma^2, \nu)$. However, it is semantically quite different, since the sampling distribution does not condition on the observed data, but instead is based on hypothetical data drawn from the model. See [BT73, p117] for more discussion of the equivalences between Bayesian and frequentist analysis of simple linear models when using uninformative priors.

15.2.2.3 Zellner's *g*-prior

It is often reasonable to assume an uninformative prior on σ^2 , since that is just a scalar that does not have much influence on the results, but using an uninformative prior for \mathbf{w} can be dangerous, since the strength of the prior controls how well regularized the model is, as we know from ridge regression.

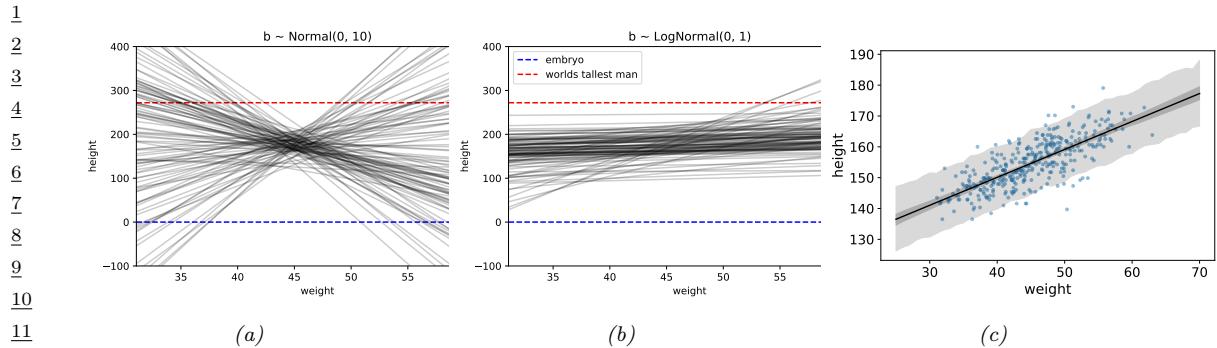


Figure 15.1: Linear regression for predicting height given weight, $y \sim \mathcal{N}(\alpha + \beta x, \sigma^2)$. (a) Prior predictive samples using a Gaussian prior for β . (b) Prior predictive samples using a Log-Gaussian prior for β . (c) Posterior predictive samples using the Log-Gaussian prior. The inner shaded band is the 95% credible interval for μ , representing epistemic uncertainty. The outer shaded band is the 95% credible interval for the observations y , which also adds data uncertainty due to σ . Adapted from Figures 4.5 and 4.10 of [McE20]. Generated by [linreg_height_weight_numpyro.ipynb](#).

A common compromise is to use an NIG prior with $\check{a} = -D/2$, $\check{b} = 0$ (to ensure $p(\sigma^2) \propto 1$) and $\check{\mathbf{w}} = \mathbf{0}$ and $\check{\Sigma} = g(\mathbf{X}^T \mathbf{X})^{-1}$, where $g > 0$ plays a role analogous to $1/\lambda$ in ridge regression. This is called Zellner's **g-prior** [Zel86].² We see that the prior covariance is proportional to $(\mathbf{X}^T \mathbf{X})^{-1}$ rather than \mathbf{I} ; this ensures that the posterior is invariant to scaling of the inputs, e.g., due to a change in the units of measurement [Min00a].

With this prior, the posterior becomes

$$p(\mathbf{w}, \sigma^2 | g, \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \mathbf{w}_N, \mathbf{V}_N, a_N, b_N) \quad (15.73)$$

$$\mathbf{V}_N = \frac{g}{g+1} (\mathbf{X}^T \mathbf{X})^{-1} \quad (15.74)$$

$$\mathbf{w}_N = \frac{g}{g+1} \hat{\mathbf{w}}_{mle} \quad (15.75)$$

$$a_N = N/2 \quad (15.76)$$

$$b_N = \frac{s^2}{2} + \frac{1}{2(g+1)} \hat{\mathbf{w}}_{mle}^T \mathbf{X}^T \mathbf{X} \hat{\mathbf{w}}_{mle} \quad (15.77)$$

Various approaches have been proposed for setting g , including cross validation, empirical Bayes [Min00a; GF00], hierarchical Bayes [Lia+08], etc.

15.2.3 Informative priors

In many problems, it is possible to use domain knowledge to come up with plausible priors. As an example, we consider the problem of predicting the height of a person given their weight. We will

² Note this prior is conditioned on the inputs \mathbf{X} , but not the outputs \mathbf{y} ; this is totally valid in a conditional (discriminative) model, where all calculations are conditioned on \mathbf{X} , which is treated like a fixed constant input.

use a dataset collected from Kalahari foragers by the anthropologist Nancy Howell (this example is from the book “Statistical Rethinking” [McE20, p93]).

Let x_i be the weight (in kg) and y_i be height (in cm) of the i 'th person, and let \bar{x} be the mean of the inputs. The observation model is given by

$$y_i \sim \mathcal{N}(\mu_i, \sigma) \quad (15.78)$$

$$\mu_i = \alpha + \beta(x_i - \bar{x}) \quad (15.79)$$

We see that the intercept α is the predicted output if $x_i = \bar{x}$, and the slope β is the predicted change in height per unit change in weight above or below the average weight.

The question is: what priors should we use? To be truly Bayesian, we should set these before looking at the data. A sensible prior for α is the height of a “typical person”, with some spread. We use $\alpha \sim \mathcal{N}(178, 20)$, since the author of the “Rethinking Statistics” book from which this example is taken is 178cm. By using a standard deviation of 20, the prior puts 95% probability on the broad range of 178 ± 40 .

What about the prior for β ? It is tempting to use a **vague prior**, or **weak prior**, such as $\beta \sim \mathcal{N}(0, 10)$, which is similar to a flat (uniform) prior, but more concentrated at 0 (a form of mild regularization). To see if this is reasonable, we can compute samples from the **prior predictive distribution**, i.e., we sample $(\alpha_s, \beta_s) \sim p(\alpha)p(\beta)$, and then plot $\alpha_s x + \beta_s$ for a range of x values, for different samples $s = 1 : S$. The results are shown in Figure 15.1a. We see that this is not a very sensible prior. For example, we see that it suggests that it is just as likely for the height to decrease with weight as increase with weight, which is not plausible. In addition, it predicts heights which are larger than the world’s tallest person (272 cm) and smaller than the world’s shortest person (an embryo, of size 0).

We can encode the monotonically increasing relationship between weight and height by restricting β to be positive. An easy way to do this is to use a log-normal or log-Gaussian prior. (If $\tilde{\beta} = \log(\beta)$ is Gaussian, then $e^{\tilde{\beta}}$ must be positive.) Specifically, we will assume $\beta \sim \mathcal{LN}(0, 1)$. Samples from this prior are shown in Figure 15.1b. This is much more reasonable.

Finally we must choose a prior over σ . In [McE20] they use $\sigma \sim \text{Unif}(0, 50)$. This ensures that σ is positive, and that the prior predictive distribution for the output is within 100cm of the average height. However, it is usually easier to specify the expected value for σ than an upper bound. To do this, we can use $\sigma \sim \text{Expon}(\lambda)$, where λ is the rate. We then set $\mathbb{E}[\sigma] = 1/\lambda$ to the value of the standard deviation that we expect. For example, we can use the empirical standard deviation of the data.

Since these priors are no longer conjugate, we cannot compute the posterior in closed form. However, we can use a variety of approximate inference methods. In this simple example, it suffices to use a quadratic (Laplace) approximation (see Section 7.4.3). The results are shown in Figure 15.1c, and look sensible.

So far, we have only considered a subset of the data, corresponding to adults over the age of 18. If we include children, we find that the mapping from weight to height is nonlinear. This is illustrated in Figure 15.2a. We can fix this problem by using **polynomial regression**. For example, consider a quadratic expansion of the standardized features x_i :

$$\mu_i = \alpha + \beta_1 x_i + \beta_2 x_i^2 \quad (15.80)$$

If we use a log-Gaussian prior for β_2 , we find that the model is too constrained, and it underfits. This is illustrated in Figure 15.2b. The reason is that we need to use an inverted quadratic with

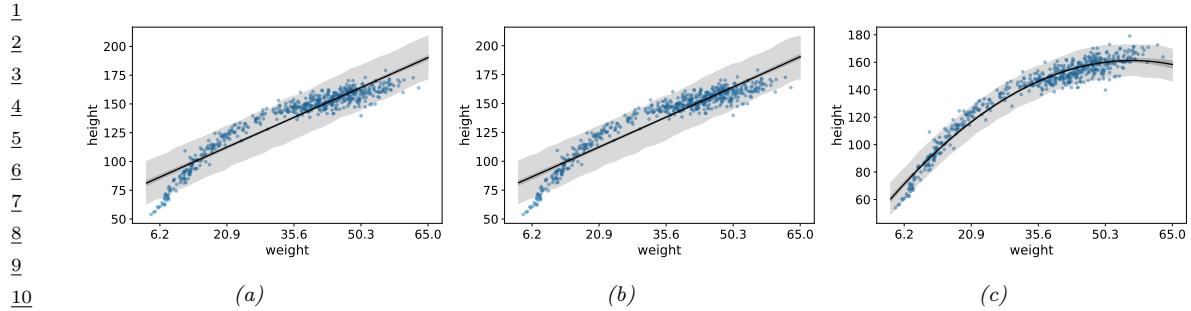


Figure 15.2: Linear regression for predicting height given weight for the full dataset (including children) using polynomial regression. (a) Posterior fit for linear model with log-Gaussian prior for β_1 . (b) Posterior fit for quadratic model with log-Gaussian prior for β_2 . (c) Posterior fit for quadratic model with Gaussian prior for β_2 . Adapted from Figure 4.11 of [McE20]. Generated by [linreg_height_weight_numpyro.ipynb](#).

a negative coefficient, but since this is disallowed by the prior, the model ends up not using this degree of freedom (we find $\mathbb{E}[\beta_2|\mathcal{D}] \approx 0.08$). If we use a Gaussian prior on β_2 , we avoid this problem, illustrated in Figure 15.2c.

This example shows that it can be useful to think about the functional form of the mapping from inputs to outputs in order to specify sensible priors.

15.2.4 Spike and slab prior

It is often useful to be able to select a subset of the input features when performing prediction, either to reduce overfitting, or to improve interpretability of the model. This can be achieved if we ensure that the weight vector \mathbf{w} is **sparse** (i.e., has many zero elements), since if $w_d = 0$, then x_d plays no role in the inner product $\mathbf{w}^\top \mathbf{x}$.

The canonical way to achieve sparsity when using Bayesian inference is to use a **spike-and-slab** (SS) prior [MB88], which has the form of a 2 component mixture model, with one component being a “spike” at 0, and the other being a uniform “slab” between $-a$ and a :

$$p(\mathbf{w}) = \prod_{d=1}^D (1 - \pi)\delta(w_d) + \pi \text{Unif}(w_d | -a, a) \quad (15.81)$$

where π is the prior probability that each coefficient is non-zero. The corresponding log prior on the coefficients is thus

$$\log p(\mathbf{w}) = \|\mathbf{w}\|_0 \log(1 - \pi) + (D - \|\mathbf{w}\|_0) \log \pi = -\lambda \|\mathbf{w}\|_0 + \text{const} \quad (15.82)$$

where $\lambda = \log \frac{\pi}{1-\pi}$ controls the sparsity of the model, and $\|\mathbf{w}\|_0 = \sum_{d=1}^D \mathbb{I}(w_d \neq 0)$ is the ℓ_0 **norm** of the weights. Thus MAP estimation with a spike and slab prior is equivalent ℓ_0 **regularization**; this penalizes the number of non-zero coefficients. Interestingly, posterior samples will also be sparse.

By contrast, consider using a Laplace prior. The **lasso** estimator uses MAP estimation, which results in a sparse estimate. However, posterior samples are not sparse. Interestingly, [EY09] show

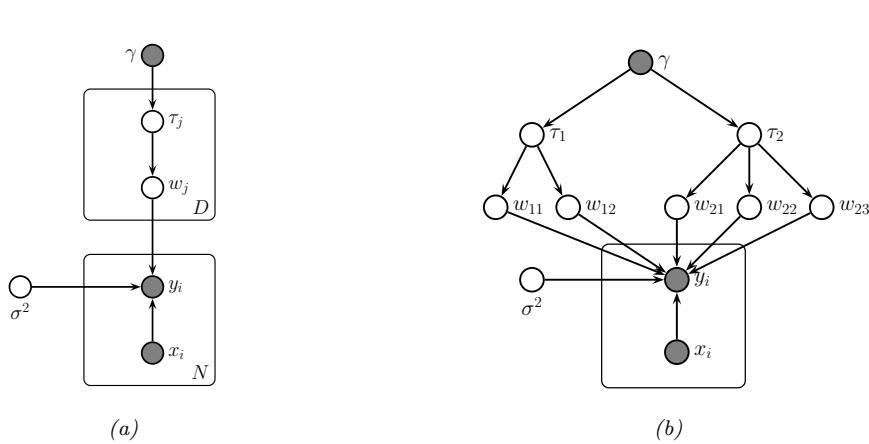


Figure 15.3: (a) Representing lasso using a Gaussian scale mixture prior. (b) Graphical model for group lasso with 2 groups, the first has size $G_1 = 2$, the second has size $G_2 = 3$.

theoretically (and [SPZ09] confirm experimentally) that using the posterior mean with a spike-and-slab prior also results in better prediction accuracy than using the posterior mode with a Laplace prior.

In practice, we often approximate the uniform slab with a broad Gaussian distribution,

$$p(\mathbf{w}) = \prod_d (1 - \pi)\delta(w_d) + \pi\mathcal{N}(w_d | 0, \sigma_w^2) \quad (15.83)$$

As $\sigma_w^2 \rightarrow \infty$, the second term approaches a uniform distribution over $[-\infty, +\infty]$. We can implement the mixture model by associating a binary random variable, $s_d \sim \text{Ber}(\pi)$, with each coefficient, to indicate if the coefficient is “on” or “off”.

Unfortunately, MAP estimation (not to mention full Bayesian inference) with such discrete mixture priors is computationally difficult. Various approximate inference methods have been proposed, including greedy search (see e.g., [SPZ09]) or MCMC (see e.g., [HS09]).

15.2.5 Laplace prior (Bayesian lasso)

A computationally cheap way to achieve sparsity is to perform MAP estimation with a Laplace prior by minimizing the penalized negative log likelihood:

$$\text{PNLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) - \log p(\mathbf{w}|\lambda) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1 \quad (15.84)$$

where $\|\mathbf{w}\|_1 \triangleq \sum_{d=1}^D |w_d|$ is the ℓ_1 norm of \mathbf{w} . This method is called **lasso**, which stands for “least absolute shrinkage and selection operator” [Tib96]. See Section 11.4 of the prequel to this book, [Mur22], for details.

In this section, we discuss posterior inference with this prior; this is known as the **Bayesian lasso** [PC08]. In particular, we assume the following prior:

$$p(\mathbf{w}|\sigma^2) = \prod_j \frac{\lambda}{2\sqrt{\sigma^2}} e^{-\lambda|w_j|/\sqrt{\sigma^2}} \quad (15.85)$$

¹ (Note that conditioning the prior on σ^2 is important to ensure that the full posterior is unimodal.)
² To simplify inference, we will represent the Laplace prior as a Gaussian scale mixture, which we
³ discussed in Section 29.2.3.2. In particular, one can show that the Laplace distribution is an infinite
⁴ weighted sum of Gaussians, where the precision comes from a Gamma distribution:
⁵

$$\text{Lap}(w|0, \lambda) = \int \mathcal{N}(w|0, \tau^2) \text{Ga}(\tau^2|1, \frac{\lambda^2}{2}) d\tau^2 \quad (15.86)$$

⁶ We can therefore represent the Bayesian lasso model as a hierarchical latent variable model, as shown
⁷ in Figure 15.3a. The corresponding joint distribution has the following form:
⁸

$$\text{p}(\mathbf{y}, \mathbf{w}, \boldsymbol{\tau}, \sigma^2 | \mathbf{X}) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \left[\prod_j \mathcal{N}(w_j | 0, \sigma^2 \tau_j^2) \text{Ga}(\tau_j^2 | 1, \lambda^2 / 2) \right] p(\sigma^2) \quad (15.87)$$

⁹ We can also create a GSM to match the **group lasso** prior, which sets multiple coefficients to
¹⁰ zero at the same time:
¹¹

$$\mathbf{w}_g | \sigma^2, \tau_g^2 \sim \mathcal{N}(\mathbf{0}, \sigma^2 \tau_g^2 \mathbf{I}_{d_g}) \quad (15.88)$$

$$\tau_g^2 \sim \text{Ga}(\frac{d_g + 1}{2}, \frac{\lambda^2}{2}) \quad (15.89)$$

¹² where d_g is the size of group g . So we see that there is one variance term per group, each of which
¹³ comes from a Gamma prior, whose shape parameter depends on the group size, and whose rate
¹⁴ parameter is controlled by γ .

¹⁵ Figure 15.3b gives an example, where we have 2 groups, one of size 2 and one of size 3. This picture
¹⁶ makes it clearer why there should be a grouping effect. For example, suppose $w_{1,1}$ is small; then τ_1^2
¹⁷ will be estimated to be small, which will force $w_{1,2}$ to be small, due to shrinkage (c.f., Section 3.5).
¹⁸ Conversely, suppose $w_{1,1}$ is large; then τ_1^2 will be estimated to be large, which will allow $w_{1,2}$ to be
¹⁹ become large as well.

²⁰ Given these hierarchical models, we can easily derive a Gibbs sampling algorithm (Section 12.3) to
²¹ sample from the posterior (see e.g., [PC08]). Unfortunately, these posterior samples are not sparse,
²² even though the MAP estimate is sparse. This is because the prior puts infinitesimal probability on
²³ the event that each coefficient is zero.
²⁴

²⁵ 15.2.6 Horseshoe prior

²⁶ The Laplace prior is not suitable for sparse Bayesian models, because posterior samples are not
²⁷ sparse. The spike and slab prior does not have this problem but is often too slow to use (although see
²⁸ [BRG20]). Fortunately, it is possible to devise continuous priors (without discrete latent variables)
²⁹ that are both sparse and computationally efficient. One popular prior of this type is the **horseshoe**
³⁰ prior [CPS10], so-named because of the shape of its density function.
³¹

³² In the horseshoe prior, instead of using a Laplace prior for each weight, we use the following
³³ Gaussian scale mixture:
³⁴

$$w_j \sim \mathcal{N}(0, \lambda_j^2 \tau^2) \quad (15.90)$$

$$\lambda_j \sim \mathcal{C}_+(0, 1) \quad (15.91)$$

$$\tau^2 \sim \mathcal{C}_+(0, 1) \quad (15.92)$$

³⁵

where $\mathcal{C}_+(0, 1)$ is the half-Cauchy distribution (Section 2.2.4), λ_j is a local shrinkage factor, and τ^2 is a global shrinkage factor. The Cauchy distribution has very fat tails, so λ_j is likely to be either 0 or very far from 0, which emulates the spike and slab prior, but in a continuous way. For more details, see e.g., [Bha+19].

15.2.7 Automatic relevancy determination

An alternative to using posterior inference with a sparsity promoting prior is to use posterior inference with a Gaussian prior, $w_j \sim \mathcal{N}(0, 1/\alpha_j)$, but where we use empirical Bayes to optimize the precisions α_j . That is, we first compute $\hat{\boldsymbol{\alpha}} = \text{argmax}_{\boldsymbol{\alpha}} p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha})$, and then compute $\hat{\mathbf{w}} = \text{argmax}_{\mathbf{w}} \mathcal{N}(\mathbf{w}|\mathbf{0}, \hat{\boldsymbol{\alpha}}^{-1})$. Perhaps surprisingly, we will see that this results in a sparse estimate, for reasons we explain in Section 15.2.7.2.

This technique is known as **sparse Bayesian learning** [Tip01] or **automatic relevancy determination (ARD)** [Mac95; Nea96]. It was originally developed for neural networks (where sparsity is applied to the first layer weights), but here we apply it to linear models.

15.2.7.1 ARD for linear models

In this section, we explain ARD in more detail, by applying it to linear regression. The likelihood is $p(y|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(y|\mathbf{w}^\top \mathbf{x}, 1/\beta)$, where $\beta = 1/\sigma^2$. The prior is $p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{A}^{-1})$, where $\mathbf{A} = \text{diag}(\boldsymbol{\alpha})$. The marginal likelihood can be computed analytically (using Equation (2.62)) as follows:

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, (1/\beta)\mathbf{I}_N) \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{A}^{-1}) d\mathbf{w} \quad (15.93)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top) \quad (15.94)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\alpha) \quad (15.95)$$

where $\mathbf{C}_\alpha \triangleq \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top$. This is very similar to the marginal likelihood under the spike-and-slab prior (Section 15.2.4), which is given by

$$p(\mathbf{y}|\mathbf{X}, \mathbf{s}, \sigma_w^2, \sigma_y^2) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}_s \mathbf{w}_s, \sigma_y^2 \mathbf{I}) \mathcal{N}(\mathbf{w}_s|\mathbf{0}_s, \sigma_w^2 \mathbf{I}) d\mathbf{w}_s = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_s) \quad (15.96)$$

where $\mathbf{C}_s = \sigma_y^2 \mathbf{I}_N + \sigma_w^2 \mathbf{X}_s \mathbf{X}_s^\top$. (Here \mathbf{X}_s refers to the design matrix where we select only the columns of \mathbf{X} where $s_d = 1$.) The difference is that we have replaced the binary $s_j \in \{0, 1\}$ variables with continuous $\alpha_j \in \mathbb{R}^+$, which makes the optimization problem easier.

The objective is the log marginal likelihood, given by

$$\ell(\boldsymbol{\alpha}, \beta) = -\frac{1}{2} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \log |\mathbf{C}_\alpha| + \mathbf{y}^\top \mathbf{C}_\alpha^{-1} \mathbf{y} \quad (15.97)$$

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$, some of which we discuss in Section 15.2.7.3.

ARD can be used as an alternative to ℓ_1 regularization. Although the ARD objective is not convex, it tends to give much sparser results [WW12]. In addition, it can be shown [WRN10] that the ARD objective has many fewer local optima than the ℓ_0 -regularized objective, and hence is much easier to optimize.

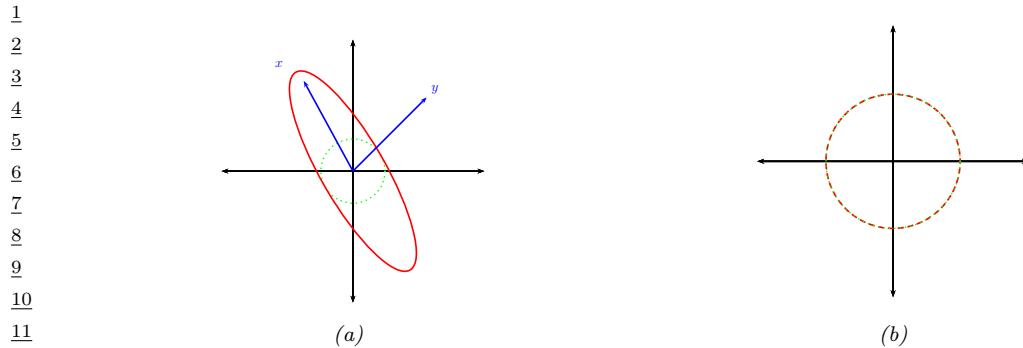


Figure 15.4: Illustration of why ARD results in sparsity. The vector of inputs \mathbf{x} does not point towards the vector of outputs \mathbf{y} , so the feature should be removed. (a) For finite α , the probability density is spread in directions away from \mathbf{y} . (b) When $\alpha = \infty$, the probability density at \mathbf{y} is maximized. Adapted from Figure 8 of [Tip01].

15.2.7.2 Why does ARD result in a sparse solution?

Once we have estimated $\boldsymbol{\alpha}$ and β , we can compute the posterior over the parameters using Bayes rule for Gaussians, to get $p(\mathbf{w}|\mathcal{D}, \hat{\boldsymbol{\alpha}}, \hat{\beta}) = \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \hat{\Sigma})$, where $\hat{\Sigma}^{-1} = \hat{\beta}\mathbf{X}^T\mathbf{X} + \mathbf{A}$ and $\hat{\mathbf{w}} = \hat{\beta}\hat{\Sigma}\mathbf{X}^T\mathbf{y}$. If we have $\hat{\alpha}_d \approx \infty$, then $\hat{w}_d \approx 0$, so the solution vector will be sparse.

We now give an intuitive argument, based on [Tip01], about when such a sparse solution may be optimal. We shall assume $\beta = 1/\sigma^2$ is fixed for simplicity. Consider a 1d linear regression with 2 training examples, so $\mathbf{X} = \mathbf{x} = (x_1, x_2)$, and $\mathbf{y} = (y_1, y_2)$. We can plot \mathbf{x} and \mathbf{y} as vectors in the plane, as shown in Figure 15.4. Suppose the feature is irrelevant for predicting the response, so \mathbf{x} points in a nearly orthogonal direction to \mathbf{y} . Let us see what happens to the marginal likelihood as we change α . The marginal likelihood is given by $p(\mathbf{y}|\mathbf{x}, \alpha, \beta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\alpha)$, where $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I} + \frac{1}{\alpha}\mathbf{x}\mathbf{x}^T$. If α is finite, the posterior will be elongated along the direction of \mathbf{x} , as in Figure 15.4(a). However, if $\alpha = \infty$, we have $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I}$, which is spherical, as in Figure 15.4(b). If $|\mathbf{C}_\alpha|$ is held constant, the latter assigns higher probability density to the observed response vector \mathbf{y} , so this is the preferred solution. In other words, the marginal likelihood “punishes” solutions where α_d is small but $\mathbf{X}_{:,d}$ is irrelevant, since these waste probability mass. It is more parsimonious (from the point of view of Bayesian Occam’s razor) to eliminate redundant dimensions.

Another way to understand the sparsity properties of ARD is as approximate inference in a hierarchical Bayesian model [BT00]. In particular, suppose we put a conjugate prior on each precision, $\alpha_d \sim \text{Ga}(a, b)$, and on the observation precision, $\beta \sim \text{Ga}(c, d)$. Since exact inference with a Student prior is intractable, we can use variational Bayes (Section 10.2.3), with a factored posterior approximation of the form

$$q(\mathbf{w}, \boldsymbol{\alpha}) = q(\mathbf{w})q(\boldsymbol{\alpha}) \approx \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \prod_d \text{Ga}(\alpha_d | \hat{\alpha}_d, \hat{b}_d) \quad (15.98)$$

ARD approximates $q(\boldsymbol{\alpha})$ by a point estimate. However, in VB, we integrate out $\boldsymbol{\alpha}$; the resulting

1 posterior marginal $q(\mathbf{w})$ on the weights is given by
2

$$\frac{4}{5} p(\mathbf{w}|\mathcal{D}) = \int \mathcal{N}(\mathbf{w}|\mathbf{0}, \text{diag}(\boldsymbol{\alpha})^{-1}) \prod_d \text{Ga}(\alpha_d | \hat{a}_d, \hat{b}) d\boldsymbol{\alpha} \quad (15.99)$$

7 This is a Gaussian scale mixture, and can be shown to be the same as a multivariate Student
8 distribution (see Section 29.2.3.1), with non-diagonal covariance. Note that the Student has a large
9 spike at 0, which intuitively explains why the posterior mean (which, for a Student distribution, is
10 equal to the posterior mode) is sparse.

11 Finally, we can also view ARD as a MAP estimation problem with a **non-factorial prior** [WN07].
12 Intuitively, the dependence between the w_j parameters arises, despite the use of a diagonal Gaussian
13 prior, because the prior precision α_j is estimated based after marginalizing out all \mathbf{w} , and hence
14 depends on all the features. Interestingly, [WRN10] prove that MAP estimation with non-factorial
15 priors is strictly better than MAP estimation with any possible factorial prior in the following
16 sense: the non-factorial objective always has fewer local minima than factorial objectives, while still
17 satisfying the property that the global optimum of the non-factorial objective corresponds to the
18 global optimum of the ℓ_0 objective — a property that ℓ_1 regularization, which has no local minima,
19 does not enjoy.

21 15.2.7.3 Algorithms for ARD

23 There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$. One approach is to use EM, in which we compute
24 $p(\mathbf{w}|\mathcal{D}, \boldsymbol{\alpha})$ in the E step and then maximize $\boldsymbol{\alpha}$ in the M step. In variational Bayes, we infer both \mathbf{w}
25 and $\boldsymbol{\alpha}$ (see [Dru08] for details). In [WN10], they present a method based on iteratively reweighted ℓ_1
26 estimation.

27 Recently, [HXW17] showed that the nested iterative computations performed these methods can
28 emulated by a recurrent neural network (Section 16.3.3). Furthermore, by training this model, it is
29 possible to achieve much faster convergence than manually designed optimization algorithms.
30

31 15.2.7.4 Relevance vector machines

33 Suppose we create a linear regression model of the form $p(y|\mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^\top \phi(\mathbf{x}), \sigma^2)$, where
34 $\phi(\mathbf{x}) = [\mathcal{K}(\mathbf{x}, \mathbf{x}_1), \dots, \mathcal{K}(\mathbf{x}, \mathbf{x}_N)]$, where $\mathcal{K}()$ is a kernel function (Section 18.2) and $\mathbf{x}_1, \dots, \mathbf{x}_N$ are
35 the N training points. This is called **kernel basis function expansion**, and transforms the input
36 from $\mathbf{x} \in \mathcal{X}$ to $\phi(\mathbf{x}) \in \mathbb{R}^N$. Obviously this model has $O(N)$ parameters, and hence is nonparametric.
37 However, we can use ARD to select a small subset of the exemplars. This technique is called the
38 relevance vector machine (RVM) [Tip01; TF03].
39

40 15.3 Logistic regression

43 **Logistic regression** is a very widely used discriminative classification model that maps input
44 vectors $\mathbf{x} \in \mathbb{R}^D$ to a distribution over class labels, $y \in \{1, \dots, C\}$. If $C = 2$, this is known as
45 **binary logistic regression**, and if $C > 2$, it is known as **multinomial logistic regression**, or
46 alternatively, **multiclass logistic regression**.

1 **15.3.1 Binary logistic regression**

3 In the binary case, where $y \in \{0, 1\}$, the model has the following form

5 $p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Ber}(y|\sigma(\mathbf{w}^\top \mathbf{x} + b))$ (15.100)

7 where \mathbf{w} are the weights, b is the bias (offset), and σ is the **sigmoid** or **logistic** function, defined by

9 $\sigma(a) \triangleq \frac{1}{1 + e^{-a}}$ (15.101)

11 Let $\eta_n = \mathbf{w}^\top \mathbf{x}_n + b$ be the **logits** for example n , and $\mu_n = \sigma(\eta_n) = p(y = 1|\mathbf{x}_n)$ be the mean of
12 the output. Then we can write the log likelihood as the negative cross entropy:

14

15 $\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^N \mu_n^{y_n} (1 - \mu_n)^{1-y_n} = \sum_{n=1}^N y_n \log \mu_n + (1 - y_n) \log(1 - \mu_n)$ (15.102)

17 We can expand this equation into a more explicit form (that is commonly seen in implementations)
18 by performing some simple algebra. First note that

20

21 $\mu_n = \frac{1}{1 + e^{-\eta_n}} = \frac{e^{\eta_n}}{1 + e^{\eta_n}}, 1 - \mu_n = 1 - \frac{e^{\eta_n}}{1 + e^{\eta_n}} = \frac{1}{1 + e^{\eta_n}}$ (15.103)

23 Hence

24

25 $\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{n=1}^N y_n [\log e^{\eta_n} - \log(1 + e^{\eta_n})] + (1 - y_n) [\log 1 - \log(1 + e^{\eta_n})]$ (15.104)

27

28 $= \sum_{n=1}^N y_n [\eta_n - \log(1 + e^{\eta_n})] + (1 - y_n) [-\log(1 + e^{\eta_n})]$ (15.105)

30

31 $= \sum_{n=1}^N y_n \eta_n - \sum_{n=1}^N \log(1 + e^{\eta_n})$ (15.106)

34 Note that the $\log(1 + e^a)$ function is often implemented using `np.log1p(np.exp(a))`.

36 **15.3.2 Multinomial logistic regression**

38 **Multinomial logistic regression** is a discriminative classification model of the following form:

40 $p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Cat}(y|\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}))$ (15.107)

41 where $\mathbf{x} \in \mathbb{R}^D$ is the input vector, $y \in \{1, \dots, C\}$ is the class label, \mathbf{W} is a $C \times D$ weight matrix, \mathbf{b}
42 is C -dimensional bias vector, and $\sigma()$ is the **softmax function**, defined as

44

45 $\sigma(\mathbf{a}) \triangleq \left[\frac{e^{a_1}}{\sum_{c'=1}^C e^{a_{c'}}}, \dots, \frac{e^{a_C}}{\sum_{c'=1}^C e^{a_{c'}}} \right]$ (15.108)

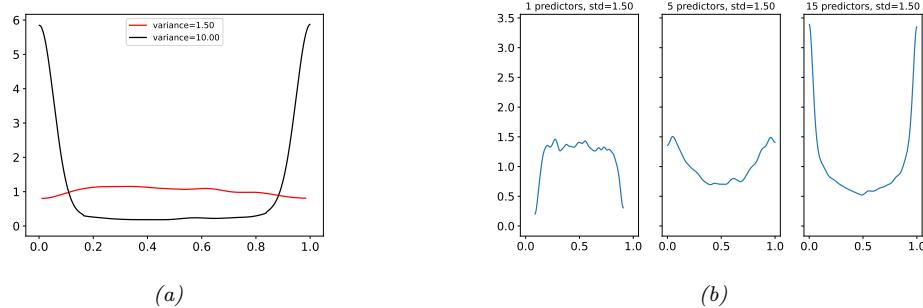


Figure 15.5: (a) Prior on logistic regression output when using $\mathcal{N}(0, \omega)$ prior for the offset term, for $\omega = 10$ or $\omega = 1.5$. Adapted from Figure 11.3 of [McE20]. Generated by `logreg_prior_offset.py`. (b) Distribution over the fraction of 1s we expect to see when using binary logistic regression applied to random binary feature vectors of increasing dimensionality. We use a $\mathcal{N}(0, 1.5)$ prior on the regression coefficients. Adapted from Figure 3 of [Gel+20]. Generated by `logreg_prior.py`.

If we define the logits as $\eta_n = \mathbf{W}\mathbf{x}_n + \mathbf{b}$, the probabilities as $\mu_n = \sigma(\eta_n)$, and let \mathbf{y}_n be the one-hot encoding of the label y_n , then the log likelihood can be written as the negative cross entropy:

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^N \prod_{c=1}^C \mu_{nc}^{y_{nc}} = \sum_{n=1}^N \sum_{c=1}^C y_{nc} \log \mu_{nc} \quad (15.109)$$

15.3.3 Priors

As with linear regression, it is standard to use Gaussian priors for the weights in a logistic regression model. It is natural to set the prior mean to 0, to reflect the fact that the output could either increase or decrease in probability depending on the input. But how do we set the prior variance? It is tempting to use a large value, to approximate a uniform distribution, but this is a bad idea. To see why, consider a binary logistic regression model with just an offset term and no features:

$$p(y|\boldsymbol{\theta}) = \text{Ber}(y|\sigma(\alpha)) \quad (15.110)$$

$$p(\alpha) = \mathcal{N}(\alpha|0, \omega) \quad (15.111)$$

If we set the prior to the large value of $\omega = 10$, the implied prior for y is an extreme distribution, with most of its density near 0 or 1, as shown in Figure 15.5a. By contrast, if we use the smaller value of $\omega = 1.5$, we get a flatter distribution, as shown.

If we have input features, the problem gets a little trickier, since the magnitude of the logits will now depend on the number and distribution of the input variables. For example, suppose we generate N random binary vectors \mathbf{x}_n , each of dimension D , where $x_{nd} \sim \text{Ber}(p)$, where $p = 0.8$. We then compute $p(y_n = 1|\mathbf{x}_n) = \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)$, where $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{0}, 1.5\mathbf{I})$. We sample S values of $\boldsymbol{\beta}$, and for each one, we sample a vector of labels, $\mathbf{y}_{1:N,s}$ from the above distribution. We then compute the fraction of positive labels, $f_s = \frac{1}{N} \sum_{n=1}^N \mathbb{I}(y_{n,s} = 1)$. We plot the distribution of $\{f_s\}$ as a function of D in

¹ Figure 15.5b. We see that the induced prior is initially flat, but eventually becomes skewed towards
² the extreme values of 0 and 1. To avoid this, we should standardize the inputs, and scale the variance
³ of the prior by $1/\sqrt{D}$. We can also use a heavier tailed distribution, such as a Cauchy or Student
⁴ [Gel+08; GLM15].
⁵

⁶

⁷ 15.3.4 Posteriors

⁸

⁹ Unfortunately, there is no tractable prior that is conjugate to the logistic likelihood. Hence we cannot
¹⁰ compute the posterior analytically, unlike with linear regression, even if we use a Gaussian prior.
¹¹ (This mirrors the case with MLE, where we have a closed form solution for linear regression, but not
¹² for logistic regression.) Fortunately, there are a range of approximate inference methods we can use,
¹³ as we discuss in the sections below.

¹⁴

¹⁵ 15.3.5 Laplace approximation

¹⁶

¹⁷ As we discuss in Section 7.4.3, the Laplace approximation approximates the posterior using a Gaussian.
¹⁸ The mean of the Gaussian is equal to the MAP estimate $\hat{\mathbf{w}}$, and the covariance is equal to the inverse
¹⁹ Hessian \mathbf{H} computed at the MAP estimate, i.e., $p(\mathbf{w}|\mathcal{D}) \approx \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \mathbf{H})$. We can find the mode using
²⁰ a standard optimization method, and we can then compute the Hessian at the mode analytically or
²¹ using automatic differentiation.

²²

²³ As an example, consider the binary data illustrated in Figure 15.6(a). There are many parameter
²⁴ settings that correspond to lines that perfectly separate the training data; we show 4 example lines.
²⁵ For each decision boundary in Figure 15.6(a), we plot the corresponding parameter vector as point
²⁶ in the log likelihood surface in Figure 15.6(b). These parameters values are $\mathbf{w}_1 = (3, 1)$, $\mathbf{w}_2 = (4, 2)$,
²⁷ $\mathbf{w}_3 = (5, 3)$, and $\mathbf{w}_4 = (7, 3)$. These points all approximately satisfy $\mathbf{w}_i(1)/\mathbf{w}_i(2) \approx \hat{\mathbf{w}}_{\text{mle}}(1)/\hat{\mathbf{w}}_{\text{mle}}(2)$,
²⁸ and hence are close to the orientation of the maximum likelihood decision boundary. The points
²⁹ are ordered by increasing weight norm (3.16, 4.47, 5.83, and 7.62). The unconstrained MLE has
³⁰ $\|\mathbf{w}\| = \infty$, so is infinitely far to the top right.

³¹

³² To ensure a unique solution, we use a (spherical) Gaussian prior centered at the origin, $\mathcal{N}(\mathbf{w}|\mathbf{0}, \sigma^2 \mathbf{I})$.
³³ The value of σ^2 controls the strength of the prior. If we set $\sigma^2 = \infty$, we force the MAP estimate
³⁴ to be $\mathbf{w} = \mathbf{0}$; this will result in maximally uncertain predictions, since all points \mathbf{x} will produce a
³⁵ predictive distribution of the form $p(y=1|\mathbf{x}) = 0.5$. If we set $\sigma^2 = 0$, the MAP estimate becomes
³⁶ the MLE, resulting in minimally uncertain predictions. (In particular, all positively labeled points
³⁷ will have $p(y=1|\mathbf{x}) = 1.0$, and all negatively labeled points will have $p(y=1|\mathbf{x}) = 0.0$, since the
³⁸ data is separable.) As a compromise (to make a nice illustration), we pick the value $\sigma^2 = 100$.

³⁹

⁴⁰ Multiplying this prior by the likelihood results in the unnormalized posterior shown in Figure 15.6(c).
⁴¹ The MAP estimate is shown by the blue dot. The Laplace approximation to this posterior is shown
⁴² in Figure 15.6(d). We see that it gets the mode correct (by construction), but the shape of the
⁴³ posterior is somewhat distorted. (The southwest-northeast orientation captures uncertainty about the
⁴⁴ magnitude of \mathbf{w} , and the southeast-northwest orientation captures uncertainty about the orientation
⁴⁵ of the decision boundary.)

⁴⁶

⁴⁷ Next we need to convert the posterior over the parameters into a posterior over predictions, as
⁴⁸ follows:

$$\frac{45}{46} p(y|\mathbf{x}, \mathcal{D}) = \int p(y|\mathbf{x}, \mathbf{w}) p(\mathbf{w}|\mathcal{D}) d\mathbf{w} \quad (15.112)$$

⁴⁷

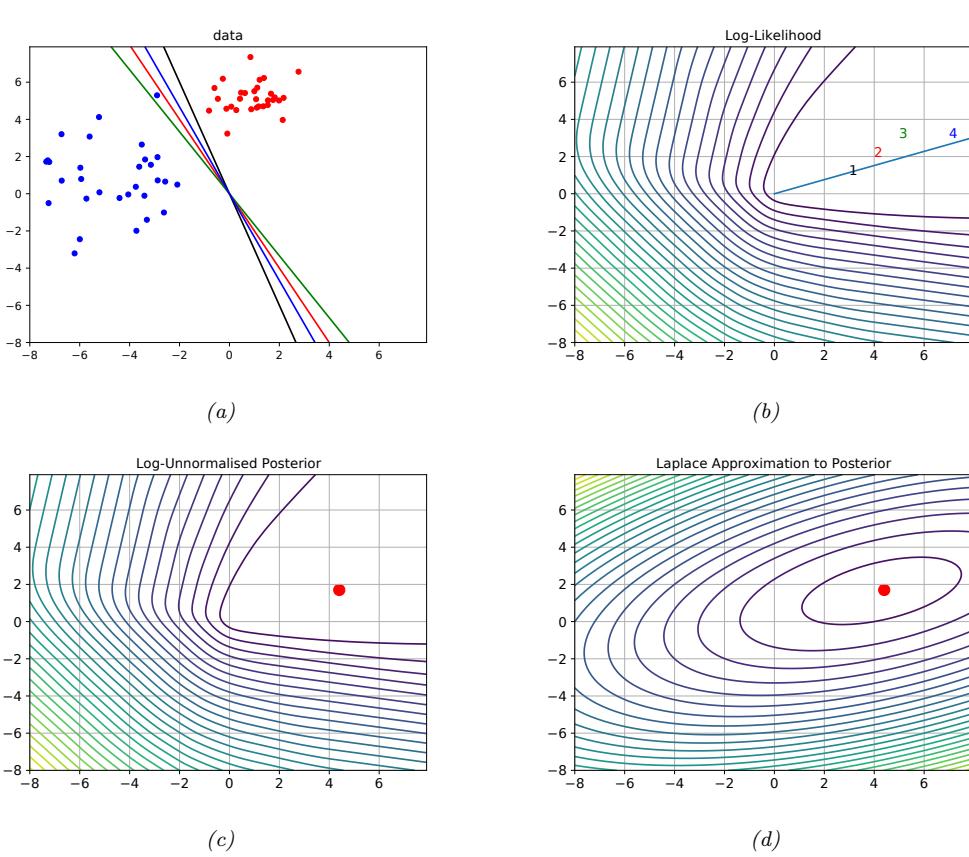


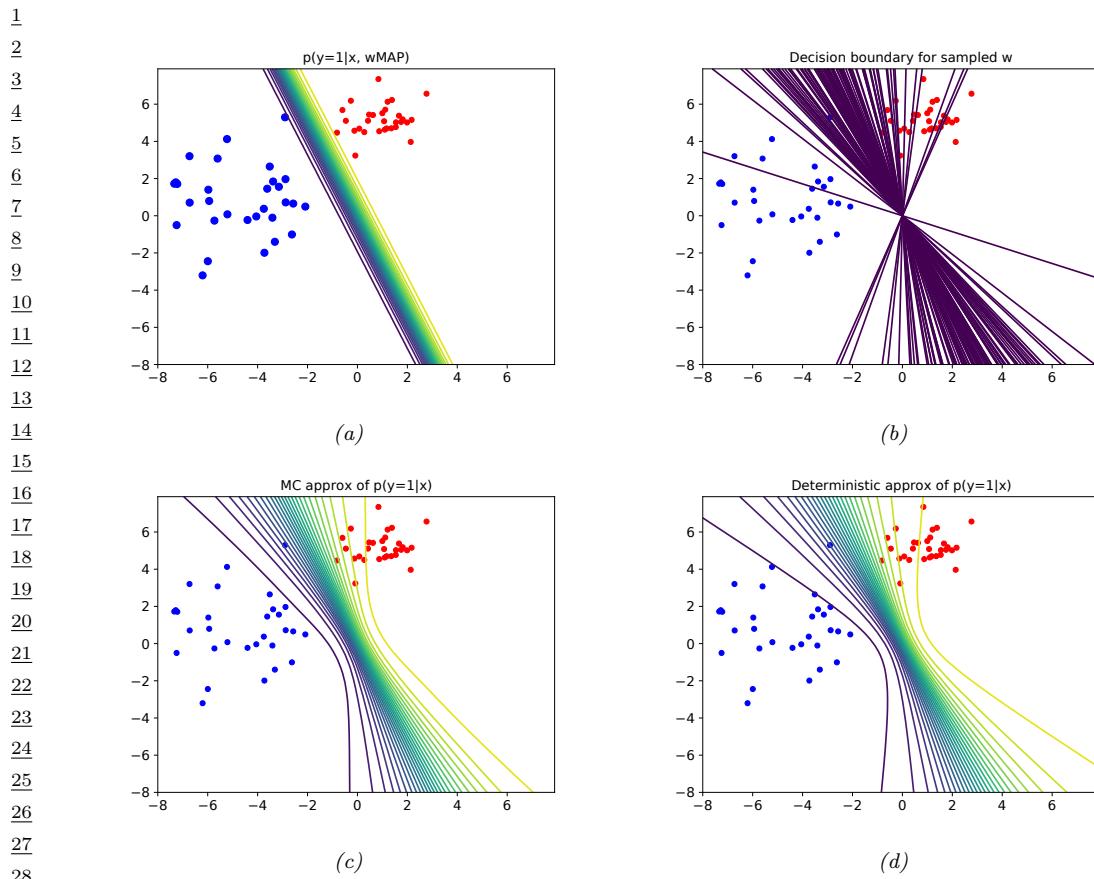
Figure 15.6: (a) Illustration of the data and some decision boundaries, where colors correspond to the points in the (b) panel. (b) Log-likelihood for a logistic regression model. The line is drawn from the origin in the direction of the MLE (which is at infinity). The numbers correspond to 4 points in parameter space, corresponding to the lines in (a). (c) Unnormalized log posterior (assuming vague spherical prior). (d) Laplace approximation to posterior. Adapted from a figure by Mark Girolami. Generated by [logreg_laplace_demo.py](#).

The simplest way to evaluate this integral is to use a Monte Carlo approximation:

$$p(y=1|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \sigma(\mathbf{w}_s^\top \mathbf{x}) \quad (15.113)$$

where $\mathbf{w}_s \sim p(\mathbf{w}|\mathcal{D})$.

Alternatively, we can use the deterministic **probit approximation** first suggested in [SL90]. If we



29 Figure 15.7: Posterior predictive distribution for a logistic regression model in 2d. (a): contours of $p(y = 30 1|\mathbf{x}, \hat{\mathbf{w}}_{map})$. (b): samples from the posterior predictive distribution. (c): Averaging over these samples. 31 (d): moderated output (probit approximation). Adapted from a figure by Mark Girolami. Generated by 32 [logreg_laplace_demo.py](#).

33
34
35 define $a = \mathbf{x}^\top \mathbf{w}$, and $p(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then we can write this approximation as
36

$$37 \quad p(y = 1|\mathbf{x}, \mathcal{D}) \approx \sigma(\kappa(v)m) \quad (15.114)$$

$$38 \quad \kappa(v) \triangleq (1 + \pi v / 8)^{-\frac{1}{2}} \quad (15.115)$$

$$39 \quad v = \mathbb{V}[a] = \mathbb{V}[\mathbf{x}^\top \mathbf{w}] = \mathbf{x}^\top \boldsymbol{\Sigma} \mathbf{x} \quad (15.116)$$

$$40 \quad m = \mathbb{E}[a] = \mathbf{x}^\top \boldsymbol{\mu} \quad (15.117)$$

43 In Figure 15.7, we show contours of the posterior predictive distribution. Figure 15.7(a) shows the
44 plugin approximation using the MAP estimate. We see that there is no uncertainty about the decision
45 boundary, even though we are generating probabilistic predictions over the labels. Figure 15.7(b)
46 shows what happens when we plug in samples from the Gaussian posterior. Now we see that there is
47

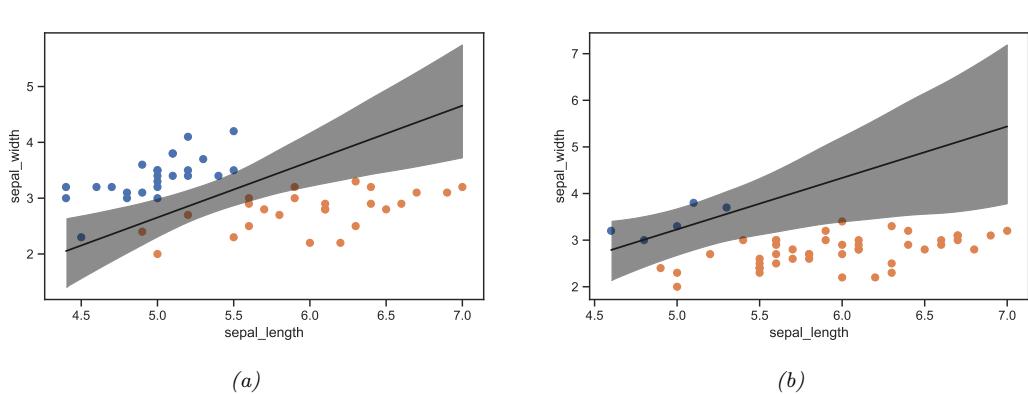


Figure 15.8: Illustration of the posterior over the decision boundary for classifying iris flowers (setosa vs versicolor) using 2 input features. (a) 25 examples per class. Adapted from Figure 4.5 of [Mar18]. (b) 5 examples of class 0, 45 examples of class 1. Adapted from Figure 4.8 of [Mar18]. Generated by [logreg_iris_bayes_2d_pymc3.py](#).

considerable uncertainty about the orientation of the “best” decision boundary. Figure 15.7(c) shows the average of these samples. By averaging over multiple predictions, we see that the uncertainty in the decision boundary “splays out” as we move further from the training data. Figure 15.7(d) shows that the probit approximation gives very similar results to the Monte Carlo approximation.

15.3.6 MCMC inference

Markov chain Monte Carlo, or MCMC, is often considered the “gold standard” for approximate inference, since it makes no explicit assumptions about the form of the posterior. Instead, it just approximates it non-parametrically using a set of S samples:

$$q(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s) \quad (15.118)$$

where $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$ are samples from the posterior.

To efficiently compute these samples, we can use the method of Hamiltonian Monte Carlo or HMC, which we describe in Section 12.5. This relies on our ability to compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\mathcal{D}, \boldsymbol{\theta})$, which we can compute using automatic differentiation.

Let us apply HMC to a 2-dimensional, 2-class version of the iris classification problem, where we just use two input features, sepal length and sepal width, and two classes, Virginica and Non-Virginica. The decision boundary is the set of points (x_1^*, x_2^*) such that $\sigma(b + w_1 x_1^* + w_2 x_2^*) = 0.5$. Such points must lie on the following line:

$$x_2^* = -\frac{b}{w_2} + \left(-\frac{w_1}{w_2} x_1^* \right) \quad (15.119)$$

We can therefore compute an MC approximation to the posterior over decision boundaries by sampling the parameters from the posterior, $(w_1, w_2, b) \sim p(\boldsymbol{\theta}|\mathcal{D})$, and plugging them into the above equation,

1 to get $p(x_1^*, x_2^* | \mathcal{D})$. The results of this method (using a vague Gaussian prior for the parameters)
2 are shown in Figure 15.8a. The solid line is the posterior mean, and the shaded interval is a 95%
3 credible interval. As before, we see that the uncertainty about the location of the boundary is higher
4 as we move away from the training data.
5

6 In Figure 15.8b, we show what happens to the decision boundary when we have unbalanced classes.
7 We notice two things. First, the posterior uncertainty increases, because we have less data from the
8 blue class. Second, we see that the posterior mean of the decision boundary shifts towards the class
9 with less data. This follows from linear discriminant analysis, where one can show that changing
10 the class prior changes the location of the decision boundary, so that more of the input space gets
11 mapped to the class which is higher a priori. (See [Mur22, Sec 9.2] for details.)
12

13 15.3.7 Variational inference

14 As we discuss in Section 10.1, variational inference converts approximate inference into an optimization
15 problem. It does this by choosing an approximate distribution $q(\mathbf{w}; \psi)$ and optimising the variational
16 parameters ψ to maximize the evidence lower bound (ELBO). This has the effect of making
17 $q(\mathbf{w}; \psi) \approx p(\mathbf{w} | \mathcal{D})$ in the sense that the KL divergence is small. There are several ways to tackle
18 this: use a stochastic estimate of the ELBO (see Section 10.3.3), use the conditionally conjugate VI
19 method of Section 10.3.8.2, or use a “local” VI method that creates a quadratic lower bound to the
20 logistic function (see supplementary material).
21

22

23 15.4 Probit regression

24 In this section, we discuss **probit regression**, which is similar to binary logistic regression except
25 it uses $\mu_n = \Phi(a_n)$ instead of $\mu_n = \sigma(a_n)$ as the mean function, where Φ is the cdf of the standard
26 normal, and $a_n = \mathbf{w}^\top \mathbf{x}_n$. The corresponding link function is therefore $a_n = \ell(\mu_n) = \Phi^{-1}(\mu_n)$; the
27 inverse of the Gaussian cdf is known as the **probit function**.
28

29 The Gaussian cdf Φ is very similar to the logistic function, as shown in Figure 15.9. Thus probit
30 regression and “regular” logistic regression behave very similarly. However, probit regression has some
31 advantages. In particular, it has a simple interpretation as a latent variable model (see Section 15.4.1),
32 which arises from the field of **choice theory** as studied in economics (see e.g., [Koo03]). This also
33 simplifies the task of Bayesian parameter inference.
34

35

36

37 15.4.1 Latent variable interpretation

38 We can interpret $a_n = \mathbf{w}^\top \mathbf{x}_n$ as a factor that is proportional to how likely a person is respond
39 positively (generate $y_n = 1$) given input \mathbf{x}_n . However, typically there are other unobserved factors that
40 influence someone’s response. Let us model these hidden factors by Gaussian noise, $\epsilon_n \sim \mathcal{N}(0, 1)$. Let
41 the combined preference for positive outcomes be represented by the latent variable $z_n = \mathbf{w}^\top \mathbf{x}_n + \epsilon_n$.
42 We assume that the person will pick the positive label iff this latent factor is positive rather than
43 negative, i.e.,
44

45

$$\underline{46} \quad y_n = \mathbb{I}(z_n \geq 0) \quad (15.120)$$

47

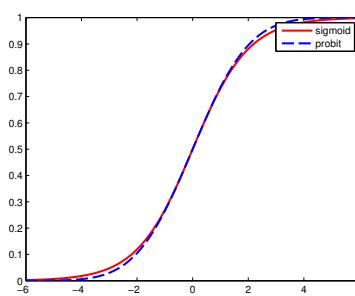


Figure 15.9: The logistic (sigmoid) function $\sigma(x)$ in solid red, with the Gaussian cdf function $\Phi(\lambda x)$ in dotted blue superimposed. Here $\lambda = \sqrt{\pi}/8$, which was chosen so that the derivatives of the two curves match at $x = 0$. Adapted from Figure 4.9 of [Bis06]. Generated by `probit_plot.py`.

When we marginalize out z_n , we recover the probit model:

$$p(y_n = 1 | \mathbf{x}_n, \mathbf{w}) = \int \mathbb{I}(z_n \geq 0) \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) dz_n \quad (15.121)$$

$$= p(\mathbf{w}^\top \mathbf{x}_n + \epsilon_n \geq 0) = p(\epsilon_n \geq -\mathbf{w}^\top \mathbf{x}_n) \quad (15.122)$$

$$= 1 - \Phi(-\mathbf{w}^\top \mathbf{x}_n) = \Phi(\mathbf{w}^\top \mathbf{x}_n) \quad (15.123)$$

Thus we can think of probit regression as a threshold function applied to noisy input.

We can interpret logistic regression in the same way. However, in that case the noise term ϵ_n comes from a **logistic distribution**, defined as follows:

$$f(y|\mu, s) \triangleq \frac{e^{-\frac{y-\mu}{s}}}{s(1 + e^{-\frac{y-\mu}{s}})^2} \quad (15.124)$$

The cdf of this distribution is given by

$$F(y|\mu, s) = \frac{1}{1 + e^{-\frac{y-\mu}{s}}} \quad (15.125)$$

It is clear that if we use logistic noise with $\mu = 0$ and $s = 1$ we recover logistic regression. However, it is computationally easier to deal with Gaussian noise, as we show below.

15.4.2 Maximum likelihood estimation

In this section, we discuss some methods for fitting probit regression using MLE.

15.4.2.1 MLE using SGD

We can find the MLE for probit regression using standard gradient methods. Let $\mu_n = \mathbf{w}^\top \mathbf{x}_n$, and let $\tilde{y}_n \in \{-1, +1\}$. Then the gradient of the log-likelihood for a single example n is given by

$$\mathbf{g}_n \triangleq \frac{d}{d\mathbf{w}} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \frac{d\mu_n}{d\mathbf{w}} \frac{d}{d\mu_n} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \mathbf{x}_n \frac{\tilde{y}_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \quad (15.126)$$

where ϕ is the standard normal pdf, and Φ is its cdf. Similarly, the Hessian for a single case is given by

$$\mathbf{H}_n = \frac{d}{d\mathbf{w}^2} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = -\mathbf{x}_n \left(\frac{\phi(\mu_n)^2}{\Phi(\tilde{y}_n \mu_n)^2} + \frac{\tilde{y}_n \mu_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \right) \mathbf{x}_n^\top \quad (15.127)$$

This can be passed to any gradient-based optimizer.

15.4.2.2 MLE using EM

We can use the latent variable interpretation of probit regression to derive an elegant EM algorithm for fitting the model. The complete data log likelihood has the following form, assuming a $\mathcal{N}(\mathbf{0}, \mathbf{V}_0)$ prior on \mathbf{w} :

$$\ell(\mathbf{z}, \mathbf{w} | \mathbf{V}_0) = \log p(\mathbf{y} | \mathbf{z}) + \log \mathcal{N}(\mathbf{z} | \mathbf{X}\mathbf{w}, \mathbf{I}) + \log \mathcal{N}(\mathbf{w} | \mathbf{0}, \mathbf{V}_0) \quad (15.128)$$

$$= \sum_n \log p(y_n | z_n) - \frac{1}{2} (\mathbf{z} - \mathbf{X}\mathbf{w})^\top (\mathbf{z} - \mathbf{X}\mathbf{w}) - \frac{1}{2} \mathbf{w}^\top \mathbf{V}_0^{-1} \mathbf{w} \quad (15.129)$$

The posterior in the E step is a **truncated Gaussian**:

$$p(z_n | y_n, \mathbf{x}_n, \mathbf{w}) = \begin{cases} \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n > 0) & \text{if } y_n = 1 \\ \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n < 0) & \text{if } y_n = 0 \end{cases} \quad (15.130)$$

In Equation (15.129), we see that \mathbf{w} only depends linearly on \mathbf{z} , so we just need to compute $\mathbb{E}[z_n | y_n, \mathbf{x}_n, \mathbf{w}]$, so we just need to compute the posterior mean. One can show that this is given by

$$\mathbb{E}[z_n | \mathbf{w}, \mathbf{x}_n] = \begin{cases} \mu_n + \frac{\phi(\mu_n)}{1 - \Phi(-\mu_n)} = \mu_n + \frac{\phi(\mu_n)}{\Phi(\mu_n)} & \text{if } y_n = 1 \\ \mu_n - \frac{\phi(\mu_n)}{\Phi(-\mu_n)} = \mu_n - \frac{\phi(\mu_n)}{1 - \Phi(\mu_n)} & \text{if } y_n = 0 \end{cases} \quad (15.131)$$

where $\mu_n = \mathbf{w}^\top \mathbf{x}_n$.

In the M step, we estimate \mathbf{w} using ridge regression, where $\boldsymbol{\mu} = \mathbb{E}[\mathbf{z}]$ is the output we are trying to predict. Specifically, we have

$$\hat{\mathbf{w}} = (\mathbf{V}_0^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{\mu} \quad (15.132)$$

The EM algorithm is simple, but can be much slower than direct gradient methods, as illustrated in Figure 15.10. This is because the posterior entropy in the E step is quite high, since we only observe that z is positive or negative, but are given no information from the likelihood about its magnitude. Using a stronger regularizer can help speed convergence, because it constrains the range of plausible z values. In addition, one can use various speedup tricks, such as data augmentation [DM01].

15.4.3 Bayesian inference

It is possible to use the latent variable formulation of probit regression in Section 15.4.2.2 to derive a simple Gibbs sampling algorithm for approximating the posterior $p(\mathbf{w} | \mathcal{D})$ (see e.g., [AC93; HH06]).

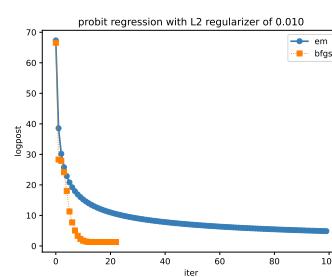


Figure 15.10: Fitting a probit regression model in 2d using a quasi-Newton method or EM. Generated by `probitRegDemo.py`.

The key idea is to use an auxiliary latent variable, which, when conditioned on, makes the whole model a conjugate linear-Gaussian model. The full conditional for the latent variables is given by

$$p(z_i|y_i, \mathbf{x}_i, \mathbf{w}) = \begin{cases} \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i > 0) & \text{if } y_i = 1 \\ \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i < 0) & \text{if } y_i = 0 \end{cases} \quad (15.133)$$

Thus the posterior is a truncated Gaussian. We can sample from a truncated Gaussian, $\mathcal{N}(z|\mu, \sigma)\mathbb{I}(a \leq z \leq b)$ in two steps: first sample $u \sim U(\Phi((a - \mu)/\sigma), \Phi((b - \mu)/\sigma))$, then set $z = \mu + \sigma\Phi^{-1}(u)$ [Rob95a].

The full conditional for the parameters is given by

$$p(\mathbf{w}|\mathcal{D}, \mathbf{z}, \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{w}_N, \mathbf{V}_N) \quad (15.134)$$

$$\mathbf{V}_N = (\mathbf{V}_0^{-1} + \mathbf{X}^T \mathbf{X})^{-1} \quad (15.135)$$

$$\mathbf{w}_N = \mathbf{V}_N(\mathbf{V}_0^{-1} \mathbf{m}_0 + \mathbf{X}^T \mathbf{z}) \quad (15.136)$$

For further details, see e.g., [AC93; FSF10]. It is also possible to use variational Bayes, which tends to be much faster (see e.g., [GR06a; FDZ19]).

15.4.4 Ordinal probit regression

One advantage of the latent variable interpretation of probit regression is that it is easy to extend to the case where the response variable is ordered in some way, such as the outputs low, medium and high. This is called **ordinal regression**. The basic idea is as follows. If there are C output values, we introduce $C + 1$ thresholds γ_j and set

$$y_n = j \quad \text{if} \quad \gamma_{j-1} < z_n \leq \gamma_j \quad (15.137)$$

where $\gamma_0 \leq \dots \leq \gamma_C$. For identifiability reasons, we set $\gamma_0 = -\infty$, $\gamma_1 = 0$ and $\gamma_C = \infty$. For example, if $C = 2$, this reduces to the standard binary probit model, whereby $z_n < 0$ produces $y_n = 0$ and $z_n \geq 0$ produces $y_n = 1$. If $C = 3$, we partition the real line into 3 intervals: $(-\infty, 0]$, $(0, \gamma_2]$, (γ_2, ∞) . We can vary the parameter γ_2 to ensure the right relative amount of probability mass falls in each interval, so as to match the empirical frequencies of each class label. See e.g., [AC93] for further details.

¹ Finding the MLEs for this model is a bit trickier than for binary probit regression, since we need
² to optimize for \mathbf{w} and γ , and the latter must obey an ordering constraint. See e.g., [KL09] for an
³ approach based on EM. It is also possible to derive a simple Gibbs sampling algorithm for this model
⁴ (see e.g., [Hof09, p216]).
⁵

⁶

⁷ 15.4.5 Multinomial probit models

⁸ Now consider the case where the response variable can take on C unordered categorical values,
⁹ $y_n \in \{1, \dots, C\}$. The **multinomial probit** model is defined as follows:
¹⁰

$$\begin{aligned} \text{11} \quad z_{nc} &= \mathbf{w}_c^\top \mathbf{x}_{nc} + \epsilon_{nc} \\ \text{12} \quad \epsilon &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned} \tag{15.138} \tag{15.139}$$

$$\begin{aligned} \text{13} \quad y_n &= \arg \max_c z_{nc} \\ \text{14} \quad & \\ \text{15} \end{aligned} \tag{15.140}$$

¹⁶ See e.g., [DE04; GR06b; Sco09; FSF10] for more details on the model and its connection to multinomial
¹⁷ logistic regression.

¹⁸ If instead of setting $y_n = \arg \max_c z_{ic}$ we use $y_{nc} = \mathbb{I}(z_{nc} > 0)$, we get a model known as
¹⁹ **multivariate probit**, which is one way to model C correlated binary outcomes (see e.g., [TMD12]).
²⁰

²¹ 15.5 Multi-level GLMs

²² Suppose we have a set of J related datasets, each of which contains a series of N_j datapoints
²³ $\mathcal{D}_j = \{(\mathbf{x}_{nj}, \mathbf{y}_{nj}) : n = 1 : N_j\}$. There are 3 main ways to fit models in such a setting: we could fit J
²⁴ separate models, $p(\mathbf{y}|\mathbf{x}; \mathcal{D}_j)$, which might result in overfitting if some \mathcal{D}_j are small; we could pool all
²⁵ the data to get $\mathcal{D} = \cup_{j=1}^J \mathcal{D}_j$ and fit a single model, $p(\mathbf{y}|\mathbf{x}; \mathcal{D})$, which might result in underfitting; or
²⁶ we can use a **hierarchical Bayesian model**, also called a **multilevel model** or **partially pooled**
²⁷ **model**, in which we assume each group has its own parameters, $\boldsymbol{\theta}_j$, but that these have something
²⁸ in common, as modeled by a shared global prior $p(\phi)$. (Note that each group could be a single
²⁹ individual.) The overall model has the form
³⁰

$$\begin{aligned} \text{31} \quad p(\phi, \boldsymbol{\theta}, \mathcal{D}) &= p(\phi) \prod_{j=1}^J p(\boldsymbol{\theta}_j | \phi) \prod_{n=1}^{N_j} p(\mathbf{y}_{nj} | \mathbf{x}_{nj}, \boldsymbol{\theta}_j) \\ \text{32} \quad & \\ \text{33} \quad & \\ \text{34} \quad & \\ \text{35} \end{aligned} \tag{15.141}$$

³⁶ If the likelihood function is a GLM, this is called a **hierarchical generalized linear model** [LN96].
³⁷

³⁸ 15.5.1 Generalized linear mixed models (GLMMs)

³⁹ If $p(\boldsymbol{\theta}_j | \phi) = \mathcal{N}(\boldsymbol{\theta}_j | \phi, \Sigma)$, and the likelihood function is a GLM, then the model can be written as
⁴⁰ follows:
⁴¹

$$\begin{aligned} \text{42} \quad p(\mathbf{y}_{nj} | \mathbf{x}_{nj}, \boldsymbol{\theta}) &= p(\mathbf{y}_{nj} | \ell(\boldsymbol{\eta}_{nj})) \\ \text{43} \quad & \\ \text{44} \quad & \end{aligned} \tag{15.142}$$

$$\begin{aligned} \text{45} \quad \boldsymbol{\eta}_{nj} &= (\phi_0 + \epsilon_{0j}) + \sum_{d=1}^D (\phi_d + \epsilon_{dj}) x_{njd} \\ \text{46} \quad & \\ \text{47} \quad & \end{aligned} \tag{15.143}$$

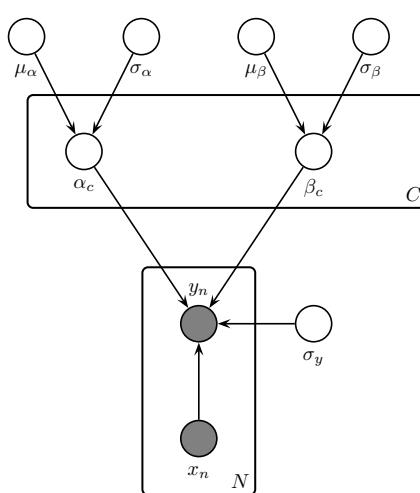


Figure 15.11: A hierarchical Bayesian linear regression model for the radon problem.

where ℓ is the link function, and $\epsilon_j \sim \mathcal{N}(\mathbf{0}, \Sigma)$. This is known as a **generalized linear mixed model** or **GLMM** or **mixed effects model**. The shared (common) parameters ϕ are called **fixed effects**, and the group-specific parameters θ_j (or equivalently ϵ_j) are called **random effects**.³ We can see that the random effects model group-specific deviations or idiosyncrasies away from the shared fixed parameters. Furthermore, we see that the random effects are correlated, which allows us to model dependencies between the observations that would not be captured by a standard GLM.

15.5.2 Model fitting

We can fit GLMMs, and hierarchical models more generally, using standard Bayesian inference methods. We can use a variety of algorithms, such as HMC (see e.g., [BG13]), variational Bayes (see e.g., [HOW11; TN13]), expectation propagation (see e.g., [KW18]), etc. In this section, we use HMC, since it is simple and efficient.⁴

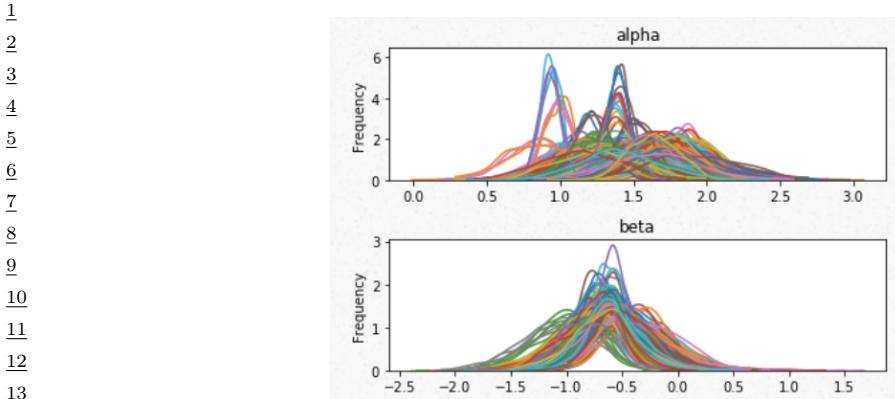
15.5.3 Example: radon regression

In this section, we give an example of a hierarchical Bayesian linear regression model. We apply it to a simplified version of the **radon** example from [Gel+14a, Sec 9.4].

Radon is known to be the highest cause of lung cancer in non-smokers, so reducing it where possible is desirable. To help with this, we fit a regression model, that predicts the (log) radon level as a function of the location of the house, as represented by a categorical feature indicating its county, and

3. Note that there are multiple definitions of the terms “fixed effects” and “random effects”, as explained in this blog post by Andrew Gelman: https://statmodeling.stat.columbia.edu/2005/01/25/why_i_dont_use/.

4. There are many standard software packages for HMC analysis of hierarchical GLMs, such as **Bambi** (<https://github.com/bambinos/bambi>), which is a Python wrapper on top of PyMC, **RStanARM** (<https://cran.r-project.org/web/packages/rstanarm/index.html>), which is an R wrapper on top of Stan, and **BRMS** (<https://cran.r-project.org/web/packages;brms/index.html>), which is another R wrapper on top of Stan, but which also needs a C++ compiler.



14 *Figure 15.12: Posterior marginals for α_c and β_c for each county in the radon model. Generated by [linreg_hierarchical_non_centered_blackjax.ipynb](#).*

15

16

17 a binary feature representing whether the house has a basement or not. We use a dataset consisting
18 of $C = 85$ counties in Minnesota; each county has between 2 and 80 measurements.
19

20 We assume the following likelihood:
21

$$\text{y}_i | \mathbf{x}_i = (c[i], f[i]), \boldsymbol{\theta} \sim \mathcal{N}(\alpha_{c[i]} + \beta_{c[i]} f[i], \sigma_y^2) \quad (15.144)$$

22 where $c[i] \in \{1, \dots, C\}$ is the county for house i , and $f[i] \in \{0, 1\}$ indicates if the floor is at level 0
23 (i.e., in the basement) or level 1 (i.e., above ground). Intuitively we expect the radon levels to be
24 lower in houses without basements, since they are more insulated from the earth where the radon
25 lives. Thus we want to compute $p(\boldsymbol{\beta}|\mathcal{D})$, so we can test this hypothesis.

26 Since some counties have very few data points, we use a hierarchical prior in which we assume
27 $\alpha_c \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2)$ and $\beta_c \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$. We use weak priors for the parameters: $\mu_\alpha \sim \mathcal{N}(0, 1)$,
28 $\mu_\beta \sim \mathcal{N}(0, 1)$, $\sigma_\alpha \sim \mathcal{C}_+(1)$, $\sigma_\beta \sim \mathcal{C}_+(1)$, $\sigma_y \sim \mathcal{C}_+(1)$. See Figure 15.11 for the graphical model.

29

30 15.5.3.1 Posterior inference

31

32 Figure 15.12 shows the posterior marginals for μ_α , μ_β , α_c and β_c . We see that μ_β is close to -0.6
33 with high probability, which confirms our suspicion that having $f = 1$ (i.e., no basement) decreases
34 the amount of radon in the house. We also see that the distribution of the α_c parameters is quite
35 variable, due to different base rates across the counties.

36 Figure 15.13 shows predictions from the hierarchical and non-hierarchical model for 3 different
37 counties. We see that the predictions from the hierarchical model are more consistent across counties,
38 and work well even if there are no examples of certain feature combinations for a given county (e.g.,
39 there are no houses without basements in the sample from Cass county). If we sample data from the
40 posterior predictive distribution, and compare it to the real data, we find that the RMSE is 0.13 for
41 the non-hierarchical model and 0.08 for the hierarchical model, indicating that the latter fits better.
42 (See [linreg_hierarchical_non_centered_blackjax.ipynb](#) for the code that reproduces these numbers.)
43

44

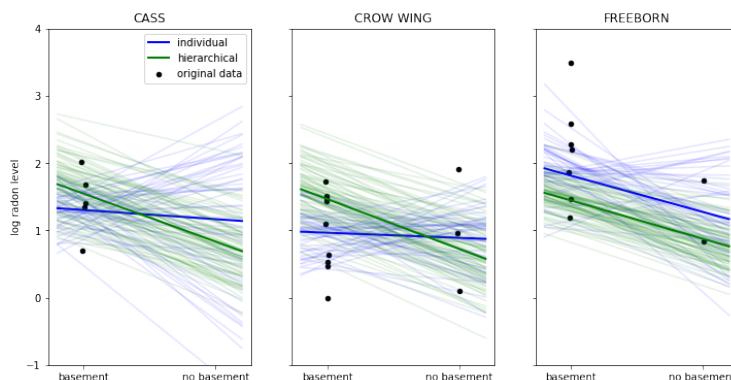


Figure 15.13: Predictions from the radon model for 3 different counties in Minnesota. Black dots are observed datapoints. Green represents results of hierarchical (shared) prior, blue represents results of non-hierarchical prior. Thick lines are the result of using the posterior mean, thin lines are the result of using posterior samples. Generated by `linreg_hierarchical_non_centered_blackjax.ipynb`.

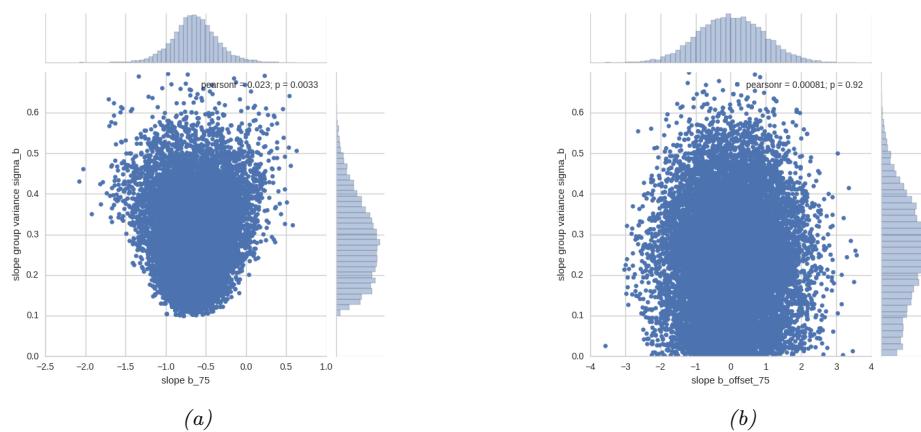


Figure 15.14: (a) Bivariate posterior $p(\beta_c, \sigma_\beta | \mathcal{D})$ for the hierarchical radon model for county $c = 75$ using centered parameterization. (b) Similar to (a) except we plot $p(\hat{\beta}_c, \sigma_\beta | \mathcal{D})$ for the non-centered parameterization. Generated by `linreg_hierarchical_non_centered_blackjax.ipynb`.

15.5.3.2 Non-centered parameterization

One problem that frequently arises in hierarchical models is that the parameters be very correlated. This can cause computational problems when performing inference.

Figure 15.14a gives an example where we plot $p(\beta_c, \sigma_\beta | \mathcal{D})$ for some specific county c . If we believe that σ_β is large, then β_c is “allowed” to vary a lot, and we get the broad distribution at the top of the figure. However, if we believe that σ_β is small, then β_c is constrained to be close to the global prior mean of μ_β , so we get the narrow distribution at the bottom of the figure. This is often called **Neal’s funnel**, after a paper by Radford Neal [Nea03]. It is difficult for many algorithms (especially

1 sampling algorithms) to explore parts of parameter space at the bottom of the funnel. This is evident
2 from the marginal posterior for σ_β shown (as a histogram) on the right hand side of the plot: we see
3 that it excludes the interval $[0, 0.1]$, thus ruling out models in which we shrink β_c all the way to 0. In
4 cases where a covariate has no useful predictive role, we would like to be able to induce sparsity, so
5 we need to overcome this problem.
6

7 A simple solution to this is to use a **non-centered parameterization** [PR03]. That is, we replace
8 $\beta_c \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$ with $\beta_c = \mu_\beta + \tilde{\beta}_c \sigma_\beta$, where $\tilde{\beta}_c \sim \mathcal{N}(0, 1)$ represents the *offset* from the global mean,
9 μ_β . The correlation between $\tilde{\beta}_c$ and σ_β is much less, as shown in Figure 15.14b. See Section 12.6.4
10 for more details.

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

16 Deep neural networks

16.1 Introduction

The term “**deep neural network**” or **DNN**, in its modern usage, refers to any kind of differentiable function that can be expressed as a **computation graph**, where the nodes are primitive operations (like matrix multiplication), and edges represent numeric data in the form of vectors, matrices, or tensors. In its simplest form, this graph can be constructed as a linear series of nodes or “**layers**”. The term “deep” refers to models with many such layers.

In Section 16.2 we discuss some of the basic building blocks (node types) that are used in the field. In Section 16.3 we give examples of common architectures which are constructed from these building blocks. In Section 6.2 we show how we can efficiently compute the gradient of functions defined on such graphs. If the function computes the scalar loss of the model’s predictions given a training set, we can pass this gradient to an optimization routine, such as those discussed in Chapter 6, in order to fit the model. Fitting such models to data is called “**deep learning**”.

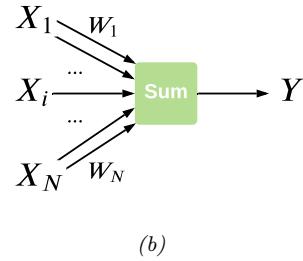
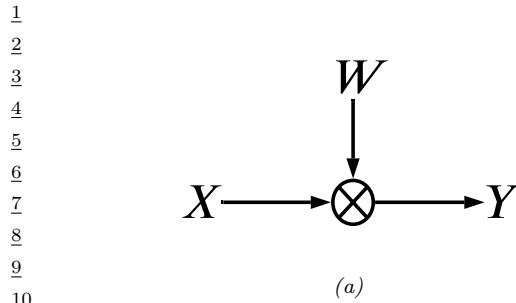
We can combine DNNs with probabilistic models in two different ways. The first is to use them to define nonlinear functions which are used inside conditional distributions. For example, we may construct a classifier using $p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\sigma(f(\mathbf{x}; \boldsymbol{\theta})))$, where $f(\mathbf{x}; \boldsymbol{\theta})$ is a neural network that maps inputs \mathbf{x} and parameters $\boldsymbol{\theta}$ to output logits. Or we may construct a joint probability distribution over multiple variables using a directed graphical model (Chapter 4) where each CPD $p(\mathbf{x}_i|\text{pa}(\mathbf{x}_i))$ is a DNN. This lets us construct expressive probability models.

The other way we can combine DNNs and probabilistic models is to use DNNs to approximate the posterior distribution, i.e., we learn a function f to compute $q(\mathbf{h}|f(\mathcal{D}; \boldsymbol{\phi}))$, where \mathbf{h} are the hidden variables (latents and/or parameters), \mathcal{D} are the observed variables (data), f is an **inference network**, and $\boldsymbol{\phi}$ are its parameters; for details, see Section 10.3.7. Note that in this latter, setting the joint model $p(\mathbf{h}, \mathcal{D})$ may be a “traditional” model without any “neural” components. For example, it could be a complex simulator. Thus the DNN is just used for computational purposes, not statistical / modeling purposes.

More details on DNNs can be found in such books as [Zha+20a; Cho21; Gér19; GBC16], as well as a multitude of online courses. For a more theoretical treatment, see e.g., [Ber+21a; Cal20; Aro+21; RY21].

16.2 Building blocks of differentiable circuits

In this section we discuss some common **building blocks** used in constructing neural networks. We denote the input to a block as \mathbf{x} and the output as \mathbf{y} .



11 *Figure 16.1: An articial “neuron”, the most basic building block of a DNN. (a) The output y is a weighted
12 combination of the inputs \mathbf{x} , where the weights vector is denoted by \mathbf{w} . (b) Alternative depiction of the
13 neuron’s behavior. The bias term b can be emulated by defining $w_N = b$ and $X_N = 1$.*

14

15

16 16.2.1 Linear layers

17

18 The most basic building block of a DNN is a single “**neuron**”, which corresponds to a real-valued
19 signal y computed by multiplying a vector-valued input signal \mathbf{x} by a weight vector \mathbf{w} , and then
20 adding a bias term b . That is,

21
$$y = f(\mathbf{x}; \theta) = \mathbf{w}^\top \mathbf{x} + b \tag{16.1}$$

22

23 where $\theta = (\mathbf{w}, b)$ are the parameters for the function f . This is depicted in Figure 16.1. (The bias
24 term is omitted for clarity.)

25 It is common to group a set of neurons together into a **layer**. We can then represent the activations
26 of a layer with D units as a vector $\mathbf{z} \in \mathbb{R}^D$. We can transform an input vector of activations \mathbf{x} into
27 an output vector \mathbf{y} by multiplying by a weight matrix \mathbf{W} , an adding an offset vector or bias term \mathbf{b}
28 to get

29
$$\mathbf{y} = f(\mathbf{x}; \theta) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{16.2}$$

30

31 where $\theta = (\mathbf{W}, \mathbf{b})$ are the parameters for the function f . This is called a **linear layer**, or **fully
32 connected layer**.

33 It is common to prepend the bias vector onto the first column of the weight matrix, and to append
34 a 1 to the vector \mathbf{x} , so that we can write this more compactly as $\mathbf{x} = \tilde{\mathbf{W}}^\top \tilde{\mathbf{x}}$, where $\tilde{\mathbf{W}} = [\mathbf{W}, \mathbf{b}]$ and
35 $\tilde{\mathbf{x}} = [\mathbf{x}, 1]$. This allows us to ignore the bias term from our notation if we want to.

36

37 16.2.2 Non-linearities

38 A stack of linear layers is equivalent to a single linear layer where we multiply together all the
39 weight matrices. To get more expressive power we can transform each layer by passing it elementwise
40 (pointwise) through a nonlinear function called an **activation function**. This is denoted by
41

42
$$\mathbf{y} = \varphi(\mathbf{x}) = [\varphi(x_1), \dots, \varphi(x_D)] \tag{16.3}$$

43

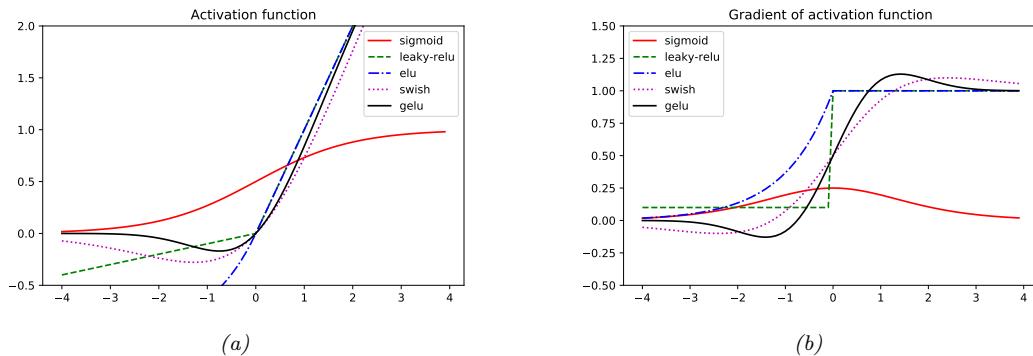
44 See Table 16.1 for a list of some common activation functions, and Figure 16.2 for a visualization.

45 For more details, see e.g., [Mur22, Sec 13.2.3].

46

Name	Definition	Range	Reference
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$[0, 1]$	
Hyperbolic tangent	$\tanh(a) = 2\sigma(2a) - 1$	$[-1, 1]$	
Softplus	$\sigma_+(a) = \log(1 + e^a)$	$[0, \infty]$	[GBB11]
Rectified linear unit	$\text{ReLU}(a) = \max(a, 0)$	$[0, \infty]$	[GBB11; KSH12]
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$	$[-\infty, \infty]$	[MHN13]
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$	$[-\infty, \infty]$	[CUH16]
Swish	$a\sigma(a)$	$[-\infty, \infty]$	[RZL17]
GELU	$a\Phi(a)$	$[-\infty, \infty]$	[HG16]

Table 16.1: List of some popular activation functions for neural networks.

Figure 16.2: (a) Some popular activation functions. “ReLU” stands for “restricted linear unit”. “GELU” stands for “Gaussian error linear unit”. (b) Plot of their gradients. Generated by `activation_fun_deriv_jax.ipynb`.

16.2.3 Convolutional layers

When dealing with image data, we can apply the same weight matrix to each local patch of the image, in order to reduce the number of parameters. If we “slide” this weight matrix over the image and add up the results, we get a technique known as **convolution**; in this case the weight matrix is often called a “**kernel**” or “**filter**”.

More precisely, let $\mathbf{X} \in \mathbb{R}^{H \times W}$ be the input image, and $\mathbf{W} \in \mathbb{R}^{h \times w}$ be the kernel. The output is denoted by $\mathbf{Z} = \mathbf{X} \circledast \mathbf{W}$, where (ignoring boundary conditions) we have the following:¹

$$Z_{i,j} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} x_{i+u, j+v} w_{u,v} \quad (16.4)$$

Essentially we compare a local patch of \mathbf{x} , of size $h \times w$ and centered at (i, j) , to the filter \mathbf{w} ; the output just measures how similar the input patch is to the filter. We can define convolution in 1d or 3d in an analogous manner. Note that the spatial size of the outputs may be smaller than inputs,

1. Note that, technically speaking, we are using **cross correlation** rather than convolution. However, these terms are used interchangeably in deep learning.

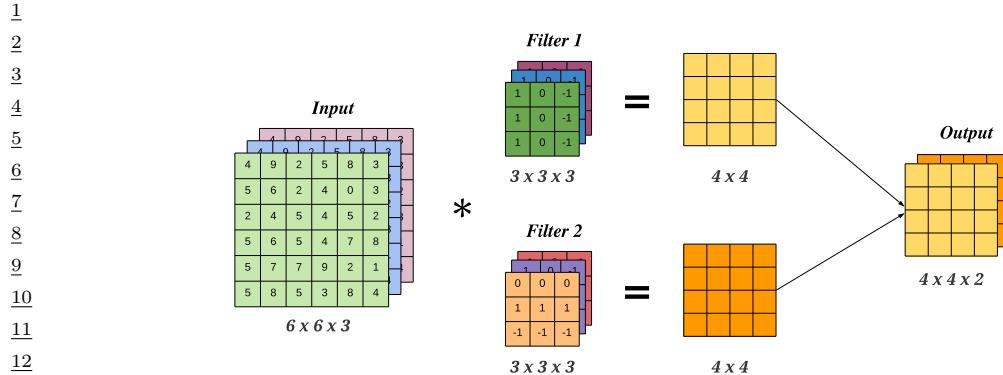


Figure 16.3: A 2d convolutional layer with 3 input channels and 2 output channels. The kernel has size 3×3 and we use stride 1 with 0 padding, so the the 6×6 input gets mapped to the 4×4 output.

due to boundary effects, although this can be solved by using **padding**. See [Mur22, Sec 14.2.1] for more details.

We can repeat this process for multiple layers of inputs, and by using multiple filters, we can generate multiple layers of output. In general, if we have C input channels, and we want to map it to D output (feature) channels, then we define D kernels, each of size $h \times w \times C$, where h, w are the height and width of the kernel. The d 'th output feature map is obtained by convolving all C input feature maps with the d 'th kernel, and then adding up the results elementwise:

$$z_{i,j,d} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} \sum_{c=0}^{C-1} x_{i+u,j+v,c} w_{u,v,c,d} \quad (16.5)$$

This is called a **convolutional layer**, and is illustrated in Figure 16.3.

The advantage of a convolutional layer compared to using a linear layer is that the weights of the kernel are shared across locations in the input. Thus if a pattern in the input shifts locations, the corresponding output activation will also shift. This is called **shift equivariance**. In some cases, we want the output to be the same, no matter where the input pattern occurs; this is called **shift invariance**, and can be obtained by using a **pooling layer**, which computes the maximum or average value in each local patch of the input. (Note that pooling layers have no free (learnable) parameters.) Other forms of invariance can also be captured by neural networks (see e.g., [CW16; FWW21]).

16.2.4 Residual (skip) connections

If we stack a large number of nonlinear layers together, the signal may get squashed to zero or may blow up to infinity, depending on the magnitude of the weights, and the nature of the nonlinearities. Similar problems can plague gradients that are passed backwards through the network (see Section 6.2). To reduce the effect of this we can add **skip connections**, also called **residual connections**, which allow the signal to skip one or more layers, which prevents it from being modified. For example,

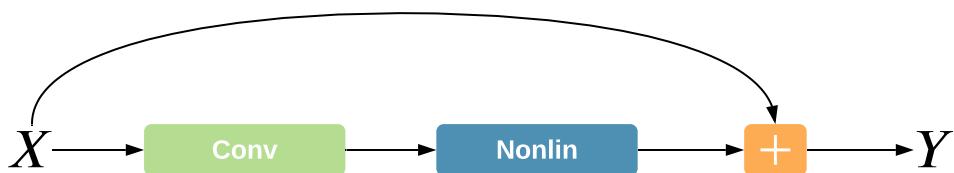


Figure 16.4: A residual connection around a convolutional layer.

Figure 16.4 illustrates a network that computes

$$\mathbf{y} = f(\mathbf{x}; \mathbf{W}) = \varphi(\text{conv}(\mathbf{x}; \mathbf{W})) + \mathbf{x} \quad (16.6)$$

Now the convolutional layer only needs to learn an offset or residual to add (or subtract) to the input to match the desired output, rather than predicting the output directly. Such residuals are often small in size, and hence are easier to learn using neurons with weights that are bounded (e.g., close to 1).

16.2.5 Normalization layers

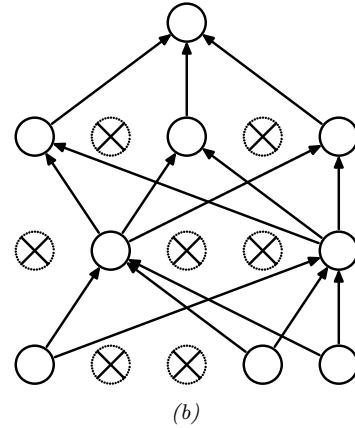
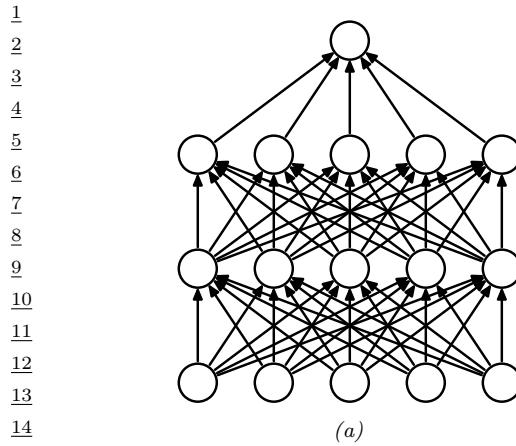
To learn an input-output mapping, it is often best if the inputs are standardized, meaning that they have zero mean and unit standard deviation. This ensures that the required magnitude of the weights is small, and comparable across dimensions. To ensure that the internal activations have this property, it is common to add **normalization layers**.

The most common approach is to use **batch normalization (BN)** [IS15]. However this relies on having access to a batch of $B > 1$ input examples. Various alternatives have been proposed to overcome the need of having an input batch, such as **layer normalization** [BKH16], **instance normalization** [UVL16], **group normalization** [WH18], **filter response normalization** [SK20], etc. More details can be found in [Mur22, Sec 14.2.4].

16.2.6 Dropout layers

Neural networks often have millions of parameters, and thus can sometimes overfit, especially when trained on small datasets. There are many ways to ameliorate this effect, such as applying regularizers to the weights, or adopting a fully Bayesian approach (see Chapter 17). Another common heuristic is known as **dropout** [Sri+14], in which edges are randomly omitted each time the network is used, as illustrated in Figure 16.5. More precisely, if w_{lij} is the weight of the edge from node i in layer $l - 1$ to node j in layer $l + 1$, then we replace it with $\theta_{lij} = w_{lij}\epsilon_{li}$, where $\epsilon_{li} \sim \text{Ber}(1 - p)$, where p is the drop probability, and $1 - p$ is the keep probability. Thus if we sample $\epsilon_{li} = 0$, then all of the weights going out of unit i in layer $l - 1$ into any j in layer l will be set to 0.

During training, the gradients will be zero for the weights connected to a neuron which has been switched “off”. However, since we resample ϵ_{lij} every time the network is used, different combinations of weights will be updated on each step. The result is an **ensemble** of networks, each with slightly different sparse graph structures.



16 *Figure 16.5: Illustration of dropout.* (a) A standard neural net with 2 hidden layers. (b) An example of a
17 thinned net produced by applying dropout with $p = 0.5$. Units that have been dropped out are marked with an
18 \times . From Figure 1 of [Sri+14]. Used with kind permission of Geoff Hinton.

19
20
21
22 At test time, we usually turn the dropout noise off, so the model acts deterministically. To ensure
23 the weights have the same expectation at test time as they did during training (so the input activation
24 to the neurons is the same, on average), at test time we should use $\mathbb{E}[\theta_{lij}] = w_{lij}\mathbb{E}[\epsilon_{li}]$. For Bernoulli
25 noise, we have $\mathbb{E}[\epsilon] = 1 - p$, so we should multiply the weights by the keep probability, $1 - p$, before
26 making predictions. We can, however, use dropout at test time if we wish. This is called **Monte
27 Carlo dropout** (see Section 17.4.5).

28
29
30 **16.2.7 Attention layers**

31 In non-parametric kernel based prediction methods, such as Gaussian processes (Chapter 18), we
32 compare the input $\mathbf{x} \in \mathbb{R}^{d_k}$ to each of the training examples $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_m)$ using a kernel to get
33 a vector of similarity scores, $\boldsymbol{\alpha} = [\mathcal{K}(\mathbf{x}, \mathbf{x}_i)]_{i=1}^m$. We then use this to retrieve a weighted combination
34 of the corresponding m target values $\mathbf{y}_i \in \mathbb{R}^{d_v}$ as follows:
35

36
37
$$\hat{\mathbf{y}} = \sum_{i=1}^m \alpha_i \mathbf{y}_i \tag{16.7}$$

38

39 See Section 18.3.7 for details.

40 We can make a differentiable and parametric version of this as follows (see [Tsa+19] for details).
41 First we replace the stored examples matrix \mathbf{X} with a learned embedding, to create a set of stored
42 **keys**, $\mathbf{K} = \mathbf{W}^K \mathbf{X} \in \mathbb{R}^{m \times d_k}$. Similarly we replace the stored output matrix \mathbf{Y} with a learned
43 embedding, to create a set of stored **values**, $\mathbf{V} = \mathbf{W}^V \mathbf{Y} \in \mathbb{R}^{m \times d_v}$. Finally we embed the input to
44 create a **query**, $\mathbf{q} = \mathbf{W}^Q \mathbf{x} \in \mathbb{R}^{d_k}$. The parameters to be learned are the three embedding matrices.
45 To ensure the output is a differentiable function of the input, we replace the fixed kernel function
46

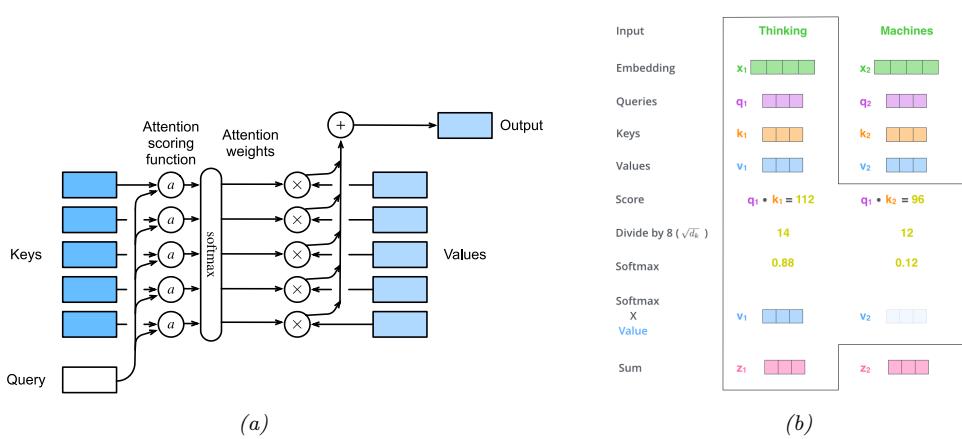


Figure 16.6: Attention layer. (a) Mapping a single query q to a single output, given a set of keys and values. From Figure 10.3.1 of [Zha+20a]. Used with kind permission of Aston Zhang. (b) Detailed visualization of attention. Here the $n = 2$ queries q_j are derived by embedding input vectors x_j corresponding to the words (discrete tokens) “Thinking” and “Machines”. We assume there are $m = 2$ keys and values, and that the queries, keys and values all have the same size, namely $d_k = 64$. (We only show 3 dimensions for brevity.) From <https://jalammar.github.io/illustrated-transformer/>. Used with kind permission of Jay Alammar.

with a soft **attention layer**. More precisely, we define

$$\text{Attn}(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)) = \text{Attn}(\mathbf{q}, (\mathbf{k}_{1:m}, \mathbf{v}_{1:m})) = \sum_{i=1}^m \alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) \mathbf{v}_i \quad (16.8)$$

where $\alpha_i(\mathbf{q}, \mathbf{k}_{1:m})$ is the i 'th **attention weight**; these weights satisfy $0 \leq \alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) \leq 1$ for each i and $\sum_i \alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) = 1$.

The attention weights can be computed from an **attention score** function $a(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$, that computes the similarity of query \mathbf{q} to key \mathbf{k}_i . For example, we can use (scaled) **dot product attention**, which has the form

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d_k} \quad (16.9)$$

(The scaling by $\sqrt{d_k}$ is to reduce the dependence of the output on the dimensionality of the vectors.) Given the scores, we can compute the attention weights using the softmax function:

$$\alpha_i(\mathbf{q}, \mathbf{k}_{1:m}) = \sigma_i([a(\mathbf{q}, \mathbf{k}_1), \dots, a(\mathbf{q}, \mathbf{k}_m)]) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^m \exp(a(\mathbf{q}, \mathbf{k}_j))} \quad (16.10)$$

See Figure 16.6a for an illustration.

In some cases, we want to restrict attention to a subset of the dictionary, corresponding to valid entries. For example, we might want to pad sequences to a fixed length (for efficient minibatching), in which case we should “mask out” the padded locations. This is called **masked attention**. We

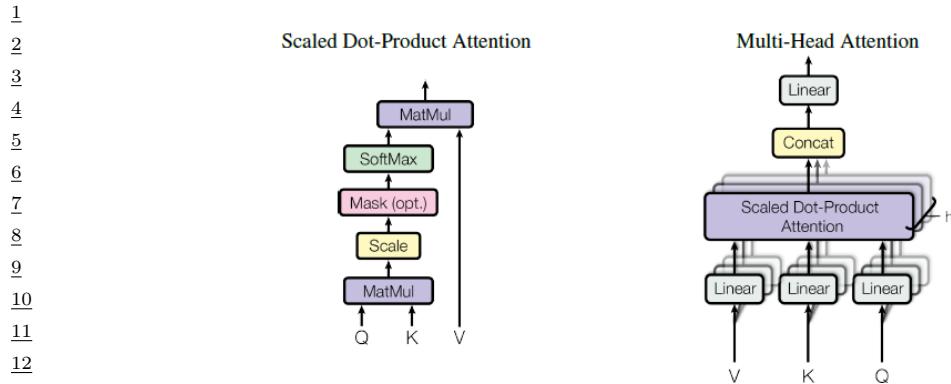


Figure 16.7: (a) Scaled dot-product attention in matrix form. (b) Multi-head attention. From Figure 2 of [Vas+17b]. Used with kind permission of Ashish Vaswani.

can implement this efficiently by setting the attention score for the masked entries to a large negative number, such as -10^6 , so that the corresponding softmax weights will be 0.

In practice, we usually deal with minibatches of n vectors at a time. Let the corresponding matrices of queries, keys and values be denoted by $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{m \times d_k}$, $\mathbf{V} \in \mathbb{R}^{m \times d_v}$. Let

$$\mathbf{z}_j = \sum_{i=1}^m \alpha_i(\mathbf{q}_j, \mathbf{K}) \mathbf{v}_i \quad (16.11)$$

be the j 'th output corresponding to the j 'th query. We can compute all outputs $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$ in parallel using

$$\mathbf{Z} = \text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \sigma\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \quad (16.12)$$

where the softmax function σ is applied row-wise. See Figure 16.7(left) for an illustration, and Figure 16.6b for a detailed worked example.

To increase the flexibility of the model, we often use a **multi-head attention** layer, as illustrated in Figure 16.7(right). Let the i 'th head be

$$\mathbf{h}_i = \text{Attn}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \quad (16.13)$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$ are linear projection matrices. We define the output of the MHA layer to be

$$\mathbf{o} = \text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\mathbf{h}_1, \dots, \mathbf{h}_h)\mathbf{W}^O \quad (16.14)$$

where h is the number of heads, and $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$. Having multiple heads can increase performance of the layer, in the event that some of the weight matrices are poorly initialized; after training, we can often remove all but one of the heads [MLN19].

When the output of one attention layer is used as input to another, the method is called **self-attention**. This is the basis of the transformer model, which we discuss in Section 16.3.4.

47

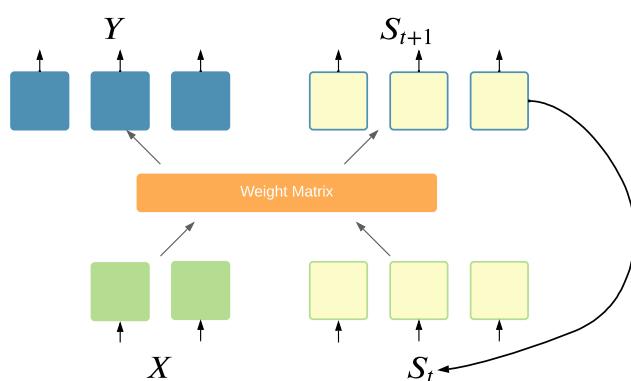


Figure 16.8: Recurrent layer.

16.2.8 Recurrent layers

We can make the model be **stateful** by augmenting the input \mathbf{x} with the current state s_t , and then computing the output and the new state using some kind of function:

$$(y, s_{t+1}) = f(\mathbf{x}, s_t) \quad (16.15)$$

This is called a **recurrent layer**, as shown in Figure 16.8. This forms the basis of **recurrent neural networks**, discussed in Section 16.3.3. In a vanilla RNN, the function f is a simple MLP, but it may also use attention (Section 16.2.7).

16.2.9 Multiplicative layers

In this section, we discuss **multiplicative layers**, which are useful for combining different information sources. Our presentation follows [Jay+20].

Suppose we have inputs $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$. In a linear layer (and, by extension, convolutional layers), it is common to concatenate the inputs to get $f(\mathbf{x}, \mathbf{z}) = \mathbf{W}[\mathbf{x}; \mathbf{z}] + \mathbf{b}$, where $\mathbf{W} \in \mathbb{R}^{k \times (m+n)}$ and $\mathbf{b} \in \mathbb{R}^k$. We can increase the expressive power of the model by using **multiplicative interactions**, such as the following **bilinear form**:

$$f(\mathbf{x}, \mathbf{z}) = \mathbf{z}^\top \mathbb{W} \mathbf{x} + \mathbf{U} \mathbf{z} + \mathbf{V} \mathbf{x} + \mathbf{b} \quad (16.16)$$

where $\mathbb{W} \in \mathbb{R}^{m \times n \times k}$ is a weight tensor, defined such that

$$(\mathbf{z}^\top \mathbb{W} \mathbf{x})_k = \sum_{ij} z_i \mathbb{W}_{ijk} x_j \quad (16.17)$$

That is, the k 'th entry of the output is the weighted inner product of \mathbf{z} and \mathbf{x} , where the weight matrix is the k 'th “slice” of \mathbb{W} . The other parameters have size $\mathbf{U} \in \mathbb{R}^{k \times m}$, $\mathbf{V} \in \mathbb{R}^{k \times n}$, and $\mathbf{b} \in \mathbb{R}^k$.

This formulation includes many interesting special cases. In particular, a **hypernetwork** [HDL17] can be viewed in this way. A hypernetwork is a neural network that generates parameters for another

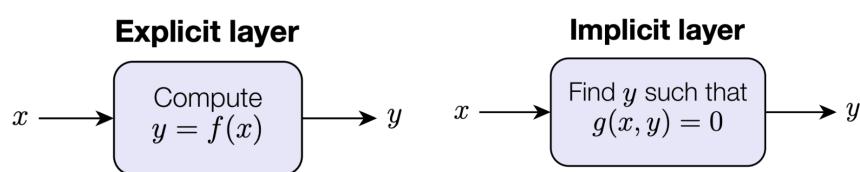


Figure 16.9: Explicit vs implicit layers.

neural network. In particular, we replace $f(\mathbf{x}; \boldsymbol{\theta})$ with $f(\mathbf{x}; g(\mathbf{z}; \boldsymbol{\phi}))$. If f and g are affine, this is equivalent to a multiplicative layer. To see this, let $\mathbf{W}' = \mathbf{z}^\top \mathbf{W} + \mathbf{V}$ and $\mathbf{b}' = \mathbf{U}\mathbf{z} + \mathbf{b}$. If we define $g(\mathbf{z}; \boldsymbol{\Phi}) = [\mathbf{W}', \mathbf{b}']$, and $f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}'\mathbf{x} + \mathbf{b}'$, we recover Equation (16.16).

We can also view the gating layers used in RNNs (Section 16.3.3) as a form of multiplicative interaction. In particular, if we the hypernetwork computes the diagonal matrix $\mathbf{W}' = \sigma(\mathbf{z}^\top \mathbf{W} + \mathbf{V}) = \text{diag}(a_1, \dots, a_n)$, then we can define $f(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta}) = \mathbf{a}(\mathbf{z}) \odot \mathbf{x}$, which is the standard gating mechanism. Attention mechanisms (Section 16.2.7) are also a form of multiplicative interaction, although they involve three-way interactions, between query, key and value.

Another variant arises if the hypernetwork just computes a scalar weight for each channel of a convolutional layer, plus a bias term:

$$f(\mathbf{x}, \mathbf{z}) = \mathbf{a}(\mathbf{z}) \odot \mathbf{x} + \mathbf{b}(\mathbf{z}) \quad (16.18)$$

This is called **FiLM**, which stands for “Feature-wise Linear Modulation” [Per+18]. For a detailed tutorial on the FiLm layer and its many applications, see <https://distill.pub/2018/feature-wise-transformations>.

16.2.10 Implicit layers

So far we have focused on **explicit layers**, which specify how to transform the input to the output using $\mathbf{y} = f(\mathbf{x})$. We can also define **implicit layers**, which specify the output indirectly, in terms of a constraint function:

$$\mathbf{y} \in \underset{\mathbf{y}}{\operatorname{argmin}} f(\mathbf{x}, \mathbf{y}) \text{ such that } g(\mathbf{x}, \mathbf{y}) = 0 \quad (16.19)$$

The details on how to find a solution to this constrained optimization problem can vary depending on the problem. For example, we may need to run an inner optimization routine, or call a differential equation solver. The main advantage of this approach is that the inner computations do not need to be stored explicitly, which saves a lot of memory. Furthermore, once the solution has been found, we can propagate gradients through the whole layer, by leveraging the implicit function theorem. This lets us use higher level primitives inside an end-to-end framework. For more details, see [GHC21] and <http://implicit-layers-tutorial.org/>.

16.3 Canonical examples of neural networks

In this section, we give several “canonical” examples of neural network architectures that are widely used for different tasks.

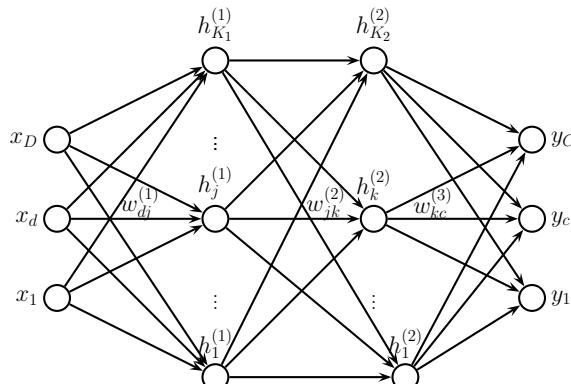


Figure 16.10: A feedforward neural network with D inputs, K_1 hidden units in layer 1, K_2 hidden units in layer 2, and C outputs. $w_{jk}^{(l)}$ is the weight of the connection from node j in layer $l - 1$ to node k in layer l .

16.3.1 Multi-layer perceptrons (MLP)

A multi-layer perceptron (MLP), also called a feedforward neural network (FFNN), is one of the simplest kinds of neural networks. It consists of a series of L linear layers, combined with elementwise nonlinearities:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_L \varphi_L (\mathbf{W}_{L-1} \varphi_{L-1} (\cdots \varphi_1 (\mathbf{W}_1 \mathbf{x}) \cdots)) \quad (16.20)$$

For example, Figure 16.10 shows an MLP with 1 input layer of D units, 2 hidden layers of K_1 and K_2 units, and 1 output layer with C units. The k 'th hidden unit in layer l is given by

$$h_k^{(l)} = \varphi_l \left(b_k^{(l)} + \sum_{j=1}^{K_{l-1}} w_{jk}^{(l)} h_j^{(l-1)} \right) \quad (16.21)$$

where φ_l is the nonlinear activation function at layer l . For a classification problem, the final nonlinearity is usually the softmax function.

16.3.2 Convolutional neural networks (CNN)

A vanilla convolutional neural network or CNN consists of a series of convolutional layers, pooling layers, linear layers, and nonlinearities. See Figure 16.11 for an example. More sophisticated architectures, such as the ResNet model [He+16a; He+16b], add skip (residual) connections, normalization layers, etc. The ConvNeXt model of [Liu+22] is considered the current (as of February 2022) state of the art CNN architecture for a wide variety of vision tasks. See e.g., [Mur22, Ch.14] for details.

16.3.3 Recurrent neural networks (RNN)

A recurrent neural network (RNN) is a network with a recurrent layer, as in Equation (16.15). This is illustrated in Figure 16.12. Formally this defines the following probability distribution over

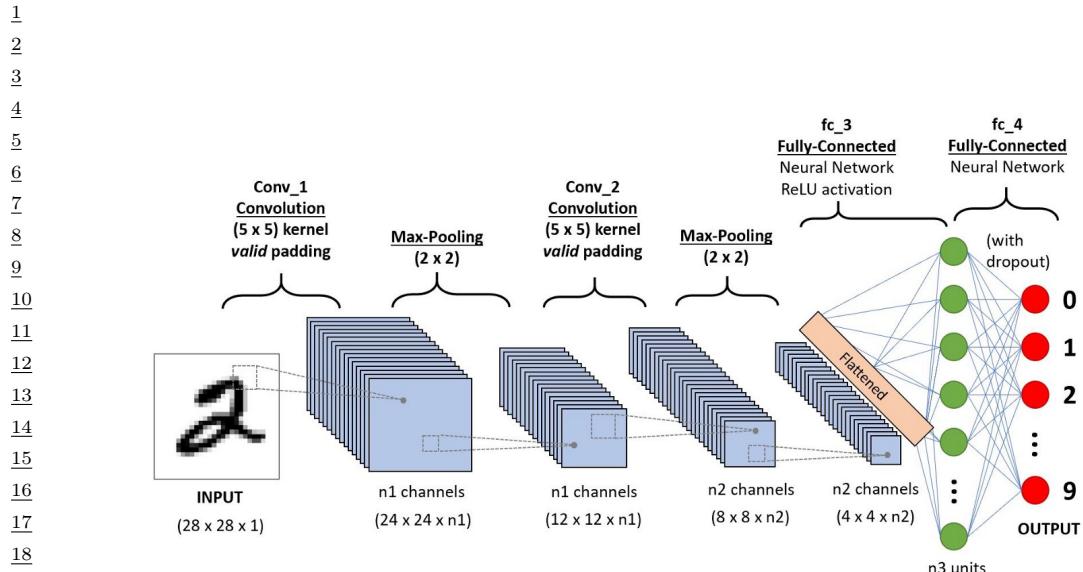


Figure 16.11: A simple CNN for classifying MNIST images. The model has 2 convolutional layers, and 2 fully connected layers, with a softmax output. The dotted square boxes denote the receptive fields. The use of a 5×5 filter with “valid” convolution means we reduce the input size from 28 pixels per side to $28 - 5 + 1 = 24$. The use of 2×2 max-pooling with stride 2 reduces the 24 pixels to 12. The second convolution reduces this to $12 - 5 + 1 = 8$ pixels per side, and the second pooling layer reduces this to 4. As we reduce the spatial size of each layer, we typically increase the number of feature channels (e.g., $n_2 = 2n_1$). The first fully connected (linear) layer (fc-3) maps from n_2 features to n_3 . The second fully connected layer (fc-4) maps from n_3 features to $C = 10$ output logits; this final layer may optionally be regularized with dropout. From <https://bit.ly/2YB9oOH>. Used with kind permission of Sumit Saha.

30

31

32

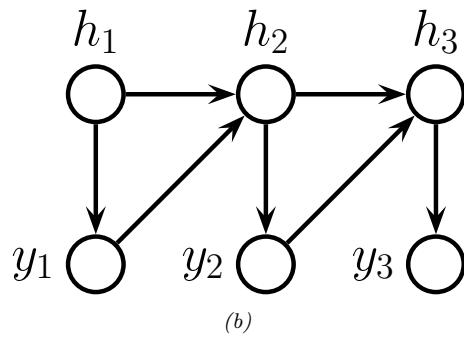
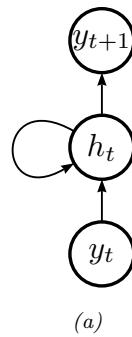


Figure 16.12: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

45

46

47

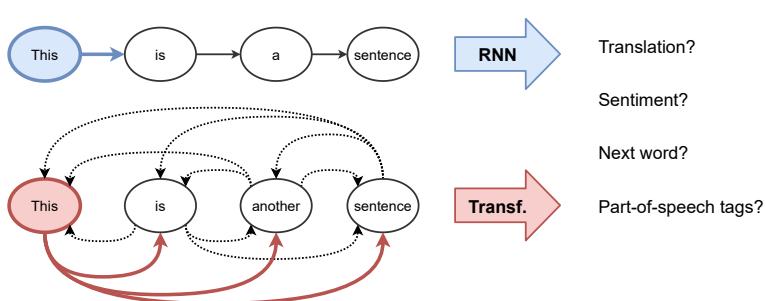


Figure 16.13: Visualizing the difference between an RNN and a transformer. From [Jos20]. Used with kind permission of Chaitanya Joshi.

sequences:

$$p(\mathbf{y}_{1:T}) = \sum_{\mathbf{h}_{1:T}} p(\mathbf{y}_{1:T}, \mathbf{h}_{1:T}) = \sum_{\mathbf{h}_{1:T}} \mathbb{I}(\mathbf{h}_1 = \mathbf{h}_1^*) p(\mathbf{y}_1 | \mathbf{h}_1) \prod_{t=2}^T p(\mathbf{y}_t | \mathbf{h}_t) \mathbb{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1})) \quad (16.22)$$

where \mathbf{h}_t is the deterministic hidden state, computed from the last hidden state and last output using $f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1})$. (At training time, \mathbf{y}_{t-1} is observed, but at prediction time, it is generated.)

In a vanilla RNN, the function f is a simple MLP. However, we can also use attention to selectively update parts of the state vector based on similarity between the input the previous state, as in the **GRU** (gated recurrent unit) model, and the **LSTM** (long short term memory model). We can also make the model into a conditional sequence model, by feeding in extra inputs to the f function. See e.g., [Mur22, Ch. 15] for details.

16.3.4 Transformers

Consider the problem of classifying each word in a sentence, for example with its part of speech tag (noun, verb, etc). That is, we want to learn a mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X} = \mathcal{V}^T$ is the set of input sequences defined over (word) vocabulary \mathcal{V} , T is the length of the sentence, and $\mathcal{Y} = \mathcal{T}^T$ is the set of output sequences, defined over (tag) vocabulary \mathcal{T} . To do well at this task, we need to learn a contextual embedding of each word. RNNs process one token at a time, so the embedding of the word at location t , \mathbf{z}_t , depends on the hidden state of the network, \mathbf{s}_t , which may be a lossy summary of all the previously seen words. We can create bidirectional RNNs so that future words can also affect the embedding of \mathbf{z}_t , but this dependence is still mediated via the hidden state. An alternative approach is to compute \mathbf{z}_t as a direct function of all the other words in the sentence, by using the attention operator discussed in Section 16.2.7 rather than using hidden state. This is called an (encoder-only) **transformer**, and is used by models such as BERT [Dev+19]. This idea is sketched in Figure 16.13.

It is also possible to create a decoder-only transformer, in which each output \mathbf{y}_t only attends to all the previously generated outputs, $\mathbf{y}_{1:t-1}$. This can be implemented using masked attention, and is useful for generative language models, such as GPT (see Section 23.4.1).

We can combine the encoder and decoder to create a conditional sequence-to-sequence model,

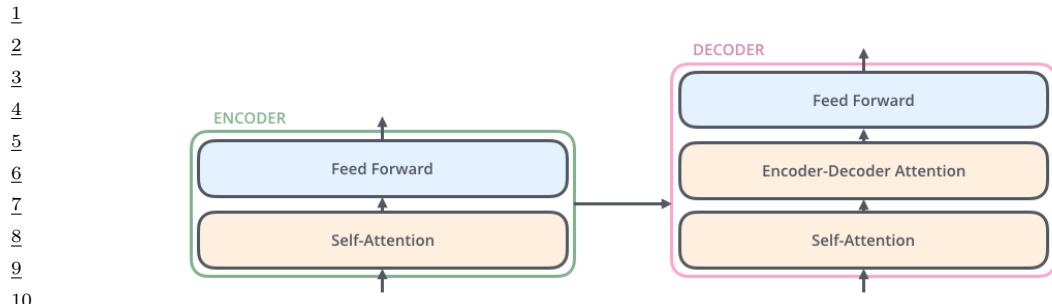


Figure 16.14: High level structure of the encoder-decoder transformer architecture. From <https://jalammar.github.io/illustrated-transformer/>. Used with kind permission of Jay Alammar.

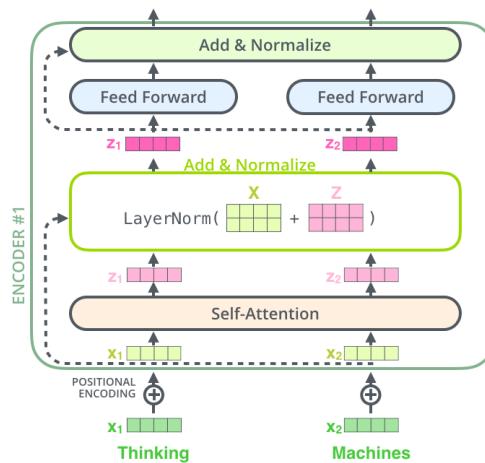


Figure 16.15: The encoder block of a transformer for two input tokens. From <https://jalammar.github.io/illustrated-transformer/>. Used with kind permission of Jay Alammar.

$p(\mathbf{y}_{1:T_y} | \mathbf{x}_{1:T_x})$, as proposed in the original transformer paper [Vas+17c]. The high level structure is shown in Figure 16.14. We give the details below.

16.3.4.1 Encoder

The details of the transformer encoder block are shown in Figure 16.15. The embedded input tokens \mathbf{X} are passed through an attention layer (typically multi-headed), and the output \mathbf{Z} is added to the input \mathbf{X} using a residual connection. This is then passed into a layer normalization layer, which normalizes and learns an affine transformation for each dimension, to ensure all hidden units have comparable magnitude. (This is necessary because the attention masks might upweight just a few locations, resulting in a skewed distribution of values.) Then the output vectors at each location are mapped through an MLP, composed of 1 linear layer, a skip connection and a normalization layer. The overall encoder is N copies of this encoder block. The result is an encoding $\mathbf{H}_x \in \mathbb{R}^{T_x \times D}$

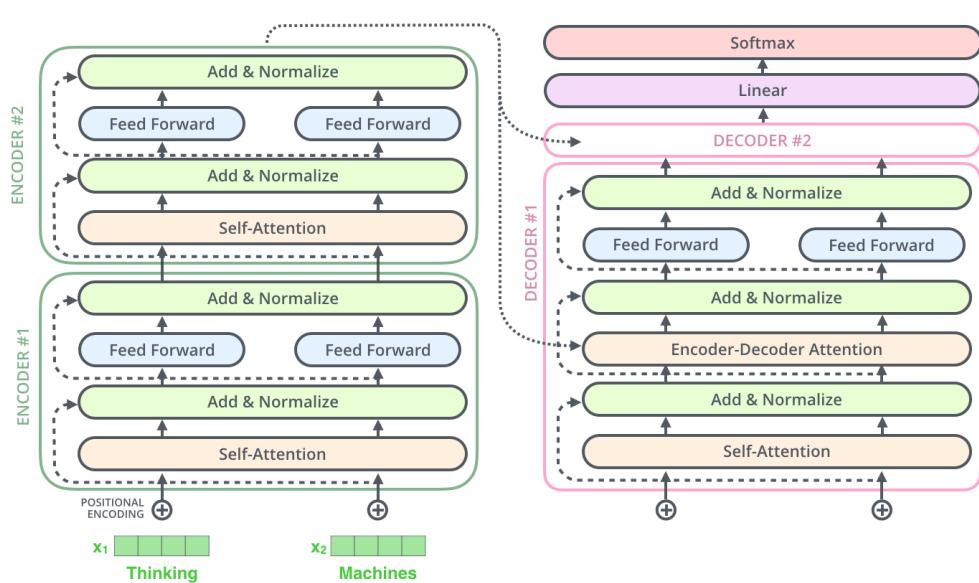


Figure 16.16: A transformer model where we use 2 encoder blocks and 2 decoder blocks. (The second decoder block is not expanded.) We assume there are 2 input and 2 output tokens. From <https://jalammar.github.io/illustrated-transformer/>. Used with kind permission of Jay Alammar.

of the input, where T_x is the number of input tokens, and D is the dimensionality of the attention vectors.

16.3.4.2 Decoder

Once the input has been encoded, the output is generated by the decoder. The first part of the decoder is the decoder attention block, that attends to all previously generated tokens, $\mathbf{y}_{1:t-1}$, and computes the encoding $\mathbf{H}_y \in \mathbb{R}^{T_y \times D}$. This block uses masked attention, so that output t can only attend to locations prior to t in \mathbf{Y} .

The second part of the decoder is the encoder-decoder attention block, that attends to both the encoding of the input, \mathbf{H}_x , and the previously generated outputs, \mathbf{H}_y . These are combined to compute $\mathbf{Z} = \text{Attn}(\mathbf{Q} = \mathbf{H}_y, \mathbf{K} = \mathbf{H}_x, \mathbf{V} = \mathbf{H}_x)$, which compares the output to the input. The joint encoding of the state \mathbf{Z} is then passed through an MLP layer. The full decoder repeats this decoder block N times.

At the end of the decoder, the final output is mapped to a sequence of T_y output logits via a final linear layer.

16.3.4.3 Putting it all together

We can combine the encoder and decoder as shown in Figure 16.16. There is one more detail we need to discuss. This concerns the fact that the attention operation pools information across all locations, so the transformer is invariant to the ordering of the inputs. To overcome this, it is standard to add

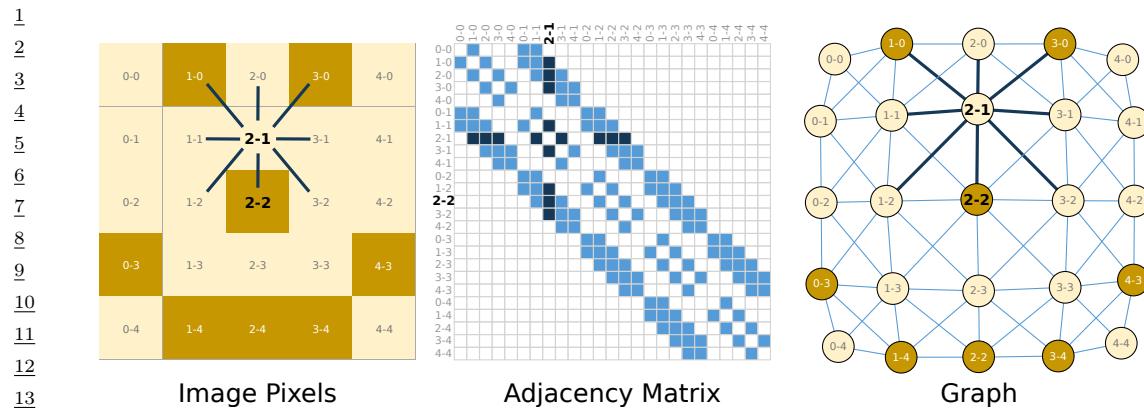


Figure 16.17: Left: Illustration of a 5×5 image, where each pixel is either off (light yellow) or on (dark yellow). Each non-border pixel has 8 nearest neighbors. We highlight the node at location $(2,1)$, where the top-left is $(0,0)$. Middle: The corresponding adjacency matrix, which is sparse and banded. Right: Visualization of the graph structure. Dark nodes correspond to pixels that are on, light nodes correspond to pixels that are off. Dark edges correspond to the neighbors of the node at $(2,1)$. From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

positional encoding vectors to the input tokens $\mathbf{x} \in \mathbb{R}^{T_x \times D}$. That is, we replace \mathbf{x} with $\mathbf{x} + \mathbf{u}$, where $\mathbf{u} \in \mathbb{R}^{T_x \times D}$ is a (possibly learned) vector, where \mathbf{u}_i is some encoding of the fact that x_i comes from the i 'th location in the N -dimensional sequence.

16.3.4.4 Discussion

It has been found that large transformers are very flexible sequence-to-sequence function approximators, if trained on enough data (see e.g., [Lin+21] for a review in the context of NLP, and [Kha+21; Han+20; Zan21] for reviews in the context of computer vision). The reasons why they work so well are still not very clear. However, some initial analysis can be found in e.g., [WGY21; Nel21; BP21]. See also Section 16.3.5.5 where we discuss the connection with graph neural networks.

16.3.5 Graph neural networks (GNNs)

In this section, we discuss **graph neural networks** or **GNNs**. Our presentation is based on [SL+21], which in turn is a summary of the **message passing neural network** framework of [Gil+17] and the **Graph Nets** framework of [Bat+18].

We assume the graph is represented as a set of N nodes or vertices, each associated with a feature vector to create the matrix $\mathbf{V} \in \mathbb{R}^{N \times D_v}$; a set of E edges, each associated with a feature vector to create the matrix $\mathbf{E} \in \mathbb{R}^{E \times D_e}$; and a global feature vector $\mathbf{u} \in \mathbb{R}^{D_u}$, representing overall properties of the graph, such as its size. (We can think of \mathbf{u} as the features associated with a global or master node.) The topology of the graph can be represented as an $N \times N$ **adjacency matrix**, but since this is usually very sparse (see Figure 16.17 for an example), a more compact representation is just to store the list of edges in an **adjacency list** (see Figure 16.18 for an example).

47

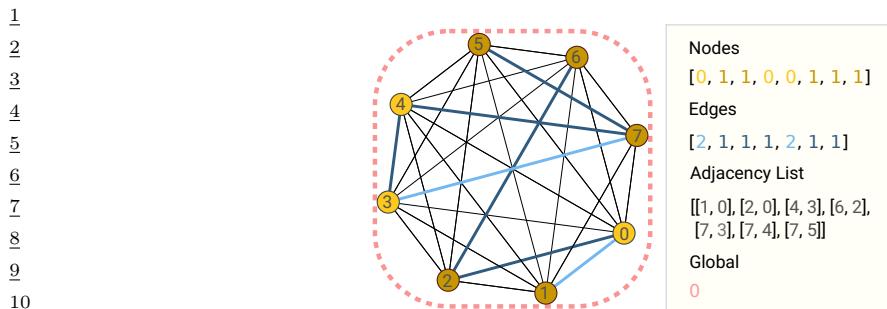


Figure 16.18: A simple graph where each node has 2 types (0=light yellow, 1=dark yellow), each edge has 2 types (1=gray, 2=blue), and the global feature vector is a constant (0=red). We represent the topology using an adjacency list. From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

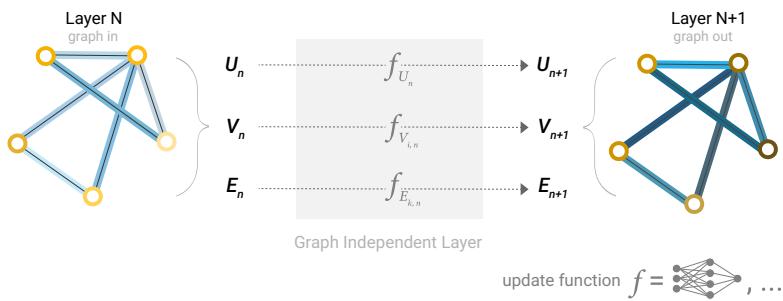


Figure 16.19: A basic GNN layer. We update the embedding vectors \mathbf{U} , \mathbf{V} and \mathbf{V} using the global, node and edge functions f . From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

16.3.5.1 Basics of GNNs

A GNN adopts a “graph in, graph out” philosophy, similar to how transformers map from sequences to sequences. A basic GNN layer updates the embedding vectors associated with the nodes, edges and whole graph, as illustrated in Figure 16.19. The update functions are typically simple MLPs, that are applied independently to each embedding vector.

To leverage the graph structure, we can combine information using a **pooling** operation. That is, for each node n , we extract the feature vectors associated with its edges, and combine it with its local feature vector using a permutation invariant operation such as summation or averaging. See Figure 16.20 for an illustration. We denote this pooling operation by $\rho_{E_n \rightarrow V_n}$. We can similarly pool from nodes to edges, $\rho_{V_n \rightarrow E_n}$, or from nodes to globals, $\rho_{V_n \rightarrow U_n}$, etc.

The overall GNN is composed of GNN layers and pooling layers. At the end of the network, we can use the final embeddings to classify nodes, edges, or the whole graph. See Figure 16.21 for an illustration.

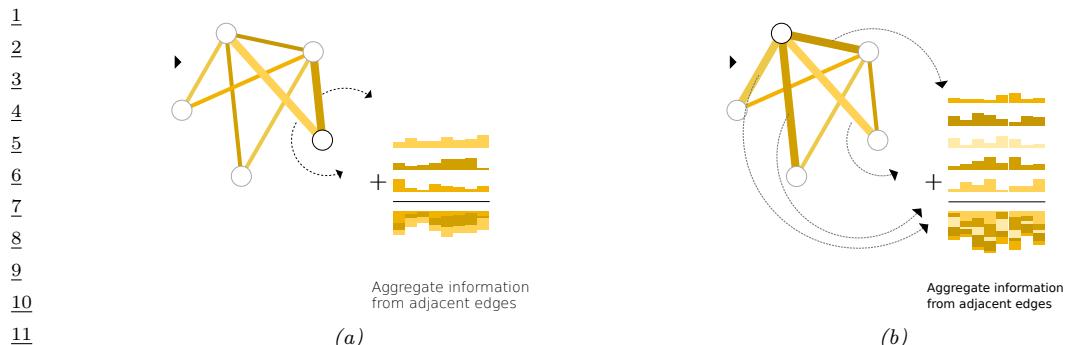


Figure 16.20: Aggregating edge information into two different nodes. From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

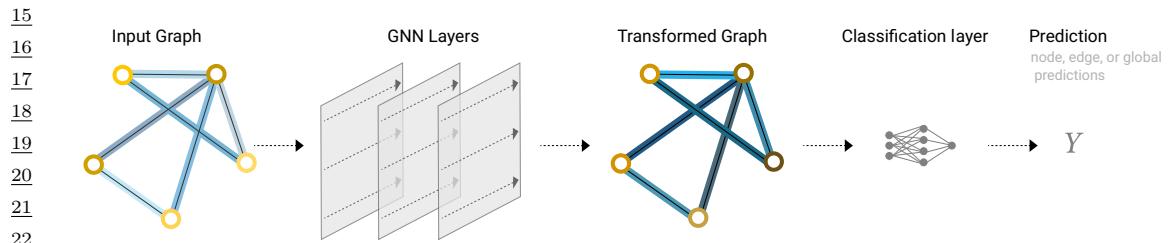


Figure 16.21: An end-to-end GNN classifier. From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

16.3.5.2 Message passing

Instead of transforming each vector independently and then pooling, we can first pool the information for each node (or edge) and then update its vector representation. That is, for node i , we **gather** information from all neighboring nodes, $\{\mathbf{h}_j : j \in \text{nbr}(i)\}$; we **aggregate** these vectors with the local vector using an operation such as sum; and then we compute the new state using an **update** function, such as

$$\mathbf{h}'_i = \text{ReLU}(\mathbf{U}\mathbf{h}_i + \sum_{j \in \text{nbr}(i)} \mathbf{V}\mathbf{h}_j) \quad (16.23)$$

See Figure 16.24a for a visualization.

The above operation can be viewed as a form of “**message passing**”, in which the values of neighboring nodes \mathbf{h}_j are sent to node i and then combined. It is more general than belief propagation (Section 9.2), since the messages are not restricted to represent probability distributions (see Section 9.3.7 for more discussion).

After K message passing layers, each node will have received information from neighbors which are K steps away in the graph. This can be “short circuited” by sending messages through the global node, which acts as a kind of bottleneck. See Figure 16.22 for an illustration.

47

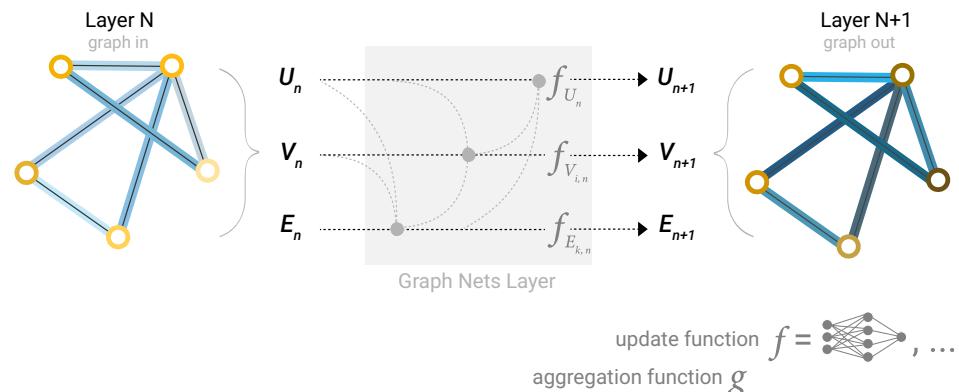


Figure 16.22: Message passing in one layer of a GNN. First the global node U_n and the local nodes V_n send messages to the edges E_n , which get updated to give E_{n+1} . Then the nodes get updated to give V_{n+1} . Finally the global node gets updated to give U_{n+1} . From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

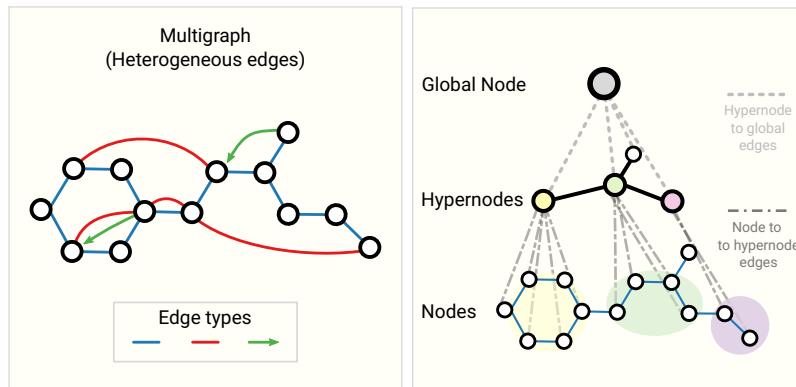


Figure 16.23: Left: a multigraph can have different edge types. Right: a hypergraph can have edges which connect multiple nodes. From [SL+21]. Used with kind permission of Benjamin Sanchez-Lengeling.

16.3.5.3 More complex types of graphs

We can easily generalize this framework to handle other graph types. For example, **multigraphs** have multiple edge types between each pair of nodes. For example, in a **knowledge graph**, we might have edge types “spouse-of”, “employed-by” or “born-in”. See Figure 16.23(left) for an example. In **hypergraphs**, each edge may connect more than two nodes. For example, in a knowledge graph, we might want to specify the three-way relation “parents-of(c, m, f)”, for child c , mother m and father f . We can “reify” such hyperedges into hypernodes, as shown in Figure 16.23(right).

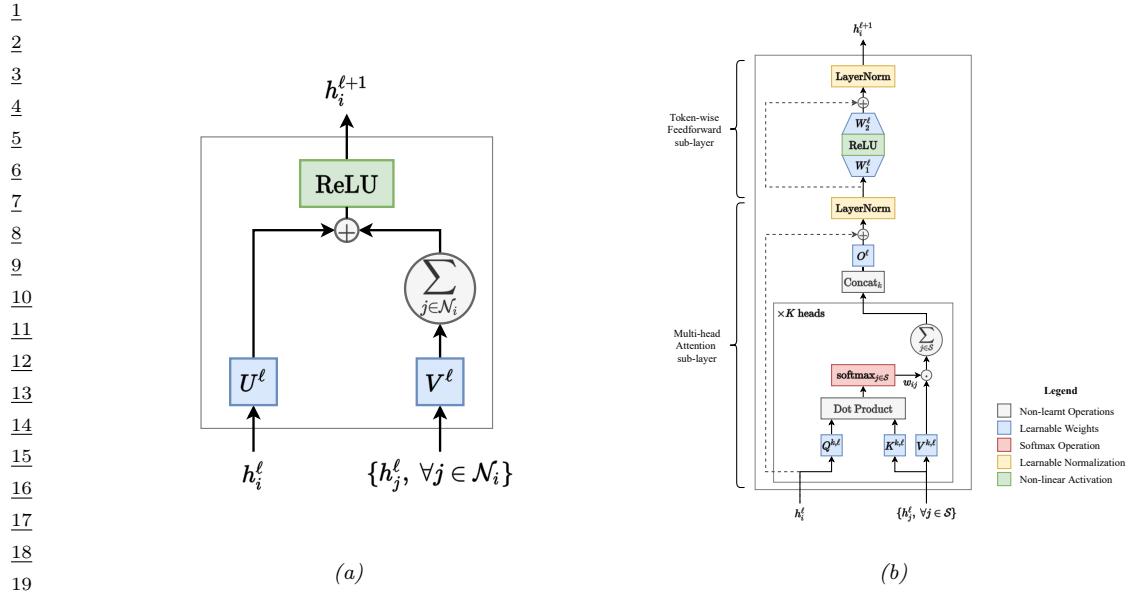


Figure 16.24: (a) A graph neural network aggregation block. Here h_i^ℓ is the hidden representation for node i in layer ℓ , and $\mathcal{N}(i)$ are i 's neighbors. The output is given by $h_i^{\ell+1} = \text{ReLU}(\mathbf{U}^\ell h_i + \sum_{j \in \text{nbr}(i)} \mathbf{V}^\ell h_j^\ell)$. (b) A transformer encoder block. Here h_i^ℓ is the hidden representation for word i in layer ℓ , and \mathcal{S} are all the words in the sentence. The output is given by $h_i^{\ell+1} = \text{Attn}(\mathbf{Q}^\ell h_i^\ell, \{\mathbf{K}^\ell h_j, \mathbf{V}^\ell h_j^\ell\})$. From [Jos20]. Used with kind permission of Chaitanya Joshi.

16.3.5.4 Graph attention networks

When performing message passing, we can generalize the linear combination used in Equation (16.23) to use a weighted combination instead, where the weights are computed an attention mechanism (Section 16.2.7). The resulting model is called a **graph attention network** or **GAT** [Vel+18]. This allows the effective topology of the graph to be context dependent.

16.3.5.5 Transformers are fully connected GNNs

Suppose we create a fully connected graph in which each node represents a word in a sentence. Let us use this to construct a GNN composed of GAT layers, where we use multi-headed scaled dot product attention. Suppose we combine each GAT block with layer normalization and an MLP. The resulting block is shown in Figure 16.24b. We see that this is identical to the transformer encoder block shown in Figure 16.15. This construction shows that transformers are just a special case of GNNs [Jos20].

The advantage of this observation is that it naturally suggests ways to overcome the $O(N^2)$ complexity of transformers. For example, in **Transformer-XL** [Dai+19c], we create blocks of nodes, and connect these together, as shown in Figure 16.25(top right). In **binary partition transformer** or **BPT** [Ye+19], we also create blocks of nodes, but add them as virtual “hypernodes”, as shown in Figure 16.25(bottom). There are many other approaches to reducing the $O(N^2)$ cost (see e.g., [Mur22, Sec 15.6]), but the GNN perspective is a helpful one.

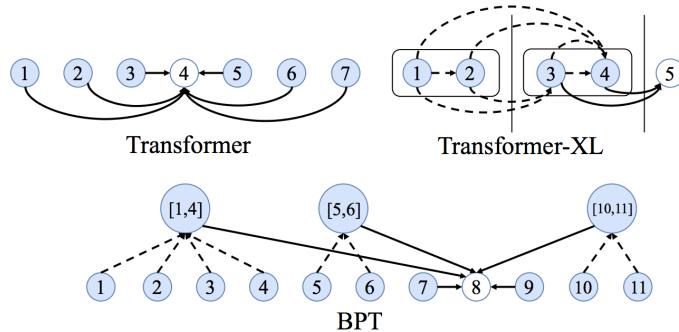


Figure 16.25: Graph connectivity for different types of transformer. Top left: in a vanilla Transformer, every node is connected to every other node. Top right: in Transformer-XL, nodes are grouped into blocks. Bottom: in BPT, we use a binary partitioning of the graph to create virtual node clusters. From <https://graphdeeplearning.github.io/post/transformers-are-gnns/>. Used with kind permission of Chaitanya Joshi.

17 Bayesian neural networks

This chapter is coauthored with Andrew Wilson.

17.1 Introduction

Modern DNNs are usually trained using a (penalized) maximum likelihood objective to find a single setting of parameters. However, large flexible models like neural networks can represent many functions, corresponding to different parameter settings, which fit the training data well, yet generalize in different ways. Considering all of these different models together can lead to improved accuracy and uncertainty representation.

Bayesian inference provides a compelling mechanism to combine these different models together. To use a **Bayesian neural network (BNN)**, we start by specifying a prior distribution over model parameters $p(\boldsymbol{\theta})$, which induces a prior distribution over neural network functions. We then infer a posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$, from which we can compute the posterior predictive distribution using Bayesian model averaging:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (17.1)$$

Early work in this space, much of which was pioneered by David MacKay and Radford Neal in the 1990s [Mac92b; Mac95; Nea95], focused on small models. In this chapter we focus on techniques that scale to newer, larger models. The resulting field is sometimes called **Bayesian deep learning**, to emphasize that we are applying Bayesian inference to “deep” models with many learnable layers. For more details, on this topic, see e.g., [PS17; Wil20; WI20; Jos+22; Kha20].

17.2 Priors for BNNs

To perform Bayesian inference for the parameters of a DNN, we need to specify a prior $p(\boldsymbol{\theta})$. [Nal18; WI20; For21] discusses the issue of prior selection at length. Here we briefly discuss common approaches, and considerations for prior specification in Bayesian deep learning.

In the case of an MLP, we will assume L hidden layers and a linear output:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_L(\cdots \varphi(\mathbf{W}_1\varphi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)) + \mathbf{b}_L \quad (17.2)$$

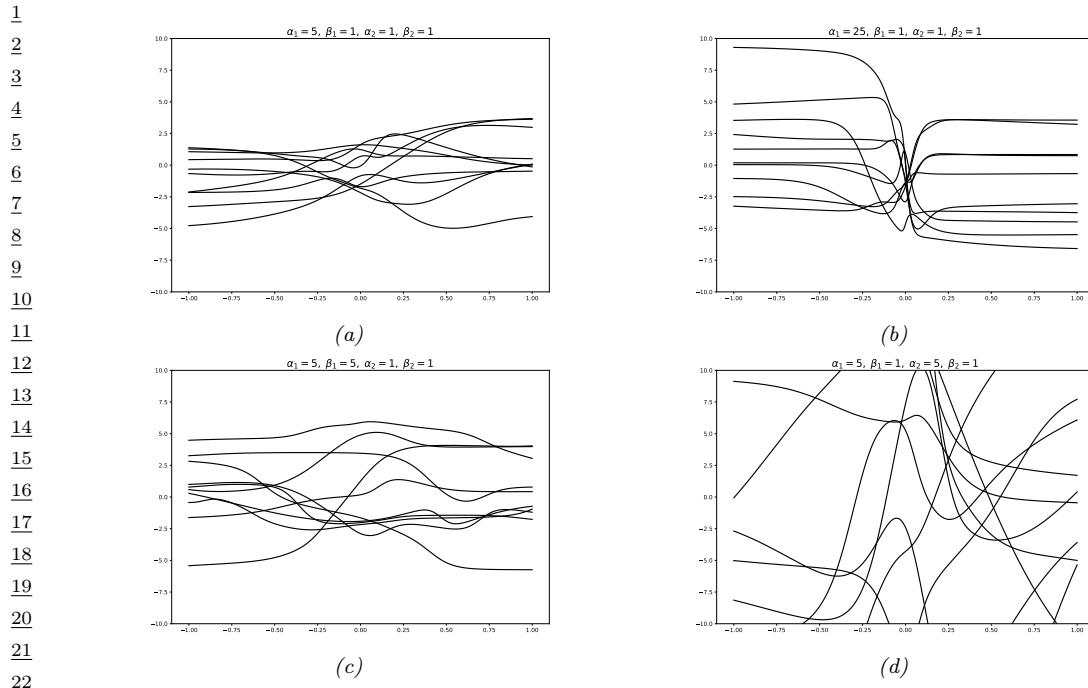


Figure 17.1: The effects of changing the hyperparameters on an MLP with one hidden layer. (a) Random functions sampled from a Gaussian prior with hyperparameters $\alpha_1 = 5, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$. (b) Increasing α_1 by factor of 5. (c) Increasing β_1 by factor of 5. (d) Increasing α_2 by factor of 5. Generated by `mlpPriorsDemo2.py`.

27

28

17.2.1 Gaussian priors

The most common choice is to use a factored Gaussian prior:

$$\mathbf{W}_\ell \sim \mathcal{N}(\mathbf{0}, \alpha_\ell^2 \mathbf{I}), \mathbf{b}_\ell \sim \mathcal{N}(\mathbf{0}, \beta_\ell^2 \mathbf{I}) \quad (17.3)$$

The **Xavier initialization** or **Glorot initialization**, named after the first author of [GB10], is to set

$$\alpha_\ell^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \quad (17.4)$$

where n_{in} is the fan-in of a node in level ℓ (number of weights coming into a neuron), and n_{out} is the fan-out (number of weights going out of a neuron). **LeCun initialization**, named after Yann LeCun, corresponds to using

$$\alpha_\ell^2 = \frac{1}{n_{\text{in}}} \quad (17.5)$$

46

47

We can get a better understanding of these priors by considering the effect they have on the corresponding distribution over functions that they define. To help understand this correspondence, let us reparameterize the model as follows:

$$\mathbf{W}_\ell = \alpha_\ell \boldsymbol{\eta}_\ell, \quad \boldsymbol{\eta}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad \mathbf{b}_\ell = \beta_\ell \boldsymbol{\epsilon}_\ell, \quad \boldsymbol{\epsilon}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (17.6)$$

Hence every setting of the prior hyperparameters specifies the following random function:

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \alpha_L \boldsymbol{\eta}_L (\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x} + \beta_1 \boldsymbol{\epsilon}_1)) + \beta_L \boldsymbol{\epsilon}_L \quad (17.7)$$

To get a feeling for the effect of these hyperparameters, we can sample MLP parameters from this prior and plot the resulting random functions. We use a sigmoid nonlinearity, so $\varphi(a) = \sigma(a)$. We consider $L = 2$ layers, so \mathbf{W}_1 are the input-to-hidden weights, and \mathbf{W}_2 are the hidden-to-output weights. We assume the input and output are scalars, so we are generating random nonlinear mappings $f : \mathbb{R} \rightarrow \mathbb{R}$.

Figure 17.1(a) shows some sampled functions where $\alpha_1 = 5$, $\beta_1 = 1$, $\alpha_2 = 1$, $\beta_2 = 1$. In Figure 17.1(b) we increase α_1 ; this allows the first layer weights to get bigger, making the sigmoid-like shape of the functions steeper. In Figure 17.1(c), we increase β_1 ; this allows the first layer biases to get bigger, which allows the center of the sigmoid to shift left and right more, away from the origin. In Figure 17.1(d), we increase α_2 ; this allows the second layer linear weights to get bigger, making the functions more “wiggly” (greater sensitivity to change in the input, and hence larger dynamic range).

The above results are specific to the case of sigmoidal activation functions. ReLU units can behave differently. For example, [WI20, App. E] show that for MLPs with ReLU units, if we set $\beta_\ell = 0$, so the bias terms are all zero, the effect of changing α_ℓ is just to rescale the output. To see this, note that Equation (17.7) simplifies to

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta} = \mathbf{0}) = \alpha_L \boldsymbol{\eta}_L (\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x})) = \alpha_L \cdots \alpha_1 \boldsymbol{\eta}_L (\cdots \varphi(\boldsymbol{\eta}_1 \mathbf{x})) \quad (17.8)$$

$$= \alpha_L \cdots \alpha_1 f(\mathbf{x}; (\boldsymbol{\alpha} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0})) \quad (17.9)$$

where we used the fact that for ReLU, $\varphi(\alpha z) = \alpha \varphi(z)$ for any positive α , and $\varphi(\alpha z) = 0$ for any negative α (since the pre-activation $z \geq 0$). In general, it is the ratio of α and β that matters for determining what happens to input signals as they propagate forwards and backwards through a randomly initialized model; for details, see e.g., [Bah+20].

We see that initializing the model’s parameters at a particular random value is like sampling a point from this prior over functions. In the limit of infinitely wide neural networks, we can derive this prior distribution analytically: this is known as a **neural network Gaussian process**, and is explained in Section 18.7.

17.2.2 Sparsity-promoting priors

Although Gaussian priors are simple and widely used, they are not the only option. For some applications, it is useful to use **sparsity promoting priors**, such as the Laplace, which encourage most of the weights (or channels in a CNN) to be zero (c.f., Section 15.2.5). For details, see [Hoe+21].

1 2 **17.2.3 Learning the prior**

3 We have seen how different priors for the parameters correspond to different priors over functions.
4 We could in principle set the hyperparameters (e.g., the α and β parameters of the Gaussian prior
5 using grid search to optimize cross-validation loss. However, cross-validation can be slow, particularly
6 if we allow different priors for each layer of the network, as our grid search will grow exponentially
7 with the number of hyperparameters we wish to determine.

8 An alternative is to use gradient based methods to optimize the marginal likelihood

$$\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \int \log p(\mathcal{D}|\boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{\alpha}, \boldsymbol{\beta}) d\boldsymbol{\theta} \quad (17.10)$$

12 This approach is known as empirical Bayes (Section 3.6) or **evidence maximization**, since
13 $\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta})$ is also called the evidence [Mac92a; WS93; Mac99b]. This can give rise to sparse models,
14 as we discussed in the context of automatic relevancy determination (Section 15.2.7). Unfortunately,
15 computing the marginal likelihood is computationally difficult for large neural networks.

17 18 **17.2.4 Priors in function space**

19 Typically, the relationship between the prior distribution over parameters and the functions preferred
20 by the prior is not transparent. In some cases, it can be possible to pick more informative priors,
21 based on principles such as desired invariances that we want the function to satisfy (see e.g., [Nal18]).
22 [FBW21] introduces *residual pathway priors*, providing a mechanism for encoding high level concepts
23 into prior distributions, such as locality, independencies, and symmetries, without constraining model
24 flexibility. A different approach to encoding interpretable priors over functions leverages kernel
25 methods such as Gaussian processes (e.g., [Sun+19a]), as we discuss in Section 18.1.

26 27 **17.2.5 Architectural priors**

29 Ultimately, the prior that impacts generalization is the prior induced in *function space*, which arises
30 from the combination of a prior over parameters $p(\boldsymbol{\theta})$ with the functional form of the model $f(\mathbf{x}; \boldsymbol{\theta})$.

31 As argued in Wilson and Izmailov [WI20], there is strong evidence to suggest that the prior over
32 functions implied by even generic $\mathcal{N}(\boldsymbol{\theta}|0, \alpha^2 I)$ priors over parameters in combination with a neural
33 architecture, has many desirable statistical properties, especially with structured DNNs such as
34 CNNs (Section 16.3.2).

35 For example, Ulyanov, Vedaldi, and Lempitsky [UVL18] showed that an untrained CNN with
36 random parameters (sampled from a Gaussian) often works very well for low-level image processing
37 tasks, such as image denoising, super-resolution and image inpainting. The resulting prior over
38 functions has been called the **deep image prior**. Similarly, Pinto and Cox [PC12] showed that
39 untrained CNNs with the right structure can do well at face recognition. Moreover, Zhang et al.
40 [Zha+17] show that randomly initialized CNNs can process data to provide features that greatly
41 improve the performance of other models, such as kernel methods. Wilson and Izmailov [WI20]
42 additionally show that the prior over functions implied by a generic Gaussian distribution over
43 parameters can induce a reasonable correlation function over images.

44 Moreover, Izmailov et al. [Izm+21b] show that using a high variance α^2 of a conventional Gaussian
45 prior $\mathcal{N}(\boldsymbol{\theta}|0, \alpha^2 I)$ leads to good performance, and that the variance scale, or changing to different
46 heavy-tailed logistic or mixture of Gaussian priors, has only a minor effect on the predictive

47

1 distribution. These results highlight the relative importance of architecture over parameter priors in
 2 specifying a useful prior over functions.
 3

4 Indeed, the architecture of a neural network encodes useful prior knowledge. A CNN architecture
 5 encodes prior knowledge about translation invariance, due to its use of convolution, and hierarchical
 6 structure, due to its use of multiple layers. Other forms of inductive bias are induced by different
 7 architectures, such as RNNs. Thus we can think of the field of **neural architecture search**
 8 (reviewed in [EMH19]) as a form of structural prior learning.
 9

10 17.3 Likelihoods for BNNS

11 In this section, we discuss the likelihood model $p(y|\mathbf{x}, \boldsymbol{\theta})$ used by common Bayesian neural networks.
 12 This is usually taken to be the same as any other classification or regression model. For example, we
 13 may use
 14

$$16 p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\sigma(f(\mathbf{x}; \boldsymbol{\theta}))) \quad (17.11)$$

17 where $f(\mathbf{x}; \boldsymbol{\theta}) \in \mathbb{R}^C$ returns the logits over the C class labels. This is the same as in multinomial
 18 logistic regression (Section 15.3.2); the only difference is that f is a nonlinear function of $\boldsymbol{\theta}$.
 19

20 However, in practice, it is often found (see e.g., [Zha+18a; Wen+20b; LST21]) that BNNS give
 21 better predictive accuracy if the likelihood function is scaled by some power α . That is, instead
 22 of targeting the posterior $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta})$, these methods target the **tempered posterior**,
 23 $p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})^\alpha p(\boldsymbol{\theta})$. In log space, we have
 24

$$25 \log p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) = \alpha \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const} \quad (17.12)$$

26 This is also called an **α -posterior** or **power posterior** [Med+21].
 27

Another common method is to target the **cold posterior**, $p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})^{1/T}$, or, in log
 28 space,
 29

$$30 \log p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{T} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \frac{1}{T} \log p(\boldsymbol{\theta}) + \text{const} \quad (17.13)$$

32 If $T < 1$, we say that the posterior is “cold”. Note that the only difference between cold and tempered
 33 posteriors is that in the tempering case, the prior is not scaled. However, in the case of a Gaussian
 34 prior, using the cold prior is the same as using the tempered prior with a different hyperparameter,
 35 since
 36

$$37 \frac{1}{T} \log \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{cold}}^2 \mathbf{I}) = -\frac{1}{2T\sigma_{\text{cold}}^2} \sum_i \theta_i^2 + \text{const} = \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{tempered}}^2 \mathbf{I}) + \text{const} \quad (17.14)$$

40 where $\sigma_{\text{tempered}}^2 = T\sigma_{\text{cold}}^2$. Thus both methods are effectively the same, and just reweight the
 41 likelihood.
 42

In BNNS for image classification problems, it has been found (see e.g., [Zha+18a; Wen+20b])
 43 that using $\alpha > 1$ (or equivalently, $T < 1$) results in better predictive accuracy. There are several
 44 explanations for this, summarized in [Noc+21]. (The main causes include: standard benchmarks are
 45 highly curated and have little noise [Ait21]; many models are trained with data augmentation which
 46 changes the likelihood [Izm+21b]; and many models use “bad priors” [Wen+20b].)
 47

¹ Note that the cold posterior effect refers to the fact that sometimes using $\alpha > 1$ (or equivalently
² $T < 1$) gives better results than $\alpha = 1$. However, in the case of **model misspecification**, it can be
³ provably better to use $\alpha < 1$ (see e.g., [GO17; KJD21; Med+21; ZN20]).

⁴
⁵ In ??, we discuss generalized variational inference, which gives a general framework for studying
⁶ such problems. In particular, it makes it clear that one can decide whether we should use $\alpha < 1$,
⁷ $\alpha > 1$ or just $\alpha = 1$ using any model selection method, such as cross validation.

⁸ 17.4 Posteriors for BNNs

⁹
¹⁰ There are a large number of different approximate inference schemes that have been applied to
¹¹ Bayesian neural networks, with different strengths and limitations. In the sections below, we briefly
¹² describe some of these.

¹³ 17.4.1 Laplace approximation

¹⁴
¹⁵ In Section 7.4.3, we introduced the Laplace approximation, which computes a Gaussian approximation
¹⁶ to the posterior, $p(\boldsymbol{\theta}|\mathcal{D})$, centered at the MAP estimate, $\boldsymbol{\theta}^*$, and whose precision is equal to the
¹⁷ Hessian of the negative log joint computed at the mode. The benefits of this approach are that it
¹⁸ is simple, and it can be used to derive a Bayesian estimate from a pretrained model. A detailed
¹⁹ explanation of the method in the context of DNNs can be found in [Dax+21]; here we just give a
²⁰ summary.

²¹ Let $\mathbf{f}(\mathbf{x}_n, \boldsymbol{\theta}) \in \mathbb{R}^C$ be the prediction function with C outputs, and $\boldsymbol{\theta} \in \mathbb{R}^P$ be the parameter
²² vector. Let $\mathbf{r}(\mathbf{y}; \mathbf{f}) = \nabla_{\mathbf{f}} \log p(\mathbf{y}|\mathbf{f})$ be the residual¹, and $\boldsymbol{\Lambda}(\mathbf{y}; \mathbf{f}) = -\nabla_{\mathbf{f}}^2 \log p(\mathbf{y}|\mathbf{f})$ be the per-input
²³ noise term. In addition, let $\mathbf{J} \in \mathbb{R}^{C \times P}$ be the Jacobian, $[\mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})]_{ci} = \frac{\partial f_c(\mathbf{x}, \boldsymbol{\theta})}{\partial \theta_i}$, and $\mathbf{H} \in \mathbb{R}^{C \times P \times P}$ be
²⁴ the Hessian, $[\mathbf{H}_{\boldsymbol{\theta}}(\mathbf{x})]_{cij} = \frac{\partial^2 f_c(\mathbf{x}, \boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}$. Then the gradient and Hessian of the log likelihood are given by
²⁵ the following [IKB21]:

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) \quad (17.15)$$

$$\nabla_{\boldsymbol{\theta}}^2 \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{H}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) - \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \boldsymbol{\Lambda}(\mathbf{y}; \mathbf{f}) \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{\theta}) \quad (17.16)$$

²⁶ Since the network Hessian \mathbf{H} is usually intractable to compute, it is usually dropped, leaving only the
²⁷ Jacobian term. This is called the **generalized Gauss-Newton** or **GGN** approximation [Sch02;
²⁸ Mar20]. The GGN approximation is guaranteed to be positive definite, whereas the original Hessian
²⁹ in Equation (17.16) (since the objective is not convex). Furthermore, computing the Jacobian term
³⁰ only takes $O(PC)$ time and space, where P is the number of parameters.

³¹ Putting it all together, for a Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}_0, \mathbf{S}_0)$, the Laplace approximation
³² becomes $p(\boldsymbol{\theta}|\mathcal{D}) \approx (\mathcal{N}|\boldsymbol{\theta}^*, \boldsymbol{\Sigma}_{\text{GGN}})$, where

$$\boldsymbol{\Sigma}_{\text{GGN}}^{-1} = \sum_{n=1}^N \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n)^T \boldsymbol{\Lambda}(\mathbf{y}_n; \mathbf{f}_n) \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n) + \mathbf{S}_0^{-1} \quad (17.17)$$

³³ Unfortunately inverting this matrix takes $O(P^3)$ time, so for models with many parameters, further
³⁴ approximations are usually used. The simplest is to use a diagonal approximation, which takes $O(P)$

³⁵
³⁶ 1. In the Gaussian case, this term becomes $\nabla_{\mathbf{f}} \|\mathbf{y} - \mathbf{f}\|^2 = 2\|\mathbf{y} - \mathbf{f}\|$, so can be interpreted as a residual error.

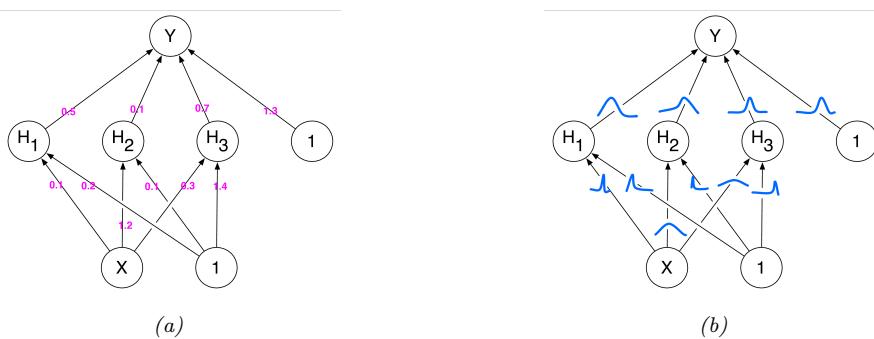


Figure 17.2: Illustration of an MLP with (a) point estimate for each weight, (b) a marginal distribution for each weight, corresponding to a fully factored posterior approximation. From Figure 1 of [Blu+15]. Used with kind permission of Charles Blundell.

time and space. A more sophisticated approach is presented in [RBB18a], which leverages the **KFAC** (Kronecker FActored Curvature) approximation of [MG15]. This approximates the covariance of each layer using a kronecker product. See Section 6.4.4 for details.

A limitation of the Laplace approximation is that the posterior covariance is derived from the Hessian evaluated at the MAP parameters. This means Laplace forms a highly *local* approximation: even if the non-Gaussian posterior could be well-described by a Gaussian distribution, the Gaussian distribution *formed using Laplace* only captures the local characteristics of the posterior at the MAP parameters — and may therefore suffer badly from local optima, providing overly compact or diffuse representations. In addition, the curvature information is only used after the model has been estimated, and not during the model optimization process. By contrast, variational inference (Section 17.4.2) can provide more accurate approximations for comparable cost.

17.4.2 Variational inference

In fixed-form variational inference (Section 10.3), we choose a distribution for the posterior approximation $q(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$, and minimize $D_{\text{KL}}(q\|p)$, with respect to $\boldsymbol{\theta}$. We often choose a Gaussian approximate posterior, $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, which lets us use the reparameterization trick to create a low variance estimator of the gradient of the ELBO (see Section 10.3.3). Despite the use of a Gaussian, the parameters that minimize the KL objective are often different what we would find with the Laplace approximation (Section 17.4.1).

Variational methods for neural networks date back to at least Hinton and Camp [HC93]. In deep learning, [Gra11] revisited variational methods, using a Gaussian approximation with a diagonal covariance matrix. This approximates the distribution of every parameter in the model by a univariate Gaussian, where the mean is the point estimate, and the variance captures the uncertainty, as shown in Figure 17.2. This approach was improved further in [Blu+15], who used the reparameterization trick to compute lower variance estimates of the ELBO; they called their method **Bayes by backprop** (**BBB**).

In [Blu+15], they used a diagonal Gaussian posterior and vanilla SGD. In [Osa+19a], they used the **variational online Gauss-Newton** (**VOGN**) method of [Kha+18], for improved scalability.

¹ VOGN is a noisy version of natural gradient descent, where the extra noise emulates the effect
² of variational inference. In [Mis+18], they replaced the diagonal approximation with a low-rank
³ plus diagonal approximation, and used VOGN for fitting. In [Tra+20b], they use a rank-one plus
⁴ diagonal approximation known as **NAGVAC** (see Section 10.3.4.2). In this case, there are only 3
⁵ times as many parameters as when computing a point estimate (for the variational mean, variance,
⁶ and rank-one vector), making the approach very scalable. In addition, in this case it is possible to
⁷ analytically compute the natural gradient, which speeds up model fitting. Many other variational
⁸ methods have also been proposed (see e.g., [LW16; Zha+18a; Wu+19a; HHK19]).

⁹ Note that VI often results in unnecessary weights being set to their prior value, in order to avoid
¹⁰ the KL penalty. This can be useful for inducing **sparsity** in the model [LUW17; MAV17]. However,
¹¹ it can also result in overconfidence, due to the **variational over-pruning** effect (Section 22.4), as
¹² discussed in [TT17].

¹³

¹⁴ 17.4.3 Expectation propagation

¹⁵ Expectation propagation is similar to variational inference, except it locally optimizes $D_{\text{KL}}(p\|q)$
¹⁶ instead of $D_{\text{KL}}(q\|p)$, where p is the exact posterior and q is the approximate posterior. For details,
¹⁷ see Section 10.7.

¹⁸ In [HLA15b], they show how to apply EP to fitting BNNs; they called their method **probabilistic**
¹⁹ **backpropagation** or **PBP**. They approximate every parameter in the model by a Gaussian factor,
²⁰ as in Figure 17.2. This work was extended to the classification setting in [BPK18]. One of the major
²¹ technical challenges is analytically propagating Gaussian densities through various nonlinear layers,
²² such as ReLU and softmax, without resorting to sampling. We discuss this in Section 17.6.2.

²³ In [HL+16a], they discussed black-box α -divergence minimization, which generalizes both VI
²⁴ ($\alpha = 0$) and EP ($\alpha = 1$). However, black-box techniques tend to be less efficient.

²⁵

²⁶ 17.4.4 Last layer methods

²⁷ Another scalable approximation is to only “be Bayesian” about the weights in the final layer, and to
²⁸ use MAP estimates for all the other parameters. This is called the **neural-linear** approximation
²⁹ (see Section 17.4.4). In [KHH20] they show this can reduce overconfidence in predictions for inputs
³⁰ that are far from the training data. However, this approach ignores uncertainty introduced by the
³¹ earlier feature extraction layers, where most of the parameters reside.

³² Earlier work on deep kernel learning [Wil+16b; Wil+16a] replaces the final linear layer with
³³ a Gaussian process. Building on this work, the **SNGP** (spectrally normalized Gaussian process)
³⁴ method of [Liu+20d] additionally constrains the feature extraction layers to be “distance preserving”,
³⁵ so that two inputs that are far apart in input space remain far apart after many layers of feature
³⁶ extraction. (This constraint is enforced using spectral normalization of the weights to bound the
³⁷ Lipschitz constant of the feature extractor.) The overall approach ensures that information that is
³⁸ relevant for computing the confidence of a prediction, but which might be irrelevant to computing
³⁹ the label of a prediction, is not lost. This can help performance in tasks such as out-of-distribution
⁴⁰ detection (Section 20.4.2).

⁴¹

⁴² 17.4.5 Dropout

⁴³ ⁴⁴ Monte Carlo dropout [GG16; KG17] is a very simple, and therefore popular, method for approx-
⁴⁵
⁴⁶

imating the Bayesian predictive distribution. The idea is to add dropout layers to the model, as described in Section 16.2.6, and then train in the usual way.

At test-time, we drop out each hidden unit by sampling from a Bernoulli(p) distribution; we repeat this procedure S times, to create S distinct models. We then create an equally weighted average of the predictive distributions for each of these models. Although it has been argued that this process approximates variational inference [GG16], this is only true under a degenerate posterior approximation, corresponding to a mixture of two delta functions, one at 0 (for dropped out nodes) and one at the MLE. This posterior will not converge to the true posterior (which is a delta function at the MLE) even as the training set size goes to infinity, since we are always dropping out hidden nodes with a constant probability p . Thus the procedure is not “truly Bayesian” [Osb16; HGMG18; NHLS19; LF+21]. Fortunately this pathology can be fixed if the noise rate is optimized [GHK17].

17.4.6 MCMC methods

Markov-chain Monte Carlo methods (Section 12.2) are particularly suitable for exploring the sophisticated multi-modal posteriors in Bayesian neural networks. Without the unimodal or strong parametric constraints of standard deterministic approximations, such as variational inference, Radford Neal’s early work [Nea96] established Hamiltonian Monte Carlo (Section 12.5) as a gold standard for Bayesian inference in neural networks [Nea+11; Izm+21b; CJ21].

However, a significant limitation of standard MCMC procedures, including HMC, is that they require access to the full training set at each step. Stochastic gradient MCMC methods operate instead using mini-batches of data, offering a scalable alternative [Wel11; CFG14b; Zha+20d]. See Section 12.7 for details.

17.4.7 Methods based on the SGD trajectory

In [MHB17], it was shown that, under some assumptions, the iterates produced by stochastic gradient descent, when run at a fixed learning rate, correspond to samples from a Gaussian approximation to the posterior centered at a local mode, $p(\theta|\mathcal{D}) \approx \mathcal{N}(\theta|\hat{\theta}, \Sigma)$. We can therefore use SGD to generate approximate posterior samples, similarly to SG-MCMC methods, except without explicit gradient noise, and the learning rate is held constant.

In [Izm+18], they noted that these SGD solutions (with fixed learning rate) surround the periphery of points of good generalization, as shown in Figure 17.3. This is in part because SGD does not converge to a local optimum unless the learning rate is annealed to 0. They therefore proposed to compute the average of several SGD samples, each one collected after a certain interval (e.g., one epoch of training), to get $\bar{\theta} = \frac{1}{S} \sum_{s=1}^S \theta_s$. They call this **stochastic weight averaging (SWA)**. They showed that the resulting point tends to correspond to a broader local minimum than the SGD solutions (c.f., Figure 17.7), resulting in better generalization performance.

The SWA approach is related to Polyak-Ruppert averaging, which is often used in convex optimization. The difference is that Polyak-Ruppert typically assumes the learning rate decays to zero, and uses an exponential moving average (EMA) of iterates, rather than an equal average; Polyak-Ruppert averaging is mainly used to reduce variance in the SGD estimate, rather than as a method to find points of better generalization.

The SWA approach is also related to **snapshot ensembles** [Hua+17a], and **fast geometric ensembles** [Gar+18c]; these methods save the parameters θ_s after increasing and decreasing

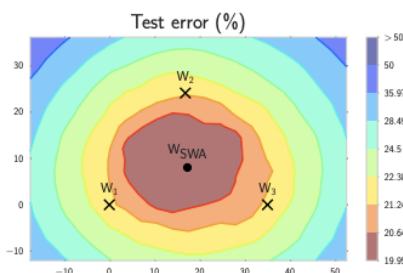


Figure 17.3: Illustration of stochastic weight averaging (SWA). The three crosses represent different SGD solutions. The star in the middle is the average of these parameter values. From Figure 1 of [Izm+18]. Used with kind permission of Andrew Wilson.

the learning rate multiple times in a cyclical fashion, and then average the predictions using $p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_s)$, rather than averaging the parameters and predicting with a single model (which is faster). Moreover, by finding a flat region, representing a “center or mass” in the posterior, SWA can be seen as approximating the Bayesian model average in Equation 17.1 with a single model.

In [Mad+19], they proposed to fit a Gaussian distribution to the set of samples produced by SGD near a local mode. They use the SWA solution as the mean of the Gaussian. For the covariance matrix, they use a low-rank plus diagonal approximation of the form $p(\boldsymbol{\theta}|\mathcal{D}) = \mathcal{N}(\boldsymbol{\theta}|\bar{\boldsymbol{\theta}}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_{\text{diag}} + \boldsymbol{\Sigma}_{\text{lr}})/2$, $\boldsymbol{\Sigma}_{\text{diag}} = \text{diag}(\bar{\boldsymbol{\theta}}^2 - (\bar{\boldsymbol{\theta}})^2)$, $\bar{\boldsymbol{\theta}} = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s$, $\bar{\boldsymbol{\theta}}^2 = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s^2$, and $\boldsymbol{\Sigma}_{\text{lr}} = \frac{1}{S} \boldsymbol{\Delta} \boldsymbol{\Delta}^\top$ is the sample covariance matrix of the last K samples of $\boldsymbol{\Delta}_i = (\boldsymbol{\theta}_i - \bar{\boldsymbol{\theta}}_i)$, where $\bar{\boldsymbol{\theta}}_i$ is the running average of the parameters from the first i samples. They call this method **SWAG**, which stands for “stochastic weight averaging with Gaussian posterior”. This can be used to generate an arbitrary number of posterior samples at prediction time. They show that SWAG scales to large residual networks with millions of parameters, and large datasets such as ImageNet, with improved accuracy and calibration over conventional SGD training, and no additional training overhead.

17.4.8 Deep ensembles

Many conventional approximate inference methods focus on approximating the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ in a local neighborhood around one of the posterior modes. While this is often not a major limitation in classical machine learning, modern deep neural networks have highly multi-modal posteriors, with parameters in different modes giving rise to very different functions. On the other hand, the functions in a neighborhood of a single mode may make fairly similar predictions. So using such a local approximation to compute the posterior predictive will underestimate uncertainty and generalize more poorly.

A simple alternative method is to train multiple models, and then to approximate the posterior using an equally weighted mixture of delta functions,

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}_m) \quad (17.18)$$

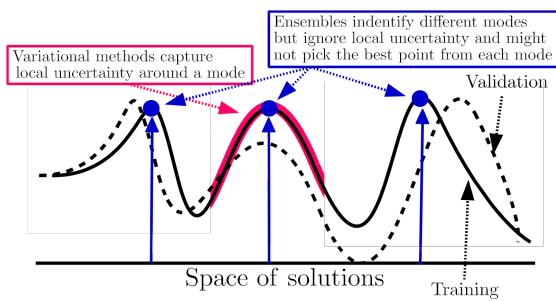


Figure 17.4: Cartoon illustration of the NLL as it varies across the parameter space. Subspace methods (red) model the local neighborhood around a local mode, whereas ensemble methods (blue) approximate the posterior using a set of distinct modes. From Figure 1 of [FHL19]. Used with kind permission of Balaji Lakshminarayanan.

where M is the number of models, and $\hat{\theta}_m$ is the MAP estimate for model m . See Figure 17.4 for a sketch. This approach is called **deep ensembles** [LPB17; FHL19].

The models can differ in terms of their random seed used for initialization [LPB17], or hyperparameters [Wen+20c], or architecture [Zai+20], or all of the above. Each local optima corresponds to a distinct prediction function, so combining these is more effective than combining multiple samples from the same basin of attraction, especially in the presence of dataset shift [Ova+19].

We can further improve on this approach by fitting a Gaussian to each local mode using the SWAG method from Section 17.4.7 to get a mixture of Gaussians approximation:

$$p(\theta|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \mathcal{N}(\theta|\hat{\theta}_m, \Sigma_m) \quad (17.19)$$

This approach is known as **MultiSWAG** [WI20]. MultiSWAG performs a Bayesian model average both across multiple basins of attraction, like deep ensembles, but also within each basin, and provides an easy way to generate an arbitrary number of posterior samples, $S > M$, in an any-time fashion.

17.4.8.1 Deep ensembles as approximate Bayesian inference

The posterior predictive distribution for a Bayesian neural network cannot be expressed in closed form. Therefore all Bayesian inference approaches in deep learning are approximate. In this context, all approximate inference procedures fall onto a spectrum, representing how closely they approximate the true posterior predictive distribution. On this spectrum, deep ensembles are often closer to the Bayesian ideal than many canonical approximate Bayesian inference procedures, such as the Laplace approximation, in deep learning. By also marginalizing within basins, MultiSWAG moves deep ensembles further towards the Bayesian predictive distribution.

In short, deep ensembles can provide better approximations to a Bayesian model average than a single basin marginalization approach, because point masses from different basins of attraction represent greater functional diversity than standard Bayesian approaches which sample within a single basin. This result is illustrated in Figure 17.5.

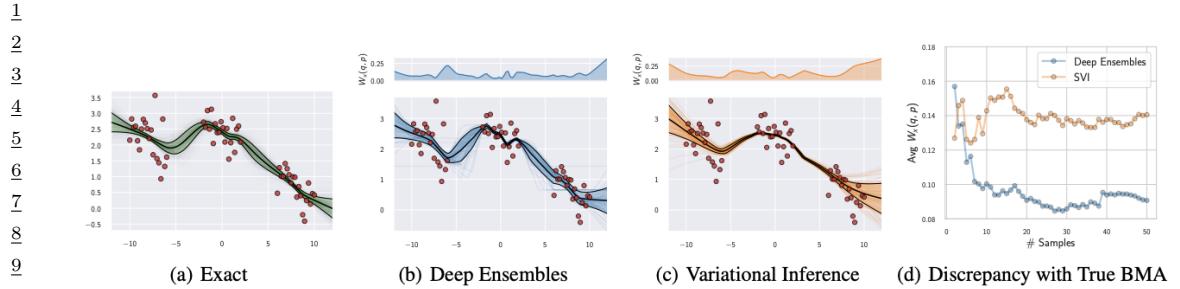


Figure 17.5: Comparison of approximate Bayesian inference methods for a 1d regression problem. (a) The “true” predictive distribution obtained by combining 200 HMC chains. (b) Deep ensembles predictive distribution using 50 independently trained networks. (c) Predictive distribution for factorized variational inference (VI). (d) Convergence of the predictive distributions for deep ensembles and variational inference as a function of the number of samples in terms of the average Wasserstein distance between the marginals in the range of input positions. The multi-basin deep ensembles approach provides a better approximation of the Bayesian predictive distribution than the conventional single-basin VI approach, which is overconfident between data clusters. The top panels show the Wasserstein distance between the true predictive distribution and the deep ensemble and VI approximations, as a function of inputs x . From Figure 4 of [WI20]. Used with kind permission of Andrew Wilson.

Note that deep ensembles is slightly different to a classical ensemble method, such as bagging and random forests, which obtains diversity of its predictors by training them on different subsets of the data (created using bootstrap resampling), or on different features. This data perturbation is necessary to get diversity when the base learner is a convex problem (such as a linear model, or shallow decision tree). In the deep ensemble approach, every model is trained on the same data, and the same features. The diversity arises due to different starting parameters, different random seeds, and SGD noise, which induces different solutions due to the nonconvex loss.

If we use weighted combinations of the models, $p(\boldsymbol{\theta}|\mathcal{D}) = \sum_{m=1}^M p(m|\mathcal{D})p(\boldsymbol{\theta}|m, \mathcal{D})$, where $p(m|\mathcal{D})$ is the marginal likelihood of model m , then, in the large sample limit, this mixture will concentrate on the MAP model, so only one component will be selected. This is because Bayes model averaging is not the same as model ensembling [Min00b], which fixes or optimizes the mixing weights; this can enlarge the expressive power of the posterior predictive distribution compared to BMA [OCM21].

We can also make the mixing weights be conditional on the inputs:

$$p(y|\mathbf{x}, \mathcal{D}) = \sum_m w_m(\mathbf{x})p(y|\mathbf{x}, \boldsymbol{\theta}_m) \quad (17.20)$$

If we constrain the weights to be non-zero and sum to one, this is called a **mixture of experts**. However, if we allow a general positive weighted combination, the approach is called **stacking** [Wol92; Bre96; Yao+18a; CAII20]. In stacking, the weights $w_m(\mathbf{x})$ are usually estimated on hold-out data, to make the method more robust to model misspecification.

47

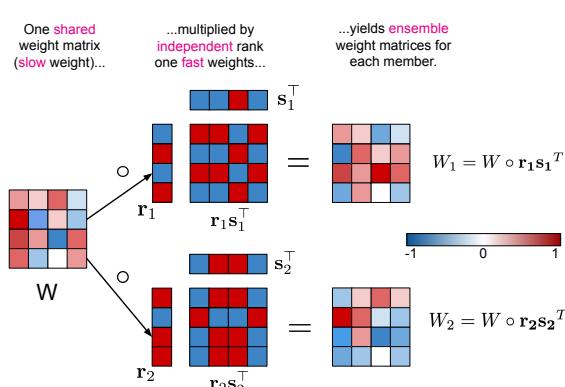


Figure 17.6: Illustration of batch ensemble with 2 ensemble members. From Figure 2 of [WTB20]. Used with kind permission of Paul Vicol.

17.4.8.2 Batch ensemble

Deep ensembles require M times more memory and time than a single model. One way to reduce the memory cost is to share most of the parameters — which we call **slow weights**, \mathbf{W} — and then let each ensemble member m estimate its own local perturbation, which we will call **fast weights**, \mathbf{F}_m . We then define $\mathbf{W}_m = \mathbf{W} \odot \mathbf{F}_m$. For efficiency, we can define \mathbf{F}_m to be a rank-one matrix, $\mathbf{F}_m = \mathbf{s}_m \mathbf{r}_m^\top$, as illustrated in Figure 17.6. This is called **batch ensemble** [WTB20].

It is clear that the memory overhead is very small compared to naive ensembles, since we just need to store $2M$ vectors (\mathbf{s}_m^l and \mathbf{r}_m^l) for every layer, which is negligible compared to the quadratic cost of storing the shared weight matrix \mathbf{W}^l .

In addition to memory savings, batch ensemble can reduce the inference time by a constant factor by leveraging within-device parallelism. To see this, consider the output of one layer using ensemble m on example n :

$$y_n^m = \varphi(\mathbf{W}_m^\top \mathbf{x}_n) = \varphi((\mathbf{W} \odot \mathbf{s}_m \mathbf{r}_m^\top)^\top \mathbf{x}_n) = \varphi((\mathbf{W}^\top (\mathbf{x}_n \odot \mathbf{s}_m) \odot \mathbf{r}_m)) \quad (17.21)$$

We can vectorize this for a minibatch of inputs \mathbf{X} by repeating \mathbf{r}_m and \mathbf{s}_m into matrices, to get

$$\mathbf{Y}_m = \varphi(((\mathbf{X} \odot \mathbf{S}_m) \mathbf{W}) \odot \mathbf{R}_m) \quad (17.22)$$

This applies the same ensemble parameters m to every example in the minibatch of size B . To achieve diversity during training, we can divide the minibatch into M sub-batches, and use sub-batch m to train \mathbf{W}_m . (Note that this reduces the batch size for training each ensemble to B/M .) At test time, when we want to average over M models, we can replicate each input M times, leading to a batch size of BM .

In [WTB20], they show that this method outperforms MC dropout at negligible extra memory cost. However, the best combination was to combine batch ensemble with MC dropout; in some cases, this approached the performance of naive ensembles.

1 **17.4.9 Approximating the posterior predictive distribution**

3 Once we have approximated the parameter posterior, $q(\boldsymbol{\theta}) \approx p(\boldsymbol{\theta}|\mathcal{D})$, we can use it to approximate
4 the posterior predictive distribution:

5

$$\underline{6} \quad p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int q(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) d\boldsymbol{\theta} \quad (17.23)$$

7

8 We usually approximate this integral using Monte Carlo:

9

$$\underline{10} \quad p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^s)) \quad (17.24)$$

11

12 where $\boldsymbol{\theta}^s \sim q(\boldsymbol{\theta})$. We discuss some extensions of this approach below.

14

15 **17.4.9.1 A linearized approximation**

16 In [IKB21] they point out that samples from an approximate posterior, $q(\boldsymbol{\theta})$, can result in bad
17 predictions when plugged into the model if the posterior puts probability density “in the wrong
18 places”. This is because $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ is a highly nonlinear function of $\boldsymbol{\theta}$ that might behave quite differently
19 when $\boldsymbol{\theta}$ is far from the MAP estimate on which $q(\boldsymbol{\theta})$ is centered. To avoid this problem, they propose
20 to replace $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ with a linear approximation centered at the MAP estimate $\boldsymbol{\theta}^*$:

21

$$\underline{22} \quad \mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^*) + \mathbf{J}_{\boldsymbol{\theta}^*}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \quad (17.25)$$

23

24 Such a model is well behaved around $\boldsymbol{\theta}^*$, and so the approximation

25

$$\underline{26} \quad p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}^s)) \quad (17.26)$$

27

28 often works better than Equation (17.24).

29 Note that $\mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta})$ is a linear function of the parameters $\boldsymbol{\theta}$, but a nonlinear function of the
30 inputs \mathbf{x} . Thus $p(\mathbf{y}|\mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}))$ is a generalized linear model (Section 15.1), so [IKB21] call this
31 approximation the **GLM predictive distribution**.

32

33 **17.4.9.2 Distillation**

34 The MC approximation to the posterior predictive is S times slower than a standard, determin-
35 istic plug-in approximation. One way to speed this up is to use **distillation** to approximate the
36 semi-parametric “teacher” model p_t from Equation (17.24) by a parametric “student” model p_s
37 by minimizing $\mathbb{E}[D_{\text{KL}}(p_t(\mathbf{y}|\mathbf{x}) \| p_s(\mathbf{y}|\mathbf{x}))]$ wrt p_s . This approach was first proposed in [HVD14],
38 who called the technique “**dark knowledge**”, because the teacher has “hidden” information in its
39 predictive probabilities (logits) than is not apparent in the raw one-hot labels.

40 In [Kor+15], this idea was used to distill the predictions from a teacher whose parameter posterior
41 was computed using HMC; this is called “**Bayesian dark knowledge**”. A similar idea was used in
42 [BP16], who distilled the predictive distribution derived from MC dropout (Section 17.4.5).

43 Since the parametric student is typically less flexible than the semi-parametric teacher, it may be
44 overconfident, and lack diversity in its predictions. To avoid this overconfidence, it is safer to make
45 the student be a mixture distribution c.f., [SG05]. See also [Tra+20a].

46

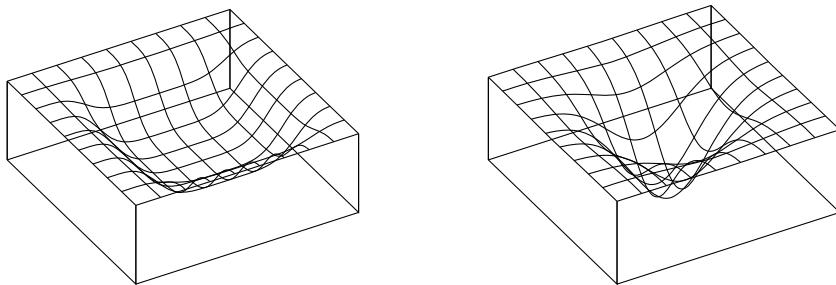


Figure 17.7: Flat vs sharp minima. From Figures 1 and 2 of [HS97]. Used with kind permission of Jürgen Schmidhuber.

17.5 Generalization in Bayesian deep learning

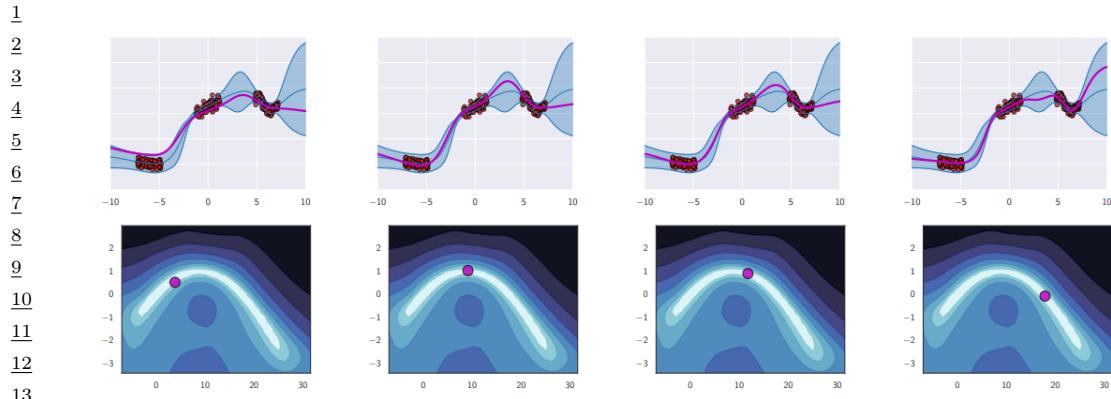
In this section, we discuss why “being Bayesian” can improve predictive accuracy and generalization performance.

17.5.1 Sharp vs flat minima

Some optimization methods (in particular, second-order batch methods) are able to find “needles in haystacks”, corresponding to narrow but deep “holes” in the loss landscape, corresponding to parameter settings with very low loss. These are known as **sharp minima**, see Figure 17.7(right). From the point of view of minimizing the empirical loss, the optimizer has done a good job. However, such solutions generally correspond to a model that has overfit the data. It is better to find points that correspond to **flat minima**, as shown in Figure 17.7(left); such solutions are more robust and generalize better. To see why, note that flat minima correspond to regions in parameter space where there is a lot of posterior uncertainty, and hence samples from this region are less able to precisely memorize irrelevant details about the training set [AS17]. Put another way, the description length for sharp minima is large, meaning you need to use many bits of precision to specify the exact location in parameter space to avoid incurring large loss, whereas the description length for flat minima is less, resulting in better generalization [Mac03].

SGD often finds such flat minima by virtue of the addition of noise, which prevents it from “entering” narrow regions of the loss landscape (see Section 12.5.7). In addition, in higher dimensional spaces, flat regions occupy a much greater volume, and are thus much more easily discoverable by optimization procedures. More precisely, the analysis in [SL18] shows that the probability of entering any given basin of attraction \mathcal{A} around a minimum is given by $p_{SGD}(\boldsymbol{\theta} \in \mathcal{A}) \propto \int_{\mathcal{A}} e^{-\mathcal{L}(\boldsymbol{\theta})} d\boldsymbol{\theta}$. Note that this is integrating over the volume of space corresponding to \mathcal{A} , and hence is proportional to the model evidence (marginal likelihood) for that region, as explained in Section 3.7.1. Since the evidence is parameterization invariant (since we marginalize out the parameters), this means that SGD will avoid regions that have low evidence (corresponding to sharp minima) regardless of how we parameterize the model (contrary to the claims in [Din+17]).

In fact, several papers have shown that we can view SGD as approximately sampling from the Bayesian posterior (see e.g., [MHB17; SL18; CS18]). The SWA [Izm+18] method can be seen as



14 *Figure 17.8: Diversity of high performing functions sampled from the posterior. Top row: we show predictions*
15 *on the 1d input domain for 4 different functions. We see that they extrapolate in different ways outside of the*
16 *support of the data. Bottom row: we show a 2d subspace spanning two distinct modes (MAP estimates), and*
17 *connected by a low-loss curved path computed as in [Gar+18c]. From Figure 8 of [WI20]. Used with kind*
18 *permission of Andrew Wilson.*

192021

22 finding a center of mass in the posterior, finding flatter solutions than SGD that generalize better
23 than SGD.

24 If we must use a single solution, a flat one will help us better approximate the Bayesian model
25 average in the integral of Equation (17.1). However, by attempting to perform a more complete
26 Bayesian model average, we will select for flatness without having to deal with the messiness of
27 having to worry about flatness definitions, or the effects of reparametrization, or unknown implicit
28 regularization, as the model average will automatically weight regions with the greatest volume.

29 Moreover, we can do much better than a *single* flat solution. Indeed, there are many low-loss
30 solutions, which provide complementary explanations of the data. In [Gar+18c], they showed that
31 two independently trained SGD solutions can be connected by a curve in a subspace, along which the
32 training loss remains near-zero, known as **mode connectivity**. Despite having the same training
33 loss, these different parameter settings give rise to very different functions, as illustrated in Figure 17.8,
34 where we show predictions on a 1d regression problem coming from different points in parameter
35 space obtained by interpolating along a mode connecting curve between two distinct MAP estimates.
36 Using a Bayesian model average, we can combine these functions together to provide much better
37 performance over a single flat solution [Izm+19].

38 Recently, it has been discovered [Ben+21] that there are in fact large multidimensional simplexes
39 of low loss solutions, which can be combined together for significantly improved performance. These
40 results further motivate the Bayesian approach (Equation (17.1)), where we perform a posterior
41 weighted model average.

42

43 17.5.2 Effective dimensionality of a model

44

45 Modern DNNs have millions of parameters, but these parameters are often not well-determined
46 by the data, i.e., there can be a lot of posterior uncertainty. By averaging over the posterior, we
47

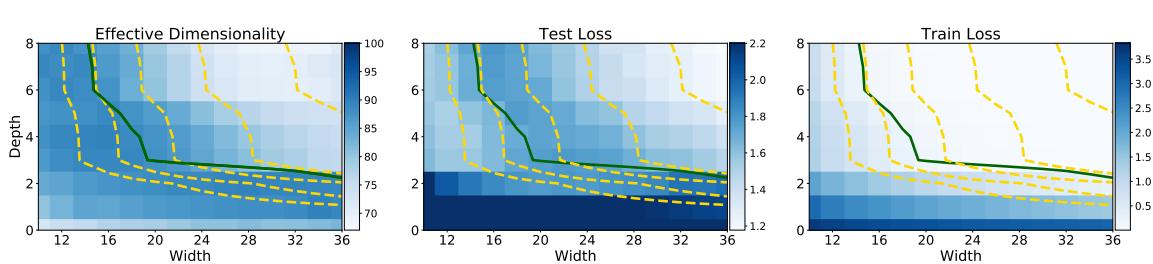


Figure 17.9: Left: Effective dimensionality as a function of model width and depth for a CNN on CIFAR-100. Center: Test loss as a function of model width and depth. Right: Train loss as a function of model width and depth. Yellow level curves represent equal parameter counts ($1e5$, $2e5$, $4e5$, $1.6e6$). The green curve separates models with near-zero training loss. Effective dimensionality serves as a good proxy for generalization for models with low train loss. We see wide but shallow models overfit, providing low train loss, but high test loss and high effective dimensionality. For models with the same train loss, lower effective dimensionality can be viewed as a better compression of the data at the same fidelity. Thus depth provides a mechanism for compression, which leads to better generalization. From Figure 2 of [MBW20]. Used with kind permission of Andrew Wilson.

reduce the chance of overfitting, because we do not use “degrees of freedom” that are not needed or warranted.

In [MBW20], they revisit the **effective dimensionality** [Mac92b] of a model, shedding light on overparametrization, double descent, and width-depth trade-offs. The effective dimensionality is defined as

$$N_{\text{eff}}(\mathbf{H}, c) = \sum_{i=1}^k \frac{\lambda_i}{\lambda_i + c}, \quad (17.27)$$

where λ_i are the eigenvalues of the Hessian matrix \mathbf{H} computed at a local mode, and $c > 0$ is a regularization parameter. Intuitively, the effective dimension counts the number of well-determined parameters. A “flat minimum” will have many directions in parameter space that are not well-determined, and hence will have low effective dimensionality. This means that we can perform Bayesian inference in a low dimensional subspace [Izm+19]: Since there is functional homogeneity in all directions but those defining the effective dimension, neural networks can be significantly compressed.

This compression perspective can also be used to understand why the effective dimension can be a good proxy for generalization. If two models have similar training loss, but one has lower effective dimension, then it is providing a better compression for the data at the same fidelity. In Figure 17.9 we show that for CNNs with low training loss (above the green partition), the effective dimensionality closely tracks generalization performance. We also see that the number of parameters alone is not a strong determinant of generalization. Indeed, models with more parameters can have a lower number of effective parameters. We also see that wide but shallow models overfit, while depth helps provide lower effective dimensionality, leading to a better compression of the data. It is depth that makes modern neural networks distinctive, providing hierarchical inductive biases making it possible to discover more regularity in the data.

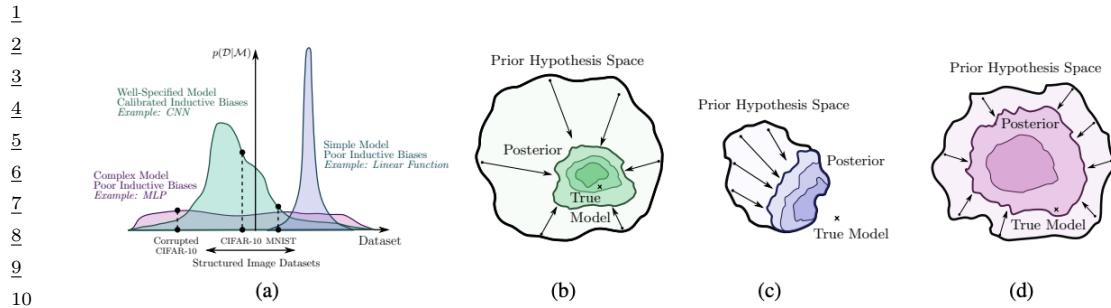


Figure 17.10: Illustration of the behavior of different kinds of model families and the prior distribution they induce over datasets. (a) The purple model is a simple linear model that has small support, and can only represent a few kinds of datasets. The pink model is an unstructured MLP: this has support over a large range of datasets but with a fairly uninformative (broad) prior. Finally the green model is a CNN; this has support over a large range of datasets but with a fairly uninformative (broad) prior. (b) The posterior for the green model (CNN) rapidly collapses to the true model. (c) The posterior for the purple model (linear) also rapidly collapses, but to a solution which cannot represent the true model. (d) The posterior for the pink model (MLP) collapses very slowly (as a function of dataset size). From Figure 2 of [WI20]. Used with kind permission of Andrew Wilson.

20
21
22
23

17.5.3 The hypothesis space of DNNs

Zhang et al. [Zha+17] showed that CNNs can fit CIFAR-10 images with random labels with zero training error, but can still generalize well on the noise-free test set. It has been claimed that this result contradicts a classical understanding of generalization, because it shows that neural networks are capable of significantly overfitting the data, but can still generalize well on structured inputs.

We can resolve this paradox by taking a Bayesian perspective. In particular, we know that modern CNNs are very flexible, so they can fit almost pattern (since they are in fact universal approximators). However, their architecture encodes a prior over what kinds of patterns they expect to see in the data (see Section 17.2.5). Image datasets with random labels *can* be represented by this function class, but such solutions receive very low marginal likelihood, since they strongly violate the prior assumptions [WI20]. By contrast, image datasets where the output labels are consistent with patterns in the input get much higher marginal likelihood.

This phenomenon is not unique to DNNs. For example, it also occurs with Gaussian processes (Chapter 18). Such models are also universal approximators, but they allocate most of their probability mass to a small range of solutions (depending on the chosen kernel). They can also fit image datasets with random labels, but such data receives a low marginal likelihood [WI20].

In general, we can distinguish the support of a model, i.e., the set of functions it can represent, from the distribution over that support, i.e., the inductive bias which leads it to prefer some functions over others. We would like to use models where the support is large, so we can capture the complexity of real-world data, but also where the inductive bias places probability mass on the kinds of functions we expect to see. If we succeed at this, the posterior will quickly converge on the true function after seeing a small amount of data. This idea is sketched in Figure 17.10.

47

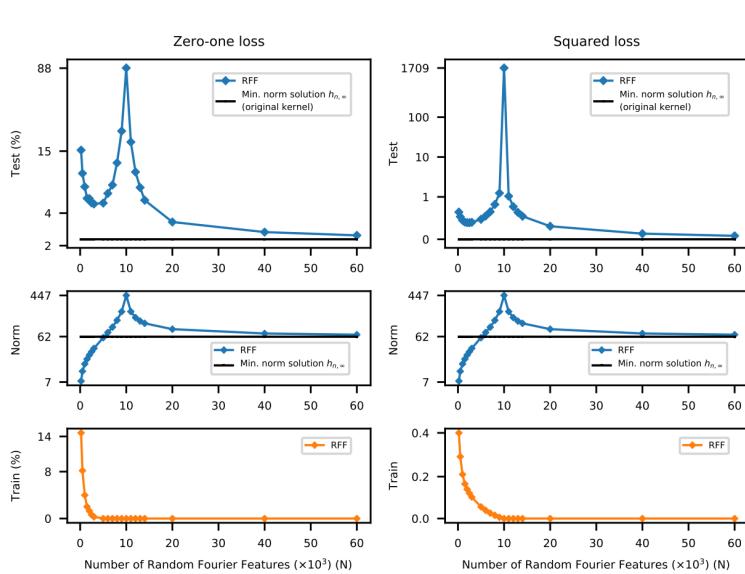


Figure 17.11: Illustration of the double descent risk curve for RFF model on a subset of MNIST. There are $N = 10^4$ examples, so interpolation occurs when the x -axis (which is number of features times 1000) equals 10. Top row: test error. Middle row: norm of \mathbf{w}^* . Bottom row: train error. From Figure 2 of [Bel+19b]. Used with kind permission of Mikhail Belkin.

17.5.4 Double descent

Practitioners often use very wide and deep models, with many more parameters than training points. These are called **over-parameterized models**. Naively one might think such models would overfit. However, they often exhibit very good generalization performance.

For example, consider a linear function of the form $f(\mathbf{x}) = \sum_{k=1}^K w_k \phi_k(\mathbf{x})$, where the $\phi_k(\mathbf{x})$ are random Fourier features (Section 18.2.3.1). If $K > N$, then there are more parameters than data points. In this setting, there are multiple solutions that can achieve zero training error. Call this set of interpolators $\mathcal{W} = \{\mathbf{w} : \text{RSS}(\mathbf{w}) = 0\}$. We pick the “simplest” solution in this set, which we define to be the one with least norm: $\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w} \in \mathcal{W}} \|\mathbf{w}\|_2$.²

In Figure 17.11, we show the results of this method when applied to a subset of MNIST.³ As K increases, the test error goes down as underfitting is eliminated. As K approaches N , test error rises dramatically, as overfitting kicks in. However, when K exceeds N , test error goes down again, until it reaches the asymptotic performance. This is known as the **double descent risk curve**. We also

2. For linear models (with fixed basis functions ϕ), we can compute the minimum norm solution, by using the psuedo inverse, as explained in [Mur22, Sec 7.7.2]. For nonlinear models, it can be shown that gradient descent often converges to the minimum norm solution when started from $\mathbf{w} = \mathbf{0}$ (see e.g., [Nac+19c]).

3. For simplicity, we treat this as a linear regression problem with C outputs (corresponding to the one-hot class labels), rather than a logistic regression problem. This is known as “**least squares classification**”, and is known to work well if we are only interested in predicting the most probable class, (i.e., in minimizing zero-one loss), rather than computing a probability distribution [RK04].

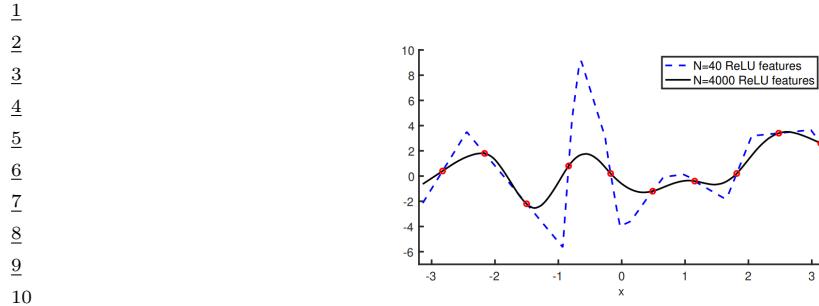


Figure 17.12: Two shallow MLPs fit to $N = 10$ data points (shown in red) using one layer of $K = 40$ or $K = 4000$ random ReLU features, i.e., the model has the form $h(x) = w_0 + \sum_{k=1}^K w_k \phi(x; \mathbf{v}_k)$, where $\phi(x; \mathbf{v}) = \max(v_0 + v_1 x, 0)$, \mathbf{v}_k are chosen randomly and \mathbf{w} is optimized. The piecewise linear nature of the fitted function is visually apparent when $K = 40$, but the function is smoother with larger K . The scaled norm of the parameter vectors, $\frac{\|\mathbf{w}\|_2}{\sqrt{K}}$, is 695 and 159 for $K = 40$ and $K = 4000$, reflecting the fact that the latter model is simpler. (Similar results hold when we optimize both \mathbf{V} and \mathbf{w} , but it is harder to implement the minimum norm solution in this case.) From Figure 3 of [Bel+19b]. Used with kind permission of Mikhail Belkin.

19

20

21 plot the norm of the optimal vector as we increase K , and we see that the over-parameterized models
22 are simpler.

23 The “spike” at the interpolation threshold, when number of parameters is equal to the number of
24 datapoints, is due to the condition number of the design matrix \mathbf{X} going to infinity. If we add some
25 ℓ_2 regularization (or increase the “label noise”) this spike can be eliminated (see [Adl] for a detailed
26 analysis). Nevertheless performance still continues to improve as the (regularized) model becomes
27 more overparameterized, even when there is no such spike, due to the increasing simplicity of the
28 model.

29 Note that this phenomenon is not specific to this kind of model or data — it occurs with many
30 kinds of models and datasets [Bel+19b; Nak+19]. For example, Figure 17.12 compares two one-layer
31 MLPs fit to $N = 10$ points from a 1d regression problem. One model has $K = 40$ hidden units,
32 and one has $K = 4000$ hidden units. Both perfectly interpolate the training data, but the latter is
33 smoother.

34 A cartoon summarization of the situation is shown in Figure 17.13. On the left we show the classic
35 U-shaped curve, where increasing the model capacity too much results in overfitting. On the right, we
36 show the situation observed above, where with overparameterized models, adding more parameters
37 actually helps generalization performance.

38 A theoretical explanation for this phenomenon is given in [BS21]. Their result, roughly speaking,
39 is that if you want to interpolate the training data of N data points each of D dimensions using a
40 smooth function, then you need at least ND parameters. So by over-parameterizing the solution, we
41 can not just interpolate, but interpolate smoothly.

42

43 17.5.5 A Bayesian Resolution to Double Descent

44

45 We have argued that we wish to build models that have large support — and thus great flexibility —
46 but with a reasonable prior (largely induced by the neural architecture). From a Bayesian perspective,
47

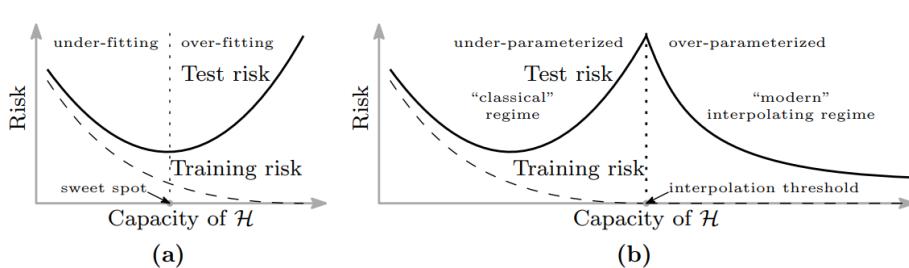


Figure 17.13: Curves for training risk (dashed line) and test risk (solid line) as a function of the size of the hypothesis space \mathcal{H} for the unknown function f . (a) Classical U-shaped curve in the under-parameterized regime. (b) Illustration of the double descent risk curve that arises when we consider models that may be over-parameterized. From Figure 1 of [Bel+19b]. Used with kind permission of Mikhail Belkin.

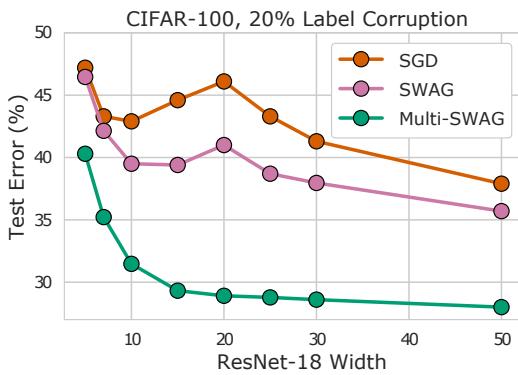


Figure 17.14: A ResNet-18 with layers of varying width trained on CIFAR-100 with 20% label corruption. Standard SGD training leads to significant double descent behavior. SWAG [Mad+19], which performs a single basin Bayesian model average, helps mitigate double descent. Multi-SWAG [WI20], which performs exhaustive multi-basin marginalization, entirely alleviates double descent. There is also a dramatic performance improvement, even for accuracy of point predictions, between Multi-SWAG and classical SGD training. From Figure 8 of [WI20]. Used with kind permission of Andrew Wilson.

one would expect that, with exhaustive Bayesian model averaging, performance should improve monotonically with model flexibility and we wouldn't observe double descent.

We indeed observe in Figure 17.14 that exhaustive Bayesian model averaging from Multi-SWAG (Section 17.4.8), which forms a mixture of Gaussians approximation to the posterior, entirely alleviates double descent. Multi-SWAG also leads to significant performance improvements over classical SGD training, even in terms of accuracy of point predictions. For example, for the ResNet-18 with layers of width 20 on CIFAR-100 with 20% label corruption, classical training achieves over 45% error, while Multi-SWAG has under 30% error.

We can also gain insights into why double descent occurs in classical training from a Bayesian perspective. In Section 17.5 we discussed how the effective dimensionality of the Hessian is proportional

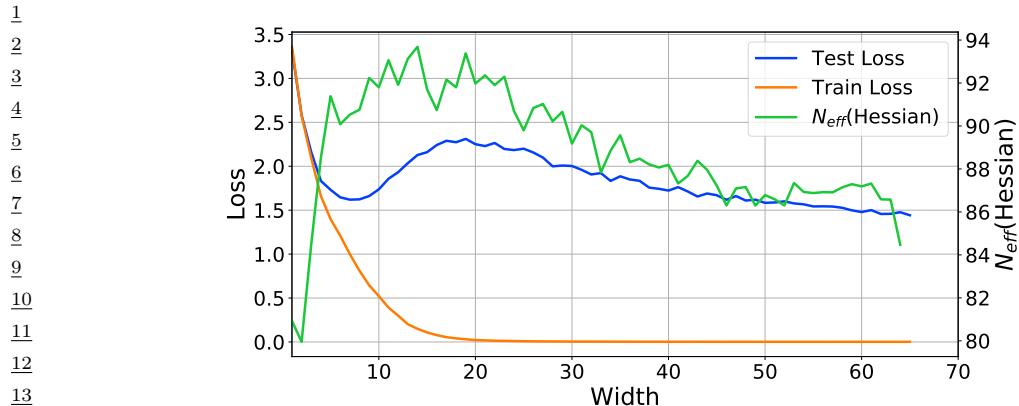


Figure 17.15: A resolution of double descent. We replicate the double descent behaviour of deep neural networks using a ResNet-18 on CIFAR-100, where train loss decreases to zero with sufficiently wide model while test loss decreases, then increases, and then decreases again. Unlike model width, the effective dimensionality computed from the eigenspectrum of the Hessian of the loss on training data alone follows the test loss in the overparameterized regime, acting as a much better proxy for generalization than naive parameter counting.

From Figure 1 of [MBW20]. Used with kind permission of Andrew Wilson.

to posterior contraction in a Bayesian neural network. In Figure 17.15, the models with no training loss can all be viewed as providing lossless compressions of the data. In this regime, we see that effective dimensionality closely tracks double descent, and that as model size increases, the effective number of parameters in fact decreases. In other words, the models with more parameters are providing a better compression of the data, corresponding to flatter regions of the loss surface. Better compressions can be found by discovering greater regularity explaining the data, and are thus more likely to generalize. From a Bayesian perspective of model selection, flatter solutions also incur less of an Occam factor penalty.

But why is SGD finding simpler solutions as we increase model size? As indicated by the effective dimensionality, these simpler solutions correspond to flatter regions of the loss surface. In higher dimensional spaces, flat regions occupy a much greater volume, and are thus much more easily discoverable by optimization procedures. Since flat solutions are associated with better generalization, we might call this behavior a “blessing of dimensionality” [Hua+19a; Izm+18].

17.5.6 PAC-Bayes

PAC-Bayes [McA99; LC02; Gue19; Alq21; GSZ21] provides a promising mechanism to derive non-vacuous generalization bounds for large *stochastic networks* [Ney+17; NBS18; DR17], with parameters sampled from a probability distribution. In particular, the difference between the train error and the generalization error can be expressed as

$$\sqrt{\frac{\text{KL}(Q||P) + c}{2(n-1)}}, \quad (17.28)$$

where c is a constant and n is the number of training points. P is the prior distribution over the

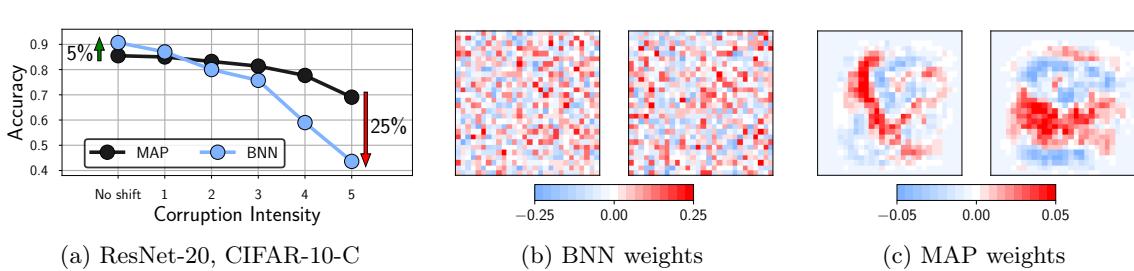


Figure 17.16: **Bayesian neural networks under covariate shift.** (a): Performance of a ResNet-20 on the pixelate corruption in CIFAR-10-C. For the highest degree of corruption, a Bayesian model average underperforms a MAP solution by 25% (44% against 69%) accuracy. See Izmailov et al. [Izm+21b] for details. (b): Visualization of the weights in the first layer of a Bayesian fully-connected network on MNIST sampled via HMC. (c): The corresponding MAP weights. We visualize the weights connecting the input pixels to a neuron in the hidden layer as a 28×28 image, where each weight is shown in the location of the input pixel it interacts with. This figure is adapted from Figure 1 of Izmailov et al. [Izm+21a].

parameters and Q is an arbitrary distribution, which can be chosen to optimize the bound.

The perspective in this chapter is largely complementary, and in some ways orthogonal, to the PAC-Bayes literature. Our focus has been on Bayesian marginalization, particularly multi-modal marginalization, and a prescriptive approach to model construction. In contrast, PAC-Bayes bounds are about bounding the empirical risk of a single sample, rather than marginalization, and are not currently prescriptive: what we would do to improve the bounds, such as reducing the number of model parameters, or using highly compact priors, does not typically improve generalization. Moreover, while we have seen Bayesian model averaging over multimodal posteriors has a significant effect on generalization, it has a minimal logarithmic effect on PAC-Bayes bounds. In general, because the bounds are lose, albeit non-vacuous in some cases, there is often room to make modeling choices that improve PAC-Bayes bounds without improving generalization, making it hard to derive a prescription for model construction from the bounds.

17.5.7 Out-of-Distribution Generalization for BNNs

Bayesian methods are often considered a robust alternative to classical training, because they represent many possible solutions to a given problem, corresponding to epistemic uncertainty. For this reason, Bayesian methods are often applied in out-of-distribution settings [Ova+19; KG17; Cha+19b; WI20; Dus+20; Ben+21]. There are two common objectives in this context: (1) to simplify detect a point as out-of-distribution and refuse to make a prediction; (2) to provide a reasonable predictive distribution for these points.

In many real-world applications we are in the second setting, where the training and test distributions are not exactly the same, but we still want to make a useful prediction: medical images taken from different devices, autonomous vehicles operating in different cities, recognizing typewritten characters from handwritten examples, and so on. Many standard out-of-distribution benchmarks, such as MNIST-C, CIFAR-10-C, ImageNet-C, and MNIST to SVHN, also represent this setting, with the test points still clearly recognizable from the training points, through noise corruptions, or mild domain shift. Approximate inference procedures have been providing increasingly good generalization

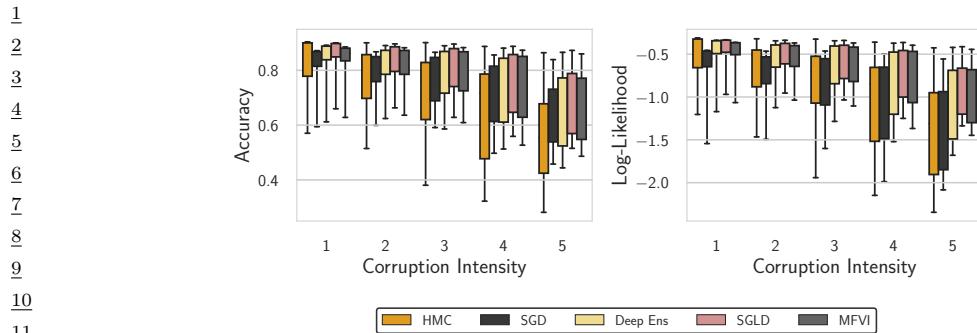


Figure 17.17: Evaluation on CIFAR-10-C. Accuracy and log-likelihood of HMC, SGD, deep ensembles, SGLD and MFVI on a distribution shift task, where the CIFAR-10 test set is corrupted in 16 different ways at various intensities on the scale of 1 to 5. We use the ResNet-20-FRN architecture. Boxes capture the quartiles of performance over each corruption, with the whiskers indicating the minimum and maximum. HMC is surprisingly the worst of the considered methods: even a single SGD solution provides better OOD robustness. This figure is adapted from Figure 7 of Izmailov et al. [Izm+21b].

18

19

accuracy on such benchmarks [Mil+21a]. However, high-fidelity approximate inference, through HMC, provides very poor generalization accuracy on out-of-distribution data, despite providing significantly better in-distribution accuracy on the analogous task, as shown in Figure 17.16 and Figure 17.17. These results stand in contrast to the good performance of many approximate inference procedures on these tasks [WI20; Dus+20; Ben+21], as well as work showing a strong correlation between in and out-of-distribution generalization accuracy on related tasks [Mil+21a], and the good performance of deep ensembles for both out-of-distribution detection and out-of-distribution generalization accuracy [Ova+19; Izm+21b].

Rather than an idiosyncracy of HMC, Izmailov et al. [Izm+21a] show this lack of robustness is a foundational issue of Bayesian model averaging under covariate shift, caused by degeneracies in the training data. As an illustrative special case, consider MNIST digits, which always have black corner pixels. The corresponding first layer weights are always multiplied by zero and have no effect on the likelihood. Consequently, these weights are simply sampled from the prior. If at test time the corner pixels are not black, e.g., due to corruption, these pixel values will be multiplied by random weights sampled from the prior, and propagated to the next layer, significantly degrading performance. On the other hand, classical MAP training or deep ensembles of MAP solutions drive the unrestricted parameters towards zero due to weight decay induced by any generic Gaussian prior, and will not be similarly affected by noise at test time. Here we indeed see a big difference between optimizing a posterior for MAP in comparison to a posterior weighted model average.

Figure 17.16(b, c) visualizes this example, where we see the weights in the first layer of a fully-connected network for a sample from the BNN posterior and the MAP solution on the MNIST dataset. The MAP solution weights are highly structured, while the BNN sample appears extremely noisy, similar to a draw from the Gaussian prior. In particular the weights corresponding to *dead pixels* (i.e. pixel positions that are black for all the MNIST images) near the boundary of the input image are set near zero (shown in white) by the MAP solution, but sampled randomly by the BNN. If at test time the data is corrupted, e.g. by Gaussian noise, and the pixels near the boundary of the image are activated, the MAP solution will ignore these pixels, while the predictions of the BNN will

47

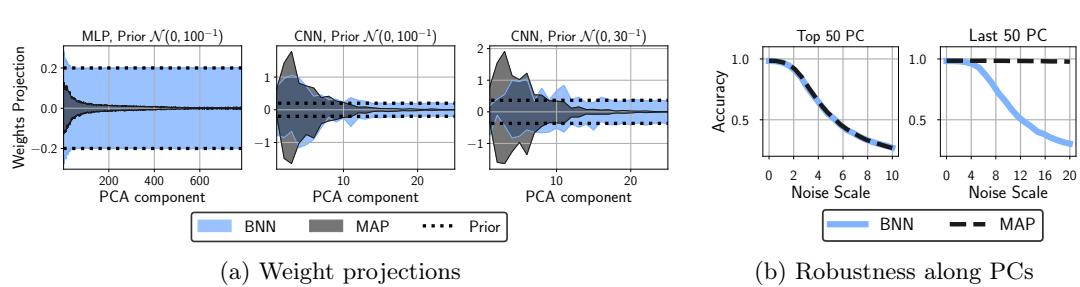


Figure 17.18: Bayesian inference samples weights along low-variance principal components from the prior, while MAP sets these weights to zero. (a): The distribution ($\text{mean} \pm 2 \text{ std}$) of projections of the weights of the first layer on the directions corresponding to the PCA components of the data for BNN samples and MAP solution using MLP and CNN architectures with different prior scales. In each case, MAP sets the weights along low-variance components to zero, while BNN samples them from the prior. (b): Accuracy of BNN and MAP solutions on the MNIST test set with Gaussian noise applied along the 50 highest and 50 lowest variance PCA components of the train data (left and right respectively). MAP is very robust to noise along low-variance PCA directions, while BMA is not; the two methods are similarly robust along the highest-variance PCA components. This figure is adapted from Figure 4 of Izmailov et al. [Izm+21a].

be significantly affected.

Izmailov et al. [Izm+21a] prove that this problem more generally occurs whenever there are linear dependencies in the input features of the data, for both fully-connected and convolutional networks. Intuitively, if there exists a direction in the input space such that all of the training data points have a constant projection on this direction (i.e. the data lies in a hyperplane), then posterior coincides with the prior in this direction. Hence, the BMA predictions are highly susceptible to perturbations that move the test inputs in a direction orthogonal to the hyperplane. The MAP solution on the other hand is completely robust to such perturbations.

We can gain some empirical intuitions into these theoretical results in Figure 17.18, by considering a fully-connected BNN on MNIST. The MNIST training dataset has linearly dependent features. In Figure 17.18(a), we see that the distribution of projections of the first layer weights onto the principal components of the data almost exactly coincides with the prior for PCA directions that are nearly constant in the data. The MAP solution, on the other hand, sets the weights along these PCA directions close to zero. In Figure 17.18(b), we see the MAP solution is very robust to noise along the low-variance directions, while BMA is not.

By introducing a prior over parameters which is aligned with the principal components of the training inputs, we can substantially improve the generalization accuracy of Bayesian neural networks in out-of-distribution settings. Izmailov et al. [Izm+21a] propose the following *EmpCov* prior: $p(w^1) = \mathcal{N}(0, \alpha\Sigma + \epsilon I)$, where w^1 are the first layer weights, $\Sigma = \frac{1}{n-1} \sum_{i=1}^n x_i x_i^T$ is the empirical covariance of the training input features x_i , $\alpha > 0$ determines the scale of the prior, and ϵ is a small positive constant to ensure the covariance matrix is positive definite.

1 **17.6 Online inference**

2 In Section 17.4, we have focused on batch or offline inference. However, an important application
3 of Bayesian inference is in sequential settings, where the data arrives in a continuous stream, and
4 the model has to “keep up”. This is called **sequential Bayesian inference**, and is one approach to
5 **online learning** (see Section 20.7.5). In this section, we discuss some algorithmic approaches to
6 this problem in the context of DNNs. These methods are widely used for continual learning, which
7 we discuss Section 20.7.

8 **17.6.1 Extended Kalman Filtering for DNNs**

9 In Section 8.4.2, we showed how Kalman filtering can be used to incrementally compute the
10 exact posterior for the weights of a linear regression model with known variance, i.e., we compute
11 $p(\boldsymbol{\theta}|\mathcal{D}_{1:t}, \sigma^2)$, where $\mathcal{D}_{1:t} = \{(\mathbf{u}_i, y_i) : i = 1 : t\}$ is the data seen so far, and

$$\underline{16} \quad p(y_t|\mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t|\boldsymbol{\theta}^\top \mathbf{u}_t, \sigma^2) \quad (17.29)$$

17 is the linear regression likelihood. The application of KF to this model is known as recursive least
18 squares.

19 Now consider the case of nonlinear regression:

$$\underline{22} \quad p(y_t|\mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t|f(\boldsymbol{\theta}, \mathbf{u}_t), \sigma^2) \quad (17.30)$$

23 where $f(\boldsymbol{\theta}, \mathbf{u}_t)$ is some nonlinear function, such as an MLP. We can use the extended Kalman filter
24 (Section 8.5.2) to approximately compute $p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t}, \sigma^2)$, where $\boldsymbol{\theta}_t$ is the hidden state (see e.g., [SW89;
25 PF03]). To see this, note that we can set the dynamics model to the identity function, $f(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t$, so
26 the parameters are propagated through unchanged, and the observation model to the input-dependent
27 function $f(\boldsymbol{\theta}_t) = f(\boldsymbol{\theta}_t, \mathbf{u}_t)$. We set the observation noise to $\mathbf{R}_t = \sigma^2$, and the dynamics noise to
28 $\mathbf{Q}_t = q\mathbf{I}$, where q is a small constant, to allow the parameters to slowly drift) according to artificial
29 **process noise**. (In practice it can be useful to anneal q from a large initial value to something near
30 0.)

31 In more detail, let \mathbf{H}_t be the Jacobian of the observation model at time t , which can be computed
32 using automatic differentiation. \mathbf{H}_t is a matrix of N_o column vectors, each of size N_w , where N_o is
33 the number of outputs of the MLP, and N_w is the number of weights (parameters). (Note that \mathbf{H}_t
34 plays the role of the observation matrix \mathbf{C}_t in the KF.) The dynamics matrix is $\mathbf{F}_t = \mathbf{I}$. The EKF
35 equations from Section 8.5.2 can then be written as follows:

$$\underline{38} \quad \mathbf{S}_t = (\mathbf{R}_t + \mathbf{H}_t^\top \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t) \quad (17.31)$$

$$\underline{39} \quad \mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t \mathbf{S}_t^{-1} \quad (17.32)$$

$$\underline{40} \quad \boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t(y_t - \hat{y}_t) \quad (17.33)$$

$$\underline{41} \quad \boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \quad (17.34)$$

$$\underline{43} \quad \boldsymbol{\mu}_{t+1|t} = \boldsymbol{\mu}_t \quad (17.35)$$

$$\underline{44} \quad \boldsymbol{\Sigma}_{t+1|t} = \boldsymbol{\Sigma}_t + \mathbf{Q}_t \quad (17.36)$$

45 where $\hat{y}_t = f(\mathbf{u}_t; \boldsymbol{\mu}_{t-1})$ is the predicted output.

46

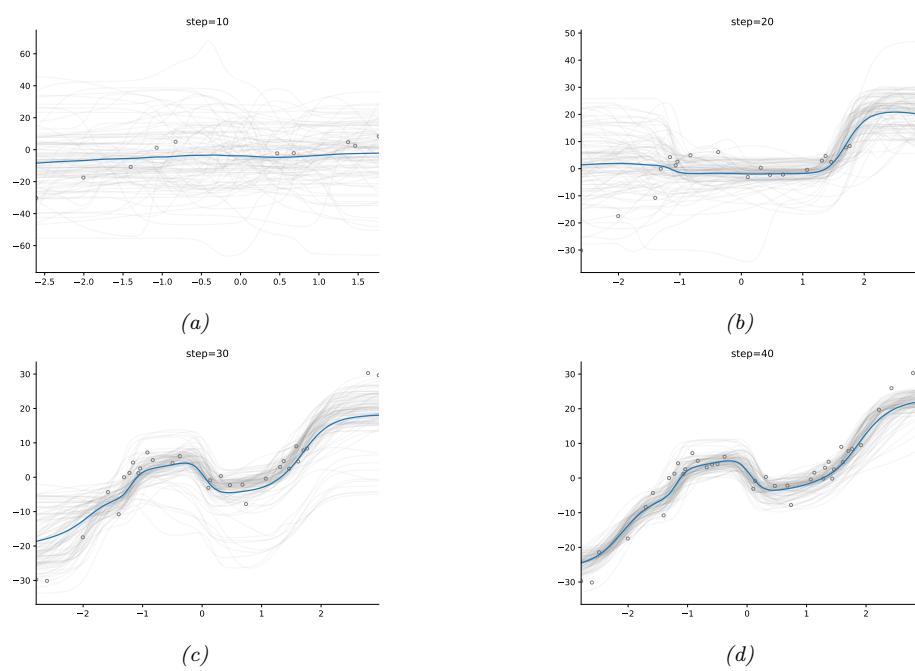


Figure 17.19: Sequential Bayesian inference for the parameters of an MLP applied to a nonlinear regression problem using the extended Kalman filter. We show results after seeing the first 10, 20, 30 and 40 observations. (For a video of this, see <https://bit.ly/3wXnWaM>.) Generated by `ekf_vs_ukf_mlp_demo.py`.

17.6.1.1 Example

We now give an example of this process in action. We sample a synthetic dataset from the true function

$$h^*(u) = x - 10 \cos(u) \sin(u) + u^3 \quad (17.37)$$

and add Gaussian noise with $\sigma = 3$. We then fit this with an MLP with one hidden layer with H hidden units, so the model has the form

$$f(\boldsymbol{\theta}, \mathbf{u}) = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{u} + \mathbf{b}_1) + \mathbf{b}_2 \quad (17.38)$$

where $\mathbf{W}_1 \in \mathbb{R}^{H \times 1}$, $\mathbf{b}_1 \in \mathbb{R}^H$, $\mathbf{W}_2 \in \mathbb{R}^{1 \times H}$, $\mathbf{b}_2 \in \mathbb{R}^1$. We set $H = 6$, so there are $D = 19$ parameters in total.

Given the data, we sequentially compute the posterior, starting from a vague Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta} | \mathbf{0}, \boldsymbol{\Sigma}_0)$, where $\boldsymbol{\Sigma}_0 = 100\mathbf{I}$. (In practice we cannot start from the prior mean, which is $\boldsymbol{\theta}_0 = \mathbf{0}$, since linearizing the model around this point results in a zero gradient, so we use an initial random sample for $\boldsymbol{\theta}_0$.) The results are shown in Figure 17.19. We can see that the model adapts to the data, without having to specify any learning rate. In addition, we see that the predictions become gradually more confident, as the posterior concentrates on the MLE.

1 **17.6.1.2 Setting the variance terms**

3 In the above example, we set the variance terms by hand. In general we need to estimate the noise
4 variance σ , which determines \mathbf{R}_t and hence the learning rate, as well as the strength of the prior Σ_0 ,
5 which controls the amount of regularization. Some methods for doing this are discussed in [FNG00].
6

7 **17.6.1.3 Reducing the computational complexity**

9 The naive EKF method described above takes $O(D^3)$ time, which is prohibitive for large neural
10 networks. A simple approximation, known as the **decoupled EKF**, was proposed in [PF91; SPD92]
11 (see [PF03] for a review). This partitions the weights into G groups or blocks, and estimates the
12 relevant matrices for each group g independently. More precisely, the update becomes

13

$$\mathbf{A}_t = \left(\mathbf{R}_t + \sum_{g=1}^G (\mathbf{H}_t^g)^\top \mathbf{P}_t^g \mathbf{H}_t^g \right)^{-1} \quad (17.39)$$

14

15 $\mathbf{K}_t^g = \mathbf{P}_t^g \mathbf{H}_t^g \mathbf{A}_t$ (17.40)

16 $\mathbf{w}_{t+1}^g = \mathbf{w}_t^g + \mathbf{K}_t^g (\mathbf{y}_t - \hat{\mathbf{y}}_t)$ (17.41)

17 $\mathbf{P}_{t+1}^g = \mathbf{P}_t^g - \mathbf{K}_t^g (\mathbf{H}_t^g)^\top \mathbf{P}_t^g + \mathbf{Q}_t^g$ (17.42)

21 If there are N_g weights in block g , $N_w = \sum_{g=1}^G N_g$ weights in total, and N_o outputs from the model,
22 the computation time is $O(N_o^2 N_w + N_o \sum_{g=1}^G N_g^2)$ and the space becomes $O(\sum_{g=1}^G N_g^2)$. If $G = 1$,
23 this reduces the standard global EKF. If we put each weight into its own group, we get a fully
24 diagonal approximation. In practice this does not work any better than SGD (possibly with diagonal
25 scaling), since it ignores correlations between the parameters. A useful compromise is to put all the
26 weights corresponding to each neuron into its own group; this is called “node decoupled EKF”.
27

28 **17.6.1.4 Beyond squared error**

30 The EKF assumes Gaussian observation noise, and thus minimizes the sum of squared errors at each
31 step between the prediction $\hat{\mathbf{y}}_t$ and the target vector \mathbf{y}_t . In Section 8.5.4, we discuss how to extend
32 the EKF so it can be applied to classification problems,
33

34 **17.6.2 Assumed Density Filtering for DNNs**

36 In Section 8.8.4, we discussed how to use assumed density filtering (ADF) to perform online (binary)
37 logistic regression. In this section, we generalize this to nonlinear predictive models, such as DNNs.
38 The key is to perform Gaussian moment matching of the hidden activations at each layer of the model.
39 This provides an alternative to the EKF approach in Section 17.6.1, which is based on linearization
40 of the network.

41 We will assume the following likelihood:

42 $p(\mathbf{y}_t | \mathbf{y}_t, \mathbf{w}_t) = \text{Expfam}(\mathbf{y}_t | \ell^{-1}(f(\mathbf{y}_t; \mathbf{w}_t)))$ (17.43)

43 where $f(\mathbf{x}; \mathbf{w})$ is the DNN, ℓ^{-1} is the inverse link function, and $\text{Expfam}()$ is some exponential family
44 distribution. For example, if f is linear and we are solving a binary classification problem, we can
45

1
2 write

3 $p(y_t | \mathbf{y}_t, \mathbf{w}_t) = \text{Ber}(y_t | \sigma(\mathbf{y}_t^\top \mathbf{w}_t))$ (17.44)

5 We discussed using ADF to fit this model in Section 8.8.4.

6 In [HLA15a], they propose **Probabilistic backpropagation (PBP)**, which is an instance of
7 ADF applied to MLPs. The basic idea is to approximate the posterior over the weights in each layer
8 using a fully factorized distribution

9

$$\underline{10} \quad p(\mathbf{w}_t | \mathcal{D}_{1:t}) \approx p_t(\mathbf{w}_t) = \prod_{l=1}^L \prod_{i=1}^{D_l} \prod_{j=1}^{D_{l-1}+1} \mathcal{N}(w_{ijl} | \mu_{ijl}^t, \tau_{ijl}^t) \quad (17.45)$$

11

13 where L is the number of layers, and D_l is the number of neurons in layer l . (The **expectation**
14 **backpropagation** algorithm of [SHM14] is a special case of this, where the variances are fixed to
15 $v = 1$.)

16 Suppose the parameters are static, so $\mathbf{w}_t = \mathbf{w}_{t-1}$. Then the new posterior, after conditioning on
17 the t 'th observation, is given by

18

$$\underline{19} \quad \hat{p}_t(\mathbf{w}) = \frac{1}{Z_t} p(\mathbf{y}_t | \mathbf{y}_t, \mathbf{w}) \mathcal{N}(\mathbf{w} | \boldsymbol{\mu}^{t-1}, \boldsymbol{\Sigma}^{t-1}) \quad (17.46)$$

20

21 where $\boldsymbol{\Sigma}^{t-1} = \text{diag}(\boldsymbol{\tau}^{t-1})$. We then project $\hat{p}_t(\mathbf{w})$ instead the space of factored Gaussians to compute
22 the new (approximate) posterior, $p_t(\mathbf{w})$. This can be done by computing the following means and
23 variances [Min01a]:

24

$$\underline{25} \quad \mu_{ijl}^t = \mu_{ijl}^{t-1} + \tau_{ijl}^{t-1} \frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \quad (17.47)$$

26

27

$$\underline{28} \quad \tau_{ijl}^t = \tau_{ijl}^{t-1} - (\tau_{ijl}^{t-1})^2 \left[\left(\frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \right)^2 - 2 \frac{\partial \ln Z_t}{\partial \tau_{ijl}^{t-1}} \right] \quad (17.48)$$

29

30 In the forwards pass, we compute Z_t by propagating the input \mathbf{y}_t through the model. Since we have
31 a Gaussian distribution over the weights, instead of a point estimate, this induces an (approximately)
32 Gaussian distribution over the values of the hidden units. For certain kinds of activation functions
33 (such as ReLU), the relevant integrals (to compute the means and variances) can be solved analytically,
34 as in GP-neural networks (Section 18.7). The result is that we get a Gaussian distribution over the
35 final layer of the form $\mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\eta}_t = f(\mathbf{y}_t; \mathbf{w}_t)$ is the output of the neural network before
36 the GLM link function induced by $p_t(\mathbf{w}_t)$. Hence we can approximate the partition function using
37

38

$$\underline{39} \quad Z_t \approx \int p(\mathbf{y}_t | \boldsymbol{\eta}_t) \mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma}) d\boldsymbol{\eta}_t \quad (17.49)$$

40

41 In the backwards pass, we take derivatives of this, and update the posteriors, \boldsymbol{m}^t and \boldsymbol{v}^t .

42 It remains to compute the marginal likelihood in Equation (17.49). In the case of probit classification,
43 with $y \in \{-1, +1\}$, we have $p(y | \mathbf{x}, \mathbf{w}) = \Phi(y\eta)$, where Φ is the cdf of the standard normal. We can
44 then use the following analytical result

45

$$\underline{46} \quad \int \Phi(y\eta) \mathcal{N}(h | \mu, \sigma) d\eta = \Phi \left(\frac{y\mu}{\sqrt{1+\sigma}} \right) \quad (17.50)$$

47

1 In the case of logistic classification, with $y \in \{0, 1\}$, we have $p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\sigma(\eta))$; in this case,
2 we can use the probit approximation from Section 15.3.5. For the multiclass case, where $\mathbf{y} \in \{0, 1\}^C$
3 (one-hot encoding), we have $p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \text{Cat}(\mathbf{y}|\sigma(\boldsymbol{\eta}))$. A variational lower bound to $\log Z_t$ for this
4 case is given in [GDFY16]. They also give an approximation to Z_t when using the Poisson likelihood,
5 where $y \in \{0, 1, 2, \dots\}$, and $p(y|\mathbf{x}, \mathbf{w}) = \text{Poi}(y|e^\eta)$.

7 17.6.3 Sequential Laplace for DNNs

8 In [RBB18b], they extended the Laplace method of Section 17.4.1 to the sequential setting. Specifically,
9 let $p(\boldsymbol{\theta}|\mathcal{D}_{1:t-1}) \approx \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}_{t-1}, \boldsymbol{\Lambda}_{t-1}^{-1})$ be the approximate posterior from the previous step; we assume
10 the precision matrix is Kronecker factored. We now compute the new mean by solving the MAP
11 problem

$$\boldsymbol{\mu}_t = \operatorname{argmax} \log p(\mathcal{D}_t|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}|\mathcal{D}_{t-1}) \quad (17.51)$$

$$= \operatorname{argmax} \log p(\mathcal{D}_t|\boldsymbol{\theta}) - \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1})\boldsymbol{\Lambda}_{t-1}^{-1}(\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \quad (17.52)$$

18 Once we have computed $\boldsymbol{\mu}_t$, we compute the approximate Hessian at this point, and get the new
19 posterior precision

$$\boldsymbol{\Lambda}_t = \lambda \mathbf{H}(\boldsymbol{\mu}_t) + \boldsymbol{\Lambda}_{t-1} \quad (17.53)$$

23 where $\lambda \geq 0$ is a weighting factor that trades off how much the model pays attention to the new data
24 vs old data.

25 Now suppose we use a diagonal approximation to the posterior prediction matrix. From Equa-
26 tion (17.52), we see that this amounts to adding a quadratic penalty to each new MAP estimate, to
27 encourage it to remain close to the parameters from previous tasks. This approach is called **elastic**
28 **weight consolidation (EWC)** [Kir+17].

30 17.6.4 Variational methods

31 A natural approach to online Bayesian inference is to use variational inference, where the prior is the
32 posterior from the previous time step. That is, we optimize

$$\mathcal{L}(\boldsymbol{\psi}_t) = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi}_t)} [\log p(\mathcal{D}_t|\boldsymbol{\theta})] - D_{\text{KL}}(q(\boldsymbol{\theta}|\boldsymbol{\psi}_t) \| q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})) \quad (17.54)$$

36 This is called **variational continual learning** or **VCL** [Ngu+18b], as we discussed in Section 10.3.9.

37 Unfortunately this method often does not work in practice, in part because of the variational
38 overpruning effect discussed in Section 17.4.2, in which hidden units from earlier tasks are “turned
39 off” (have their weights set to 0), which is an effect that cannot be easily undone using gradient
40 based learning. Some generalizations of VCL that ameliorate this problem are discussed in [LST21].

42 17.7 Hierarchical Bayesian neural networks

43 In some problems, we have multiple related datasets, such as a set of medical images from different
44 hospitals. Some aspects of the data (e.g., the shape of cells as a function of their state) is generally
45

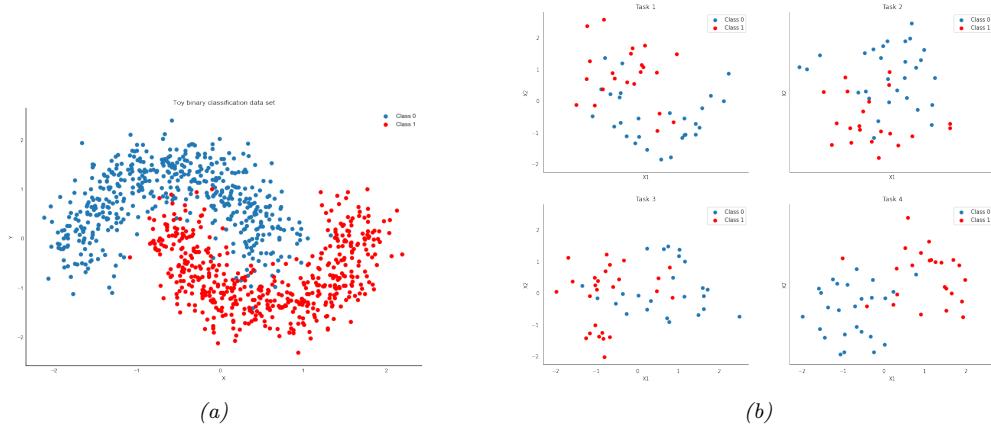


Figure 17.20: (a) Two moons synthetic dataset. (b) Multi-task version, where we rotate the data to create 18 related tasks (groups). Each dataset has 50 training and 50 test points. Here we show the first 4 tasks. Generated by [bnn_hierarchical_blackjax.ipynb](#).

the same across datasets, but other aspects may be unique or idiosyncratic (e.g., each hospital may use a different colored die for staining). To model this, we can use a hierarchical Bayesian model, in which we allow the parameters for each dataset to be different (to capture random effects), while coming from a common prior (to capture shared effects). This is a neural net version of the mixed effects models we studied in Section 15.5.1. We give an example below.

17.7.1 Solving multiple related classification problems

In this section, we consider an example⁴ where we want to solve multiple related nonlinear binary classification problems. We assume that each subproblem or “task” t only has a small number N_t of examples, which share the same shaped decision boundary, but modified in a way that is unique to each task. (This is an example of multi-task learning, which we discuss in more detail in Section 20.5.5, and is related to domain generalization, which we discuss in more detail in Section 20.5.6.)

We first create some synthetic 2d data for the $T = 18$ tasks. We start with the “two-moons” dataset, illustrated in Figure 17.20a. Each task is obtained by rotating the 2d inputs by a different amount, to create 18 related classification problems (see Figure 17.20b). Since we have multiple versions of the two-moons dataset, we call it the “multi-moons” dataset. See Figure 17.20b for the training data for 4 tasks.

To handle the nonlinear decision boundary, we use a multilayer perceptron. Since the dataset is low-dimensional (2d input), we use a shallow model with just 2 hidden layers, each with 5 neurons. We could fit a separate MLP to each task, but since we have limited data per task ($N_t = 50$ examples for training), this works poorly, as we show below. We could also pool all the data and fit a single model, but this does even worse, since the datasets come from different underlying distributions. Instead we adopt a hierarchical Bayesian approach, as illustrated in Figure 17.21. In particular, we

4. This example is from https://twiecki.io/blog/2018/08/13/hierarchical_bayesian_neural_network/.

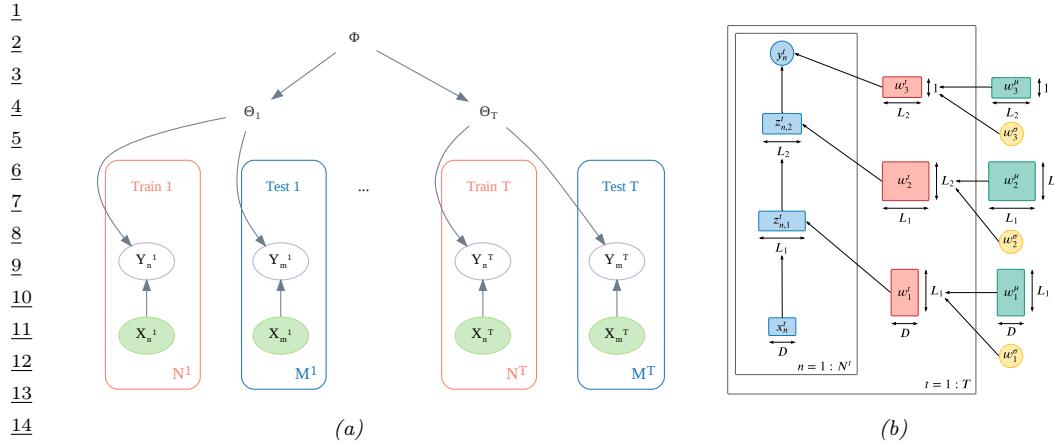


Figure 17.21: (a) Illustration of a hierarchical Bayesian discriminative model with T groups or tasks. Generated by [hbayes_figures2.ipynb](#). (b) Zoom-in on the dependency structure, for the case where $p(y|\mathbf{x}, \theta_t)$ is an MLP with 2 hidden layers, with sizes L_1 and L_2 . The input is D -dimensional, and output is the scalar logit for the binary output. Used with kind permission of Aleyna Kara.

assume the weight from unit i to unit j in layer l for task t , denoted $w_{i,j,l}^t$, come from a common (task independent) prior $\mathcal{N}(\mu_{i,j,l}, \sigma_l^2)$. We use the non-centered parameterization from Section 12.6.4 to write

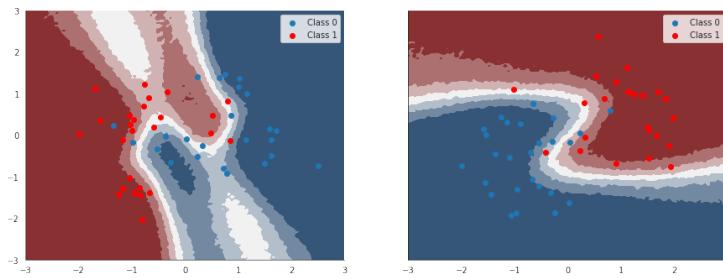
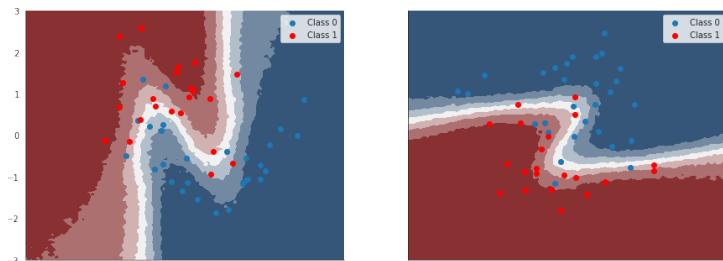
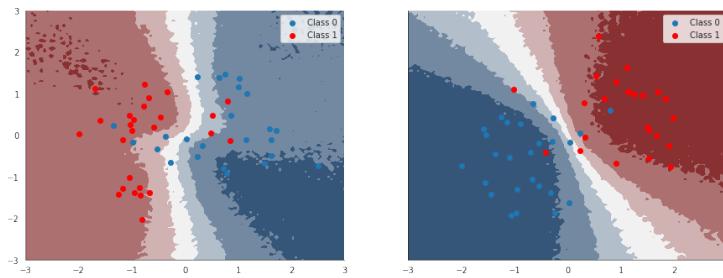
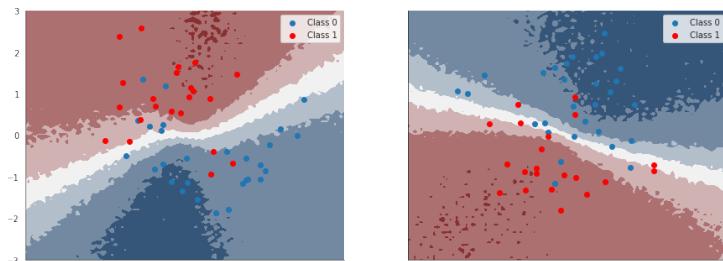
$$w_{i,j,l}^t = \mu_{i,j,l} + \tilde{w}_{i,j,l}^t \times \sigma_l \quad (17.55)$$

where $\tilde{w}_{i,j,l}^t \sim \mathcal{N}(0, 1)$. For the hyper-parameters, we put $\mathcal{N}(0, 1)$ priors on $\mu_{i,j,l}$, and $\mathcal{N}_+(1)$ priors on σ_l . By allowing a different σ_l per layer, we let the model control the degree of shrinkage to the prior for each layer separately.

We compute the posterior $p(\tilde{\mathbf{w}}^{1:T}, \boldsymbol{\mu}, \boldsymbol{\sigma} | \mathcal{D})$ using HMC (Section 12.5). The average classification accuracy on the train and test sets for the non-hierarchical model (one MLP per group, fit separately) is 86% and 83%. For the hierarchical model, this improves to 91% and 89% respectively.

To see why the hierarchical model works better, we will plot the posterior predictive distribution in 2d. Figure 17.22 shows the results for the non-hierarchical models; we see that the method fails to learn the common underlying Z-shaped decision boundary. By contrast, Figure 17.23 shows that the method has correctly recovered the common pattern, while still allowing group variation.

Of course, multi-task learning does not always help (see e.g., [WZR20]), because sometimes there can be “**task interference**” or “**negative transfer**”. In Bayesian terms, this just means the modeling assumptions about a shared unimodal prior being useful for all tasks is inappropriate. This can be solved by using richer priors, such as mixture distributions, or sparse priors.



18 Gaussian processes

This chapter is co-authored with Andrew Wilson.

18.1 Introduction

Deep neural networks are a family of flexible function approximators of the form $f(\mathbf{x}; \boldsymbol{\theta})$, where the dimensionality of $\boldsymbol{\theta}$ (i.e., the number of parameters) is fixed, and independent of the size N of the training set. However, such parametric models can overfit when N is small, and can underfit when N is large, due to their fixed capacity. In order to create models whose capacity automatically adapts to the amount of data, we turn to **nonparametric models**.

There are many approaches to building nonparametric models for classification and regression (see e.g., [Was06]). In this chapter, we consider a Bayesian approach in which we represent uncertainty about the input-output mapping f by defining a prior distribution over functions, and then updating it given data. This is an example of a Bayesian nonparametric model — see Chapter 33 for other examples.

To explain this procedure in more detail, recall that a Gaussian random vector of length N , $\mathbf{f} = [f_1, \dots, f_N]$, is defined by its mean $\boldsymbol{\mu} = \mathbb{E}[\mathbf{f}]$ and its covariance $\boldsymbol{\Sigma} = \text{Cov}[\mathbf{f}]$. Now consider a function $f : \mathcal{X} \rightarrow \mathbb{R}$ evaluated at a set of inputs, $\mathbf{X} = \{\mathbf{x}_n \in \mathcal{X}\}_{n=1}^N$. Let $\mathbf{f}_X = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_N)]$ be the set of unknown function values at these points. If \mathbf{f}_X is jointly Gaussian for any set of $N \geq 1$ points, then we say that $f : \mathcal{X} \rightarrow \mathbb{R}$ is a **Gaussian process**. Such a process is defined by its **mean function** $m(\mathbf{x}) \in \mathbb{R}$ and a **covariance function**, $\mathcal{K}(\mathbf{x}, \mathbf{x}') \geq 0$, which is any positive definite **Mercer kernel** (see Section 18.2). For example, we might use an RBF kernel of the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') \propto \exp(-\|\mathbf{x} - \mathbf{x}'\|^2)$ (see Section 18.2 for details).

We denote the corresponding GP by

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), \mathcal{K}(\mathbf{x}, \mathbf{x}')) \tag{18.1}$$

where

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \tag{18.2}$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))^\top] \tag{18.3}$$

This means that, for any finite set of points $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, we have

$$p(\mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | \boldsymbol{\mu}_X, \mathbf{K}_{X,X}) \tag{18.4}$$

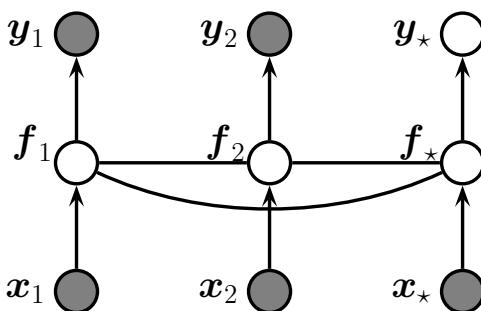


Figure 18.1: A Gaussian process for 2 training points, \mathbf{x}_1 and \mathbf{x}_2 , and 1 testing point, \mathbf{x}_* , represented as a graphical model representing $p(\mathbf{y}, \mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | m(\mathbf{X}), \mathcal{K}(\mathbf{X})) \prod_i p(y_i | f_i)$. The hidden nodes $f_i = f(\mathbf{x}_i)$ represent the value of the function at each of the data points. These hidden nodes are fully interconnected by undirected edges, forming a Gaussian graphical model; the edge strengths represent the covariance terms $\Sigma_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$. If the test point \mathbf{x}_* is similar to the training points \mathbf{x}_1 and \mathbf{x}_2 , then the value of the hidden function f_* will be similar to f_1 and f_2 , and hence the predicted output y_* will be similar to the training values y_1 and y_2 .

where $\mu_X = (m(\mathbf{x}_1), \dots, m(\mathbf{x}_N))$ and $\mathbf{K}_{X,X}(i,j) \triangleq \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$.

A GP can be used to define a prior over functions. We can evaluate this prior at any set of points we choose. However, to learn about the function from data, we have to update this prior with a likelihood function. We typically assume we have a set of N iid observations $\mathcal{D} = \{(\mathbf{x}_i, y_i) : i = 1 : N\}$, where $y_i \sim p(y|f(\mathbf{x}_i))$, as shown in Figure 18.1. If we use a Gaussian likelihood, we can compute the posterior $p(f|\mathcal{D})$ in closed form, as we discuss in Section 18.3. For other kinds of likelihoods, we will need to use approximate inference, as we discuss in Section 18.4. In many cases f is not directly observed, and instead forms part of a latent variable model, both in supervised and unsupervised settings such as in Section 29.3.6.

The generalization properties of a Gaussian process are controlled by its covariance function (kernel), which we describe in Section 18.2. These kernels live in a reproducing kernel Hilbert space (RKHS), described in Section 18.3.7.1.

GPs were originally designed for spatial data analysis, where the input is 2d. This special case is called **kriging**. However, they can be applied to higher dimensional inputs. In addition, while they have been traditionally limited to small datasets, it is now possible to apply GPs to problems with millions of points, with essentially exact inference. We discuss these scalability advances in Section 18.5.

Moreover, while Gaussian processes have historically been considered smoothing interpolators, GPs now routinely perform representation learning, through covariance function learning, and multilayer models. These advances have clearly illustrated that GPs are neural networks are not competing, but complementary, and can be combined for better performance than would be achieved by deep learning alone. We describe GPs for representation learning in Section 18.6.

The connections between Gaussian processes and neural networks can also be further understood

1 by considering infinite limits of neural networks that converge to Gaussian processes with particular
 2 covariance functions, which we describe in Section 18.7.

3 So Gaussian processes are non-parametric models which can scale and do representation learning.
 4 But why, in the age of deep learning, should we want to use a Gaussian process? There are several
 5 compelling reasons to prefer a GP, including:

- 6 • Gaussian processes typically provide well-calibrated predictive distributions, with a good char-
 7 acterization of epistemic (model) uncertainty — uncertainty arising from not knowing which of
 8 many solutions is correct. For example, as we move away from the data, there are a greater variety
 9 of consistent solutions, and so we expect greater uncertainty.
- 10 • Gaussian processes are often state-of-the-art for continuous regression problems, especially spa-
 11 tiotemporal problems, such as weather interpolation and forecasting. In regression, Gaussian
 12 process inference can also typically be performed in closed form.
- 13 • The marginal likelihood of a Gaussian process provides a powerful mechanism for flexible kernel
 14 learning. Kernel learning enables us to provide long-range extrapolations, but also tells us
 15 interpretable properties of the data that we didn't know before, towards scientific discovery.
- 16 • Gaussian processes are often used as a probabilistic surrogate for objectives in optimization, in a
 17 procedure known as **Bayesian optimization** (Section 6.9). To maximize an objective, we wish
 18 to move where there is a high expected value, but also to explore where we have large uncertainty.
 19 The ability for a Gaussian process to provide closed form inference in regression, in conjunction
 20 with high quality uncertainty representations, make them particularly impactful in this setting.
 21 Bayesian optimization has a wide range of applications, including A/B testing, experimental
 22 design, protein engineering, hyperparameter tuning, and AutoML. See Section 6.9 for details.

27 18.2 Mercer kernels

28 The generalization properties of Gaussian processes boil down to how we encode prior knowledge
 29 about the similarity of two input vectors. If we know that \mathbf{x}_i is similar to \mathbf{x}_j , then we can encourage
 30 the model to make the predicted output at both locations (i.e., $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$) to be similar.

31 To define similarity, we introduce the notion of a **kernel function**. The word “kernel” has many
 32 different meanings in mathematics; here we consider a **Mercer kernel**, also called a **positive**
 33 **definite kernel**. This is any symmetric function $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$ such that

$$34 \quad \sum_{i=1}^N \sum_{j=1}^N \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) c_i c_j \geq 0 \quad (18.5)$$

35 for any set of N (unique) points $\mathbf{x}_i \in \mathcal{X}$, and any choice of numbers $c_i \in \mathbb{R}$. We assume $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) > 0$,
 36 so that we can only achieve equality in the above equation if $c_i = 0$ for all i .

37 Another way to understand this condition is the following. Given a set of N datapoints, let us
 38 define the **Gram matrix** as the following $N \times N$ similarity matrix:

$$39 \quad \mathbf{K} = \begin{pmatrix} \mathcal{K}(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ \mathcal{K}(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} \quad (18.6)$$

1 We say that \mathcal{K} is a Mercer kernel iff the Gram matrix is positive definite for any set of (distinct)
2 inputs $\{\mathbf{x}_i\}_{i=1}^N$.

3 The most widely used kernel for real-valued inputs is the **radial basis function kernel** (RBF)
4 kernel), also called the **exponentiated quadratic kernel**, **Gaussian kernel**, **squared exponential**
5 **(SE) kernel**. This kernel is defined as

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \quad (18.7)$$

6 Here ℓ corresponds to the length-scale of the kernel, i.e., the distance over which we expect differences
7 to matter. This is also sometimes known as the **bandwidth** parameter. The RBF kernel measures
8 similarity between two vectors in \mathbb{R}^D using (scaled) Euclidean distance. In Section 18.2.1, we will
9 discuss several other kinds of kernel.

10

11 18.2.1 Some popular Mercer kernels

12 In the sections below, we describe some popular Mercer kernels. More details can be found at [Wil14]
13 and <https://www.cs.toronto.edu/~duvenaud/cookbook/>. See also Section 18.6 where we discuss
14 how to learn kernels from data.

15

16 18.2.1.1 Stationary kernels for real-valued vectors

17 For real-valued inputs, $\mathcal{X} = \mathbb{R}^D$, it is common to use **stationary kernels**, which are functions of
18 the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}(\mathbf{r})$, where $\mathbf{r} = \mathbf{x} - \mathbf{x}'$; thus the output only depends on the relative difference
19 between the inputs. Furthermore, in many cases, it is only the magnitude of the difference, $r = \|\mathbf{r}\|_2$,
20 that matters. We give some examples below.

21

22 Squared exponential kernel

23 The **squared exponential** (SE) kernel is defined as

$$\mathcal{K}(r; \ell) = \exp\left(-\frac{r^2}{2\ell^2}\right) \quad (18.8)$$

24 Here ℓ corresponds to the length-scale of the kernel, i.e., the distance over which we expect differences
25 to matter.

26

27 ARD kernel

28 We can generalize the RBF kernel by replacing Euclidean distance with Mahalanobis distance, as
29 follows:

$$\mathcal{K}(\mathbf{r}; \Sigma, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \mathbf{r}^\top \Sigma^{-1} \mathbf{r}\right) \quad (18.9)$$

30 where $\mathbf{r} = \mathbf{x} - \mathbf{x}'$. If Σ is diagonal, this can be written as

$$\mathcal{K}(\mathbf{r}; \ell, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} r_d^2\right) = \prod_{d=1}^D \mathcal{K}(r_d; \ell_d, \sigma^{2/d}) \quad (18.10)$$

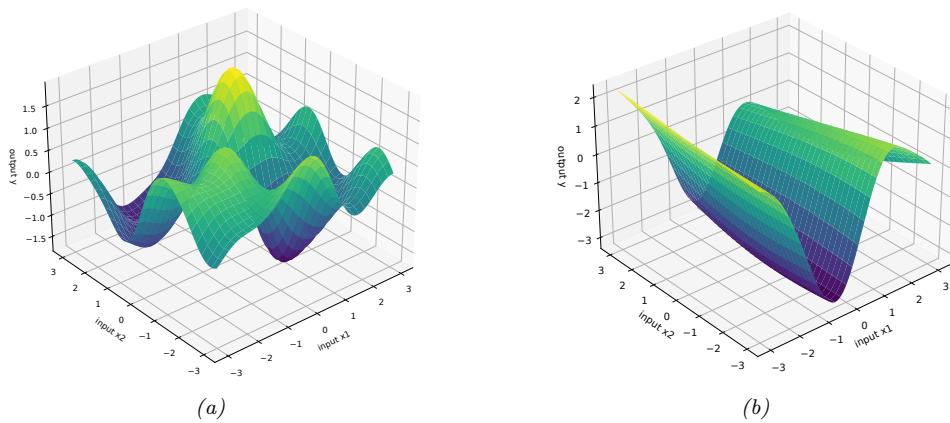


Figure 18.2: Function samples from a GP with an ARD kernel. (a) $\ell_1 = \ell_2 = 1$. Both dimensions contribute to the response. (b) $\ell_1 = 1, \ell_2 = 5$. The second dimension is essentially ignored. Adapted from Figure 5.1 of [RW06]. Generated by [gprDemoArd.py](#).

where

$$\mathcal{K}(r; \ell, \tau^2) = \tau^2 \exp\left(-\frac{1}{2} \frac{1}{\ell^2} r^2\right) \quad (18.11)$$

We can interpret σ^2 as the overall variance, and ℓ_d as defining the **characteristic length scale** of dimension d . If d is an irrelevant input dimension, we can set $\ell_d = \infty$, so the corresponding dimension will be ignored. This is known as **Automatic Relevance Determination** or **ARD** (Section 15.2.7). Hence the corresponding kernel is called the **ARD kernel**. See Figure 18.2 for an illustration of some 2d functions sampled from a GP using this prior.

Matern kernels

The SE kernel gives rise to functions that are infinitely differentiable, and therefore are very smooth. For many applications, it is better to use the **Matern kernel**, which gives rise to “rougher” functions, which can better model local “wiggles” without having to make the overall length scale very small.

The Matern kernel has the following form:

$$\mathcal{K}(r; \nu, \ell) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu}r}{\ell} \right) \quad (18.12)$$

where K_ν is a modified Bessel function and ℓ is the length scale. Functions sampled from this GP are k -times differentiable iff $\nu > k$. As $\nu \rightarrow \infty$, this approaches the SE kernel.

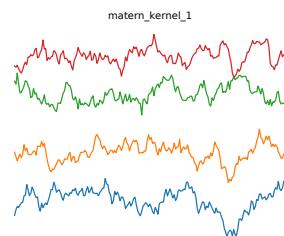
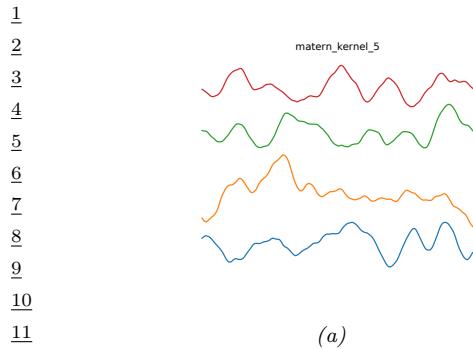


Figure 18.3: Functions sampled from a GP with a Matern kernel. (a) $\nu = 5/2$. (b) $\nu = 1/2$. Generated by [gpKernelPlot.py](#).

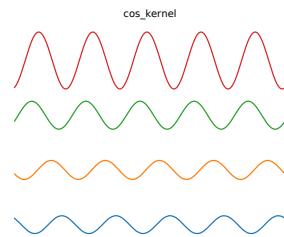
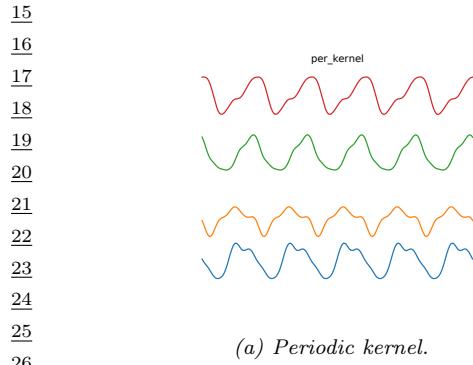


Figure 18.4: Functions sampled from a GP using various stationary periodic kernels. Generated by [gpKernelPlot.py](#).

For values $\nu \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}\}$, the function simplifies as follows:

$$\mathcal{K}(r; \frac{1}{2}, \ell) = \exp\left(-\frac{r}{\ell}\right) \quad (18.13)$$

$$\mathcal{K}(r; \frac{3}{2}, \ell) = \left(1 + \frac{\sqrt{3}r}{\ell}\right) \exp\left(-\frac{\sqrt{3}r}{\ell}\right) \quad (18.14)$$

$$\mathcal{K}(r; \frac{5}{2}, \ell) = \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}r}{\ell}\right) \quad (18.15)$$

The value $\nu = \frac{1}{2}$ corresponds to the **Ornstein-Uhlenbeck process**, which describes the velocity of a particle undergoing Brownian motion. The corresponding function is continuous but not differentiable, and hence is very “jagged”. See Figure 18.3b for an illustration.

1
2 **Periodic kernels**

3 One way to create a periodic 1d random function is to map x to the 2d space $\mathbf{u}(x) = (\cos(x), \sin(x))$,
4 and then use an SE kernel in \mathbf{u} -space:

5

$$\mathcal{K}(x, x') = \exp\left(-\frac{2\sin^2((x - x')/2)}{\ell^2}\right) \quad (18.16)$$

6 which follows since $(\cos(x) - \cos(x'))^2 + (\sin(x) - \sin(x'))^2 = 4\sin^2((x - x')/2)$. We can generalize this
7 by specifying the period p to get the **periodic kernel**, also called the **exp-sine-squared kernel**:

8

$$\mathcal{K}_{\text{per}}(r; \ell, p) = \exp\left(-\frac{2}{\ell^2} \sin^2(\pi \frac{r}{p})\right) \quad (18.17)$$

9 where p is the period and ℓ is the length scale. See Figure 18.4a for an illustration.

10 A related kernel is the **cosine kernel**:

11

$$\mathcal{K}(r; p) = \cos\left(2\pi \frac{r}{p}\right) \quad (18.18)$$

12 See Figure 18.4b for an illustration.

13 **Rational quadratic kernel**

14 We define the **rational quadratic** kernel to be

15

$$\mathcal{K}_{RQ}(r; \ell, \alpha) = \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha} \quad (18.19)$$

16 We recognize this is proportional to a Student T density. Hence it can be interpreted as a scale
17 mixture of SE kernels of different characteristic lengths.

18 **18.2.1.2 Making new kernels from old**

19 Given two valid kernels $\mathcal{K}_1(\mathbf{x}, \mathbf{x}')$ and $\mathcal{K}_2(\mathbf{x}, \mathbf{x}')$, we can create a new kernel using any of the following
20 methods:

21

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = c\mathcal{K}_1(\mathbf{x}, \mathbf{x}'), \text{ for any constant } c > 0 \quad (18.20)$$

22

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})\mathcal{K}_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}'), \text{ for any function } f \quad (18.21)$$

23

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = q(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \text{ for any function polynomial } q \text{ with nonneg. coef.} \quad (18.22)$$

24

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \quad (18.23)$$

25

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{A} \mathbf{x}', \text{ for any psd matrix } \mathbf{A} \quad (18.24)$$

26 For example, suppose we start with the linear kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'$. We know this is a valid
27 Mercer kernel, since the corresponding Gram matrix is just the (scaled) covariance matrix of the
28 data. From the above rules, we can see that the **polynomial kernel**

29

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}')^M \quad (18.25)$$

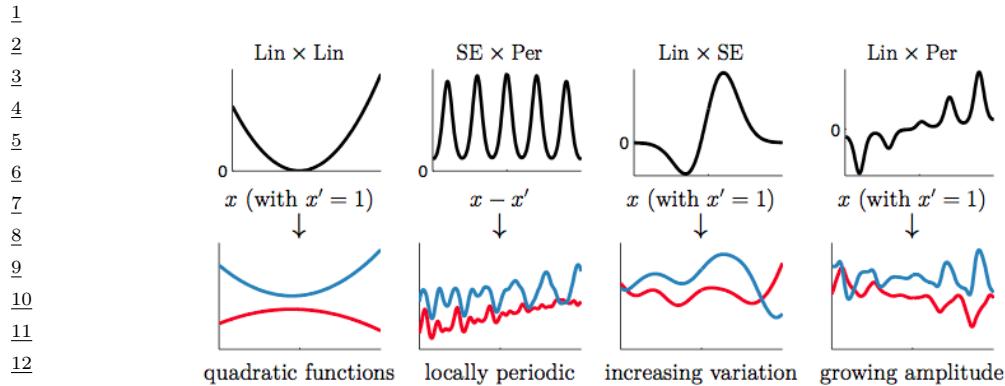


Figure 18.5: Examples of 1d structures obtained by multiplying elementary kernels. Top row shows $\mathcal{K}(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, \mathcal{K})$. From Figure 2.2 of [Duv14]. Used with kind permission of David Duvenaud.

is a valid Mercer kernel. This contains all monomials of order M . For example, if $M = 2$ and the inputs are 2d, we have

$$(\mathbf{x}^\top \mathbf{x}')^2 = (x_1 x'_1 + x_2 x'_2)^2 = (x_1 x'_1)^2 + (x_2 x'_2)^2 + (x_1 x'_1)(x_2 x'_2) \quad (18.26)$$

We can generalize this to contain all terms up to degree M by using the kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + c)^M$. For example, if $M = 2$ and the inputs are 2d, we have

$$\begin{aligned} (\mathbf{x}^\top \mathbf{x}' + 1)^2 &= (x_1 x'_1)^2 + (x_1 x'_1)(x_2 x'_2) + (x_1 x'_1) \\ &\quad + (x_2 x'_2)(x_1 x'_1) + (x_2 x'_2)^2 + (x_2 x'_2) \\ &\quad + (x_1 x'_1) + (x_2 x'_2) + 1 \end{aligned} \quad (18.27)$$

We can also use the above rules to establish that the Gaussian kernel is a valid kernel. To see this, note that

$$\|\mathbf{x} - \mathbf{x}'\|^2 = \mathbf{x}^\top \mathbf{x} + (\mathbf{x}')^\top \mathbf{x}' - 2\mathbf{x}^\top \mathbf{x}' \quad (18.28)$$

and hence

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2) = \exp(-\mathbf{x}^\top \mathbf{x} / 2\sigma^2) \exp(\mathbf{x}^\top \mathbf{x}' / \sigma^2) \exp(-(\mathbf{x}')^\top \mathbf{x}' / 2\sigma^2) \quad (18.29)$$

is a valid kernel.

18.2.1.3 Combining kernels by addition and multiplication

We can also combine kernels using addition or multiplication:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') + \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.30)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') \times \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.31)$$

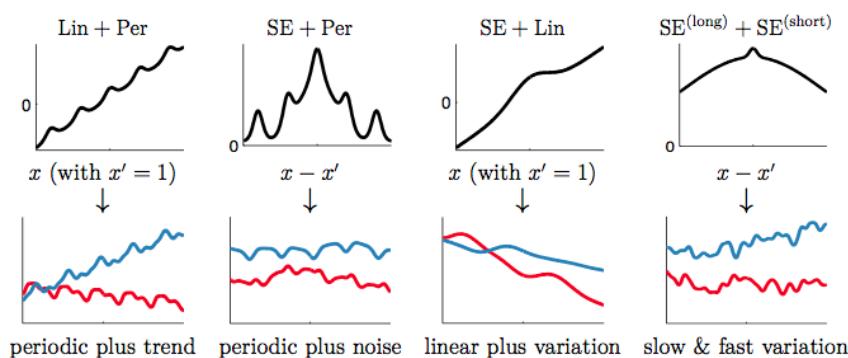


Figure 18.6: Examples of 1d structures obtained by adding elementary kernels. Here $SE^{(\text{short})}$ and $SE^{(\text{long})}$ are two SE kernels with different length scales. From Figure 2.4 of [Duv14]. Used with kind permission of David Duvenaud.

Multiplying two positive-definite kernels together always results in another positive definite kernel. This is a way to get a conjunction of the individual properties of each kernel, as illustrated in Figure 18.5.

In addition, adding two positive-definite kernels together always results in another positive definite kernel. This is a way to get a disjunction of the individual properties of each kernel, as illustrated in Figure 18.6.

For an example of combining kernels to forecast some timeseries data, see Section 19.3.3.1.

18.2.1.4 Kernels for structured inputs

Kernels are particularly useful when the inputs are structured objects, such as strings and graphs, since it is often hard to “featurize” variable-sized inputs. For example, we can define a **string kernel** which compares strings in terms of the number of n-grams they have in common [Lod+02; BC17].

We can also define kernels on graphs [KJM19]. For example, the **random walk kernel** conceptually performs random walks on two graphs simultaneously, and then counts the number of paths that were produced by both walks. This can be computed efficiently as discussed in [Vis+10]. For more details on graph kernels, see [KJM19].

For a review of kernels on structured objects, see e.g., [Gär03].

18.2.2 Mercer’s theorem

Recall that any positive definite matrix \mathbf{K} can be represented using an eigendecomposition of the form $\mathbf{K} = \mathbf{U}^T \boldsymbol{\Lambda} \mathbf{U}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$, and \mathbf{U} is a matrix containing the eigenvectors. Now consider element (i, j) of \mathbf{K} :

$$k_{ij} = (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,i})^T (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,j}) \quad (18.32)$$

1 where $\mathbf{U}_{:i}$ is the i 'th column of \mathbf{U} . If we define $\phi(\mathbf{x}_i) = \Lambda^{\frac{1}{2}} \mathbf{U}_{:i}$, then we can write
2

3

$$\underline{4} \quad k_{ij} = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = \sum_{m=1}^M \phi_m(\mathbf{x}_i) \phi_m(\mathbf{x}_j) \quad (18.33)$$

5

6 where M is the rank of the kernel matrix. Thus we see that the entries in the kernel matrix can be
7 computed by performing an inner product of some feature vectors that are implicitly defined by the
8 eigenvectors of the kernel matrix.
9

10 This idea can be generalized to apply to kernel functions, not just kernel matrices, as we now show.
11 First, we define an **eigenfunction** $\phi()$ of a kernel \mathcal{K} with eigenvalue λ wrt measure μ as a function
12 that satisfies

13

$$\underline{14} \quad \int \mathcal{K}(\mathbf{x}, \mathbf{x}') \phi(\mathbf{x}) d\mu(\mathbf{x}) = \lambda \phi(\mathbf{x}') \quad (18.34)$$

15 We usually sort the eigenfunctions in order of decreasing eigenvalue, $\lambda_1 \geq \lambda_2 \geq \dots$. The eigenfunctions
16 are orthogonal wrt μ :
17

18

$$\underline{19} \quad \int \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mu(\mathbf{x}) = \delta_{ij} \quad (18.35)$$

20 where δ_{ij} is the Kronecker delta. With this definition in hand, we can state **Mercer's theorem**.
21 Informally, it says that any positive definite kernel function can be represented as the following
22 infinite sum:
23

24

$$\underline{25} \quad \mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{m=1}^{\infty} \lambda_m \phi_m(\mathbf{x}) \phi_m(\mathbf{x}') \quad (18.36)$$

26

27 where ϕ_m are eigenfunctions of the kernel, and λ_m are the corresponding eigenvalues. This is the
28 functional analog of Equation (18.33).

29 A **degenerate kernel** has only a finite number of non-zero eigenvalues. In this case, we can
30 rewrite the kernel function as an inner product between two finite-length vectors. For example,
31 consider the **quadratic kernel** $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle^2$. In 2d, we have

32

$$\underline{33} \quad \mathcal{K}(\mathbf{x}, \mathbf{x}') = (x_1 x'_1 + x_2 x'_2)^2 = x_1^2(x'_1)^2 + 2x_1 x_2 x'_1 x'_2 + x_2^2(x'_2)^2 \quad (18.37)$$

34 If we define $\phi(x_1, x_2) = [x_1^2, \sqrt{2}x_1 x_2, x_2^2] \in \mathbb{R}^3$, then we can write this as $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$.
35 Thus we see that this kernel is degenerate.

36 Now consider the RBF kernel. In this case, the corresponding feature representation is infinite
37 dimensional (see Section 18.2.3.1 for details). However, by working with kernel functions, we can
38 avoid having to deal with infinite dimensional vectors.

39 From the above, we see that we can replace inner product operations in an explicit (possibly infinite
40 dimensional) feature space with a call to a kernel function, i.e., we replace $\phi(\mathbf{x})^T \phi(\mathbf{x})$ with $\mathcal{K}(\mathbf{x}, \mathbf{x}')$.
41 This is called the **kernel trick**.

42

43 18.2.3 Kernels from Spectral Densities

44

45 Consider the case of a **shift-invariant kernel**, also known as a *stationary kernel*, which satisfies
46 $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}(\boldsymbol{\delta})$, where $\boldsymbol{\delta} = \mathbf{x} - \mathbf{x}'$, for $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$. Let us further assume that $\mathcal{K}(\boldsymbol{\delta})$ is positive definite.
47

In this case, **Bochner's theorem** tells us that we can represent $\mathcal{K}(\delta)$ by its Fourier transform:

$$\mathcal{K}(\delta) = \int_{\mathbb{R}^d} p(\omega) e^{j\omega^\top \delta} d\omega \quad (18.38)$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j \sin(\theta)$, and $p(\omega)$, the **spectral density**, is a distribution over frequencies.

We can easily derive and gain intuitions into several kernels from spectral densities. If we take the Fourier transform of an RBF kernel we find the spectral density $p(\omega) = \sqrt{2\pi\ell^2} \exp(-2\pi^2\omega^2\ell^2)$. Thus the spectral density is also Gaussian, but with a bandwidth *inversely* proportional to the length-scale hyperparameter ℓ . That is, as ℓ becomes large, the spectral density collapses onto a point mass. This result is intuitive: as we increase the length-scale our model treats points as correlated over large distances, and becomes very smooth and slowly varying, and thus low-frequency. In general, since the Gaussian distribution has relatively light tails, we can see that RBF kernels won't generally support high frequency solutions.

We can instead use a Student- t spectral density, which has heavy tails that will provide greater support for higher frequencies. Taking the inverse Fourier transform of this spectral density, we recover the Matern kernel, with degrees of freedom ν corresponding to the degrees of freedom in the spectral density. Indeed, the smaller we make ν the less smooth and higher frequency are the associated fits to data using a Matern kernel.

We can also derive **spectral mixture kernels** by modelling the spectral density as a scale-location mixture of Gaussians and taking the inverse Fourier transform [WA13]. Since scale-location mixtures of Gaussians are dense in the set of distributions, and can therefore approximate any spectral density, this kernel can approximate any stationary kernel to arbitrary precision. The spectral mixture kernel thus forms a powerful approach to kernel learning, which we discuss further in Section 18.6.5.

18.2.3.1 Random feature kernels

Although the power of kernels resides in the ability to avoid working with featurized representations of the inputs, such kernelized methods can take $O(N^3)$ time, in order to invert the Gram matrix \mathbf{K} . This can make it difficult to use such methods on large scale data. Fortunately, we can approximate the feature map for many kernels using a randomly chosen finite set of M basis functions, thus reducing the cost to $O(NM + M^3)$.

We will show how to do this for shift-invariant kernels by returning to Bochner's theorem in Eq. (18.38):

$$\mathcal{K}(\delta) = \int_{\mathbb{R}^d} p(\omega) e^{j\omega^\top \delta} d\omega \quad (18.39)$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j \sin(\theta)$, and the spectral density $p(\omega)$ is a distribution over frequencies.

In the case of a Gaussian RBF kernel, we have seen that the spectral density is a Gaussian distribution. Hence we can easily compute a Monte Carlo approximation to this integral by sampling random Gaussian vectors. This yields the following approximation: $\mathcal{K}(\mathbf{x}, \mathbf{x}') \approx \phi(\mathbf{x})^\top \phi(\mathbf{x}')$, where

1 the (real-valued) feature vector is given by
2

3

$$\phi(\mathbf{x}) = \sqrt{\frac{1}{D}} [\sin(\mathbf{z}_1^\top \mathbf{x}), \dots, \sin(\mathbf{z}_D^\top \mathbf{x}), \cos(\mathbf{z}_1^\top \mathbf{x}), \dots, \cos(\mathbf{z}_D^\top \mathbf{x})] \quad (18.40)$$

4

5

$$= \sqrt{\frac{1}{D}} [\sin(\mathbf{Z}^\top \mathbf{x}), \cos(\mathbf{Z}^\top \mathbf{x})] \quad (18.41)$$

6

7 Here $\mathbf{Z} = (1/\sigma)\mathbf{G}$, and $\mathbf{G} \in \mathbb{R}^{d \times D}$ is a random Gaussian matrix, where the entries are sampled
8 iid from $\mathcal{N}(0, 1)$. The representation in Equation (18.41) are called **random Fourier features**
9 (**RFF**) [RR08] or “weighted sums of random kitchen sinks” [RR09]. (One can obtain an even better
10 approximation by ensuring that the rows of \mathbf{Z} are random but orthogonal; this is called **orthogonal**
11 **random features** [Yu+16].)

12 One can create similar random feature representations for other kinds of kernels. We can then
13 use such features for supervised learning by defining $f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}\varphi(\mathbf{Z}\mathbf{x}) + \mathbf{b}$, where \mathbf{Z} is a random
14 Gaussian matrix, and the form of φ depends on the chosen kernel. This is equivalent to a one layer
15 MLP with random input-to-hidden weights; since we only optimize the hidden-to-output weights
16 $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$, the model is equivalent to a linear model with fixed random features. If we use enough
17 random features, we can approximate the performance of a kernelized prediction model, but the
18 computational cost is now $O(N)$ rather than $O(N^2)$.

19

20 18.3 GPs with Gaussian likelihoods

21 In this section, we discuss GPs for regression, using a Gaussian likelihood. In this case, all the
22 computations can be performed in closed form, using standard linear algebra methods. We extend
23 this framework to non-Gaussian likelihoods later in the chapter.

24

25 18.3.1 Predictions using noise-free observations

26

27 Suppose we observe a training set $\mathcal{D} = \{(\mathbf{x}_n, y_n) : n = 1 : N\}$, where $y_n = f(\mathbf{x}_n)$ is the noise-free
28 observation of the function evaluated at \mathbf{x}_n . If we ask the GP to predict $f(\mathbf{x})$ for a value of \mathbf{x} that it
29 has already seen, we want the GP to return the answer $f(\mathbf{x})$ with no uncertainty. In other words, it
30 should act as an **interpolator** of the training data. Here we assume the observed function values
31 are noiseless. We will consider the case of noisy observations shortly.

32 Now we consider the case of predicting the outputs for new inputs that may not be in \mathcal{D} . Specifically,
33 given a test set \mathbf{X}_* of size $N_* \times D$, we want to predict the function outputs $\mathbf{f}_* = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_{N_*})]$.
34 By definition of the GP, the joint distribution $p(\mathbf{f}_X, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$ has the following form

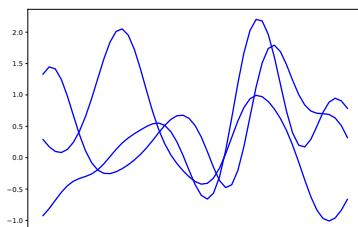
35

$$\begin{pmatrix} \mathbf{f}_X \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\top & \mathbf{K}_{*,*} \end{pmatrix} \right) \quad (18.42)$$

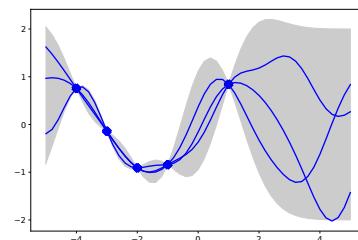
36

37 where $\boldsymbol{\mu}_X = (m(\mathbf{x}_1), \dots, m(\mathbf{x}_N))$, $\boldsymbol{\mu}_* = (m(\mathbf{x}_1^*), \dots, m(\mathbf{x}_{N_*}^*))$, $\mathbf{K}_{X,X} = \mathcal{K}(\mathbf{X}, \mathbf{X})$ is $N \times N$, $\mathbf{K}_{X,*} =$
38 $\mathcal{K}(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$, and $\mathbf{K}_{*,*} = \mathcal{K}(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. See Figure 18.1 for a static illustration,
39 and <http://www.infinitecuriosity.org/vizgp/> for an interactive visualization.

40 By the standard rules for conditioning Gaussians (Section 2.3.3), the posterior has the following
41



(a)



(b)

Figure 18.7: Left: some functions sampled from a GP prior with RBF kernel. Right: some samples from a GP posterior, after conditioning on 5 noise-free observations. The shaded area represents $\mathbb{E}[f(\mathbf{x})] \pm 2\text{std}[f(\mathbf{x})]$. Adapted from Figure 2.2 of [RW06]. Generated by `gprDemoNoiseFree.py`.

form

$$p(f_* | \mathbf{X}_*, \mathcal{D}) = \mathcal{N}(f_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.43)$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_X + \mathbf{K}_{X,*}^\top \mathbf{K}_{X,X}^{-1} (\mathbf{f}_X - \boldsymbol{\mu}_*) \quad (18.44)$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\top \mathbf{K}_{X,X}^{-1} \mathbf{K}_{X,*} \quad (18.45)$$

This process is illustrated in Figure 18.7. On the left we show some samples from the prior, $p(f)$, where we use an RBF kernel (Section 18.2) and a zero mean function. On the right, we show samples from the posterior, $p(f|\mathcal{D})$. We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.

Note that the cost of the above method for sampling N_* points is $O(N_*^3)$. This can be reduced to $O(N_*)$ time using the methods in [Ple+18; Wil+20a].

18.3.2 Predictions using noisy observations

In Section 18.3.1, we showed how to do GP regression when the training data was noiseless. Now let us consider the case where what we observe is a noisy version of the underlying function, $y_n = f(\mathbf{x}_n) + \epsilon_n$, where $\epsilon_n \sim \mathcal{N}(0, \sigma_y^2)$. In this case, the model is not required to interpolate the data, but it must come “close” to the observed data. The covariance of the observed noisy responses is

$$\text{Cov}[y_i, y_j] = \text{Cov}[f_i, f_j] + \text{Cov}[\epsilon_i, \epsilon_j] = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) + \sigma_y^2 \delta_{ij} \quad (18.46)$$

where $\delta_{ij} = \mathbb{I}(i = j)$. In other words

$$\text{Cov}[\mathbf{y} | \mathbf{X}] = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I}_N \quad (18.47)$$

The joint density of the observed data and the latent, noise-free function on the test points is given by

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\top & \mathbf{K}_{*,*} \end{pmatrix}\right) \quad (18.48)$$

1 Hence the posterior predictive density at a set of test points \mathbf{X}_* is
2

$$\underline{3} \quad p(\mathbf{f}_* | \mathcal{D}, \mathbf{X}_*) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.49)$$

$$\underline{5} \quad \boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) \quad (18.50)$$

$$\underline{6} \quad \boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{K}_{X,*} \quad (18.51)$$

8 In the case of a single test input, this simplifies as follows
9

$$\underline{10} \quad p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | m_* + \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X), k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_*) \quad (18.52)$$

11 where $\mathbf{k}_* = [\mathcal{K}(\mathbf{x}_*, \mathbf{x}_1), \dots, \mathcal{K}(\mathbf{x}_*, \mathbf{x}_N)]$ and $k_{**} = \mathcal{K}(\mathbf{x}_*, \mathbf{x}_*)$. If the mean function is zero, we can
12 write the posterior mean as follows:
13

$$\underline{15} \quad \boldsymbol{\mu}_{*|X} = \mathbf{k}_*^\top \underbrace{\mathbf{K}_\sigma^{-1} \mathbf{y}}_{\boldsymbol{\alpha}} = \sum_{n=1}^N \mathcal{K}(\mathbf{x}_*, \mathbf{x}_n) \alpha_n \quad (18.53)$$

18 where
19

$$\underline{20} \quad \mathbf{K}_\sigma = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} \quad (18.54)$$

$$\underline{21} \quad \boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.55)$$

23 Fitting this model amounts to computing $\boldsymbol{\alpha}$ in Equation (18.55). This is usually done by computing
24 the Cholesky decomposition of \mathbf{K}_σ , as described in Section 18.3.6. Once we have computed $\boldsymbol{\alpha}$, we
25 can compute predictions for each test point in $O(N)$ time for the mean, and $O(N^2)$ time for the
26 variance.
27

28 18.3.3 Weight space vs function space 29

30 In this section, we show how Bayesian linear regression is a special case of a GP.

31 Consider the linear regression model $y = f(\mathbf{x}) + \epsilon$, where $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$ and $\epsilon \sim \mathcal{N}(0, \sigma_y^2)$. If we
32 use a Gaussian prior $p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \boldsymbol{\Sigma}_w)$, then the posterior is as follows (see Section 15.2.1 for the
33 derivation):
34

$$\underline{35} \quad p(\mathbf{w} | \mathcal{D}) = \mathcal{N}(\mathbf{w} | \frac{1}{\sigma_y^2} \mathbf{A}^{-1} \boldsymbol{\Phi}^T \mathbf{y}, \mathbf{A}^{-1}) \quad (18.56)$$

37 where $\boldsymbol{\Phi}$ is the $N \times D$ design matrix, and
38

$$\underline{39} \quad \mathbf{A} = \sigma_y^{-2} \boldsymbol{\Phi}^T \boldsymbol{\Phi} + \boldsymbol{\Sigma}_w^{-1} \quad (18.57)$$

41 The posterior predictive distribution for $f_* = f(\mathbf{x}_*)$ is therefore
42

$$\underline{43} \quad p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \frac{1}{\sigma_y^2} \boldsymbol{\phi}_*^\top \mathbf{A}^{-1} \boldsymbol{\Phi}^T \mathbf{y}, \boldsymbol{\phi}_*^\top \mathbf{A}^{-1} \boldsymbol{\phi}_*) \quad (18.58)$$

45 where $\boldsymbol{\phi}_* = \boldsymbol{\phi}(\mathbf{x}_*)$. This views the problem of inference and prediction in **weight space**.
46

We now show that this is equivalent to the predictions made by a GP using a kernel of the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \Sigma_w \phi(\mathbf{x}')$. To see this, let $\mathbf{K} = \Phi \Sigma_w \Phi^\top$, $\mathbf{k}_* = \Phi \Sigma_w \phi_*$, and $k_{**} = \phi_*^\top \Sigma_w \phi_*$. Using this notation, and the matrix inversion lemma, we can rewrite Equation (18.58) as follows

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.59)$$

$$\boldsymbol{\mu}_{*|X} = \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} = \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.60)$$

$$\boldsymbol{\Sigma}_{*|X} = \phi_*^\top \Sigma_w \phi_* - \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \Phi \Sigma_w \phi_* = k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_* \quad (18.61)$$

which matches the results in Equation (18.52), assuming $m(\mathbf{x}) = 0$. A non-zero mean can be captured by adding a constant feature with value 1 to $\phi(\mathbf{x})$.

Thus we can derive a GP from Bayesian linear regression. Note, however, that linear regression assumes $\phi(\mathbf{x})$ is a finite length vector, whereas a GP allows us to work directly in terms of kernels, which may correspond to infinite length feature vectors (see Section 18.2.2). That is, a GP works in **function space**.

18.3.4 Semi-parametric GPs

So far, we have mostly assumed the mean of the GP is 0, and have relied on its interpolation abilities to model the mean function. Sometimes it is useful to fit a global linear model for the mean, and use the GP to model the residual errors, as follows:

$$g(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\beta}^\top \phi(\mathbf{x}) \quad (18.62)$$

where $f(\mathbf{x}) \sim \text{GP}(0, \mathcal{K}(\mathbf{x}, \mathbf{x}'))$, and $\phi()$ are some fixed basis functions. This combines a parametric and a non-parametric model, and is known as a **semi-parametric model**.

If we assume $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{b}, \mathbf{B})$, we can integrate these parameters out to get a new GP [O'H78]:

$$g(\mathbf{x}) \sim \text{GP}(\phi(\mathbf{x})^\top \mathbf{b}, \mathcal{K}(\mathbf{x}, \mathbf{x}') + \phi(\mathbf{x})^\top \mathbf{B} \phi(\mathbf{x}')) \quad (18.63)$$

Let $\mathbf{H}_X = \phi(\mathbf{X})^\top$ be the $D \times N$ matrix of training examples, and $\mathbf{H}_* = \phi(\mathbf{X}_*)^\top$ be the $D \times N_*$ matrix of test examples. The corresponding predictive distribution for test inputs \mathbf{X}_* has the following form [RW06, p28]:

$$\mathbb{E}[g(\mathbf{X}_*) | \mathcal{D}] = \mathbf{H}_{*,*}^\top \mathbf{K}_\sigma^{-1} (\mathbf{y} - \mathbf{H}_X^\top \bar{\boldsymbol{\beta}}) = \mathbb{E}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top \bar{\boldsymbol{\beta}} \quad (18.64)$$

$$\text{Cov}[g(\mathbf{X}_*) | \mathcal{D}] = \text{Cov}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} \mathbf{R} \quad (18.65)$$

$$\bar{\boldsymbol{\beta}} = (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} + \mathbf{B}^{-1} \mathbf{b}) \quad (18.66)$$

$$\mathbf{R} = \mathbf{H}_* - \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{K}_{X,*} \quad (18.67)$$

These results can be interpreted as follows: the mean is the usual mean from the GP, plus a global offset from the linear model, using $\bar{\boldsymbol{\beta}}$; and the covariance is the usual covariance from the GP, plus an additional positive term due to the uncertainty in $\boldsymbol{\beta}$.

In the limit of an uninformative prior for the regression parameters, as $\mathbf{B} \rightarrow \infty \mathbf{I}$, this simplifies to

$$\mathbb{E}[g(\mathbf{X}_*) | \mathcal{D}] = \mathbb{E}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.68)$$

$$\text{Cov}[g(\mathbf{X}_*) | \mathcal{D}] = \text{Cov}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} \mathbf{R} \quad (18.69)$$

1 **18.3.5 Marginal likelihood**

3 Most kernels have some free parameters. For example, the RBF-ARD kernel (Section 18.2.1.1) has
4 the form
5

6
$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp \left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} (x_d - x'_d)^2 \right) = \prod_{d=1}^D \mathcal{K}_{\ell_d}(x_d, x'_d)$$
 (18.70)
7

9 where each ℓ_d is a length scale for feature dimension d . Let these (and the observation noise variance
10 σ_y^2 , if present) be denoted by $\boldsymbol{\theta}$. We can compute the likelihood of these parameters as follows:

12
13
$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = p(\mathcal{D}|\boldsymbol{\theta}) = \int p(\mathbf{y}|\mathbf{f}_X, \boldsymbol{\theta}) p(\mathbf{f}_X|\mathbf{X}, \boldsymbol{\theta}) d\mathbf{f}_X$$
 (18.71)
14

15 Since we are integrating out the function f , we often call $\boldsymbol{\theta}$ hyperparameters, and the quantity $p(\mathcal{D}|\boldsymbol{\theta})$
16 the marginal likelihood.

17 Since f is a GP, we can compute the above integral using the marginal likelihood for the corre-
18 sponding Gaussian. This gives

20
21
$$\log p(\mathcal{D}|\boldsymbol{\theta}) = -\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu}_X)^\top \mathbf{K}_\sigma^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) - \frac{1}{2} \log |\mathbf{K}_\sigma| - \frac{N}{2} \log(2\pi)$$
 (18.72)
22

23 The first term term is the square of the Mahalanobis distance between the observations and the
24 predicted values: better fits will have smaller distance. The second term is the log determinant
25 of the covariance matrix, which measures model complexity: smoother functions will have smaller
26 determinants, so $-\log |\mathbf{K}_\sigma|$ will be larger (less negative) for simpler functions. The marginal likelihood
27 measures the tradeoff between fit and complexity.

28 In Section 18.6.1, we discuss how to learn the kernel parameters from data by maximizing the
29 marginal likelihood wrt $\boldsymbol{\theta}$.

30

31 **18.3.6 Computational and numerical issues**

33 In this section, we discuss computational and numerical issues which arise when implementing the
34 above equations. For notational simplicity, we assume the prior mean is zero, $m(\mathbf{x}) = 0$.

35 The posterior predictive mean is given by $\mu_* = \mathbf{k}_*^\top \mathbf{K}_\sigma^{-1} \mathbf{y}$. For reasons of numerical stability, it is
36 unwise to directly invert \mathbf{K}_σ . A more robust alternative is to compute a Cholesky decomposition,
37 $\mathbf{K}_\sigma = \mathbf{L}\mathbf{L}^\top$, which takes $O(N^3)$ time. Given this, we can compute

39
$$\mu_* = \mathbf{k}_*^\top \mathbf{K}_\sigma^{-1} \mathbf{y} = \mathbf{k}_*^\top \mathbf{L}^{-\top} (\mathbf{L}^{-1} \mathbf{y}) = \mathbf{k}_*^\top \boldsymbol{\alpha}$$
 (18.73)
40

41 Here $\boldsymbol{\alpha} = \mathbf{L}^\top \backslash (\mathbf{L} \backslash \mathbf{y})$, where we have used the backslash operator to represent backsubstitution.

42 We can compute the variance in $O(N^2)$ time for each test case using

44
$$\sigma_*^2 = k_{**} - \mathbf{k}_*^\top \mathbf{L}^{-T} \mathbf{L}^{-1} \mathbf{k}_* = k_{**} - \mathbf{v}^\top \mathbf{v}$$
 (18.74)
45

46 where $\mathbf{v} = \mathbf{L} \backslash \mathbf{k}_*$.

47

Finally, the log marginal likelihood (needed for kernel learning, Section 18.6) can be computed using

$$\log p(\mathbf{y}|\mathbf{X}) = -\frac{1}{2}\mathbf{y}^\top \boldsymbol{\alpha} - \sum_{n=1}^N \log L_{nn} - \frac{N}{2} \log(2\pi) \quad (18.75)$$

We see that overall cost is dominated by $O(N^3)$. We discuss faster, but approximate, methods in Section 18.5.

18.3.7 Kernel ridge regression

The term **ridge regression** refers to linear regression with an ℓ_2 penalty on the regression weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + \lambda \|\mathbf{w}\|_2^2 \quad (18.76)$$

where $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$. The solution for this is

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I} \right)^{-1} \left(\sum_{n=1}^N \mathbf{x}_n y_n \right) \quad (18.77)$$

In this section, we consider a function space version of this:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \lambda \|f\|^2 \quad (18.78)$$

For this to make sense, we have to define the function space \mathcal{F} and the norm $\|f\|$. If we use a function space derived from a positive definite kernel function \mathcal{K} , the resulting method is called **kernel ridge regression** (KRR). We will see that the resulting estimate $f^*(\mathbf{x}_*)$ is equivalent to the posterior mean of a GP. We give the details below.

18.3.7.1 Reproducing kernel Hilbert spaces

In this section, we briefly introduce the relevant mathematical “machinery” needed to explain KRR.

Let $\mathcal{F} = \{f : \mathcal{X} \rightarrow \mathbb{R}\}$ be a space of real-valued functions. Elements of this space (i.e., functions) can be added and scalar multiplied as if they were vectors. That is, if $f \in \mathcal{F}$ and $g \in \mathcal{F}$, then $\alpha f + \beta g \in \mathcal{F}$ for $\alpha, \beta \in \mathbb{R}$. We can also define an **inner product** for \mathcal{F} , which is a mapping $\langle f, g \rangle \in \mathbb{R}$ which satisfies the following:

$$\langle \alpha f_1 + \beta f_2, g \rangle = \alpha \langle f_1, g \rangle + \beta \langle f_2, g \rangle \quad (18.79)$$

$$\langle f, g \rangle = \langle g, f \rangle \quad (18.80)$$

$$\langle f, f \rangle \geq 0 \quad (18.81)$$

$$\langle f, f \rangle = 0 \text{ iff } f(x) = 0 \text{ for all } x \in \mathcal{X} \quad (18.82)$$

1 We define the norm of a function using
2

$$\underline{3} \quad \|\mathbf{f}\| \triangleq \sqrt{\langle \mathbf{f}, \mathbf{f} \rangle} \quad \underline{4} \quad (18.83)$$

5 A function space \mathcal{H} with an inner product operator is called a **Hilbert space**. (We also require
6 that the function space be complete, which means that every Cauchy sequence of functions $f_i \in \mathcal{H}$
7 has a limit that is also in \mathcal{H} .)

8 The most common Hilbert space is the space known as L^2 . To define this, we need to specify a
9 **measure** μ on the input space \mathcal{X} ; this is a function that assigns any (suitable) subset A of \mathcal{X} to a
10 positive number, such as its volume. This can be defined in terms of the density function $w : \mathcal{X} \rightarrow \mathbb{R}$,
11 as follows:

$$\underline{12} \quad \mu(A) = \int_A w(x) dx \quad \underline{13} \quad (18.84)$$

15 Thus we have $\mu(dx) = w(x)dx$. We can now define $L^2(\mathcal{X}, \mu)$ to be the space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$
16 that satisfy
17

$$\underline{18} \quad \int_{\mathcal{X}} f(x)^2 w(x) dx < \infty \quad \underline{19} \quad (18.85)$$

20 This is known as the set of **square-integrable functions**. This space has an inner product defined
21 by
22

$$\underline{23} \quad \langle f, g \rangle = \int_{\mathcal{X}} f(x)g(x)w(x) dx \quad \underline{24} \quad (18.86)$$

26 We define a **Reproducing Kernel Hilbert Space** or **RKHS** as follows. Let \mathcal{H} be a Hilbert
27 space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$. We say that \mathcal{H} is an RKHS endowed with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ if there
28 exists a (symmetric) **kernel function** $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ with the following properties:

- 29 • For every $\mathbf{x} \in \mathcal{X}$, $\mathcal{K}(\mathbf{x}, \cdot) \in \mathcal{H}$.
- 30 • \mathcal{K} satisfies the **reproducing property**:

$$\underline{31} \quad \langle f(\cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = f(\mathbf{x}') \quad \underline{32} \quad (18.87)$$

33 The reason for the term “reproducing property” is as follows. Let $f(\cdot) = \mathcal{K}(\mathbf{x}, \cdot)$. Then we have that
34

$$\underline{35} \quad \langle \mathcal{K}(\mathbf{x}, \cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = \mathcal{K}(\mathbf{x}, \mathbf{x}') \quad \underline{36} \quad (18.88)$$

37 18.3.7.2 Complexity of a function in an RKHS

38 The main utility of RKHS from the point of view of machine learning is that it allows us to define a
39 notion of a function’s “smoothness” or “complexity” in terms of its norm, as we now discuss.

41 Suppose we have a positive definite kernel function \mathcal{K} . From Mercer’s theorem we have $\mathcal{K}(\mathbf{x}, \mathbf{x}') =$
42 $\sum_{i=1}^{\infty} \lambda_i \phi_i(\mathbf{x}) \phi_i(\mathbf{x}')$. Now consider a Hilbert space \mathcal{H} defined by functions of the form $f(\mathbf{x}) =$
43 $\sum_{i=1}^{\infty} f_i \phi_i(\mathbf{x})$, with $\sum_{i=1}^{\infty} f_i^2 / \lambda_i < \infty$. The inner product of two functions in this space is

$$\underline{44} \quad \langle f, g \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i g_i}{\lambda_i} \quad \underline{45} \quad (18.89)$$

46

1 Hence the (squared) norm is given by
2

$$\frac{4}{5} \|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i^2}{\lambda_i} \quad (18.90)$$

7 This is analogous to the quadratic form $\mathbf{f}^\top \mathbf{K}^{-1} \mathbf{f}$ which occurs in some GP objectives (see Equa-
8 tion (18.99)). Thus the smoothness of the function is controlled by the properties of the corresponding
9 kernel.

11 18.3.7.3 Representer theorem

12 In this section, we consider the problem of (regularized) empirical risk minimization in function space.
13 In particular, consider the following problem:

$$\frac{16}{17} f^* = \operatorname{argmin}_{f \in \mathcal{H}_K} \sum_{n=1}^N \ell(y_n, f(\mathbf{x}_n)) + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2 \quad (18.91)$$

19 where \mathcal{H}_K is an RKHS with kernel K and $\ell(y, \hat{y}) \in \mathbb{R}$ is a loss function. Then one can show [KW70;
20 SHS01] the following result:

$$\frac{22}{23} f^*(x) = \sum_{n=1}^N \alpha_n K(x, x_n) \quad (18.92)$$

25 where $\alpha_n \in \mathbb{R}$ are some coefficients that depend on the training data. This is called the **representer
26 theorem**.

27 Now consider the special case where the loss function is squared loss, and $\lambda = \sigma_y^2$. We want to
28 minimize

$$\frac{30}{31} \mathcal{L}(f) = \frac{1}{2\sigma_y^2} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \frac{1}{2} \|f\|_{\mathcal{H}}^2 \quad (18.93)$$

33 Substituting in Equation (18.92), and using the fact that $\langle K(\cdot, \mathbf{x}_i), K(\cdot, \mathbf{x}_j) \rangle = K(\mathbf{x}_i, \mathbf{x}_j)$, we obtain
34

$$\frac{35}{36} \mathcal{L}(f) = \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \|\mathbf{y} - \mathbf{K} \boldsymbol{\alpha}\|^2 \quad (18.94)$$

$$\frac{37}{38} = \frac{1}{2} \boldsymbol{\alpha}^\top (\mathbf{K} + \frac{1}{\sigma_y^2} \mathbf{K}^2) \boldsymbol{\alpha} - \frac{1}{\sigma_y^2} \mathbf{y}^\top \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \mathbf{y}^\top \mathbf{y} \quad (18.95)$$

40 Minimizing this wrt $\boldsymbol{\alpha}$ gives $\hat{\boldsymbol{\alpha}} = (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y}$, which is the same as Equation (18.55). Furthermore,
41 the prediction for a test point is

$$\frac{43}{44} \hat{f}(\mathbf{x}_*) = \mathbf{k}_*^\top \boldsymbol{\alpha} = \mathbf{k}_*^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.96)$$

45 This is known as **kernel ridge regression** [Vov13]. We see that the result matches the posterior
46 predictive mean of a GP in Equation (18.53).

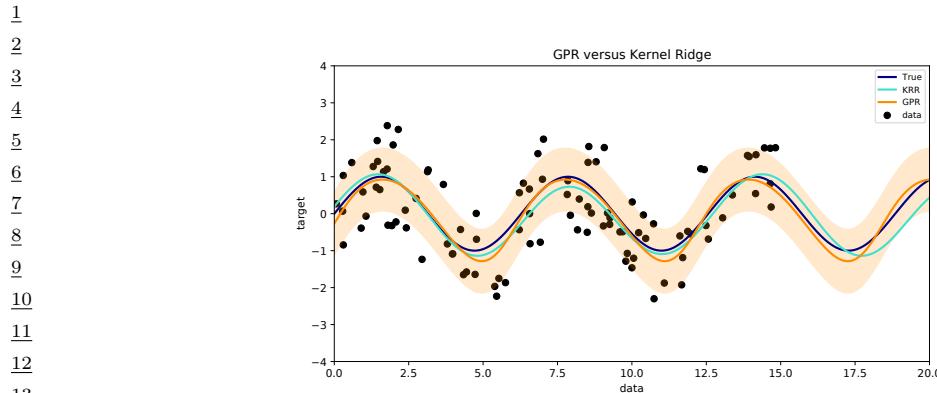


Figure 18.8: Kernel ridge regression (KRR) compared to Gaussian process regression (GPR) using the same kernel. Generated by `krr_vs_gpr.py`.

18.3.7.4 Example of KRR vs GPR

In this section, we compare KRR with GP regression on a simple 1d problem. Since the underlying function is believed to be periodic, we use the periodic kernel from Equation (18.17). To capture the fact that the observations are noisy, we add to this a **white noise kernel**

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sigma_y^2 \delta(\mathbf{x} - \mathbf{x}') \quad (18.97)$$

as in Equation (18.46). Thus there are 3 GP hyper-parameters: the kernel length scale ℓ , the kernel periodicity p , and the noise level σ_y^2 . We can optimize these by maximizing the marginal likelihood using gradient descent (see Section 18.6.1). For KRR, we also have 3 hyper-parameters (ℓ , p and $\lambda = \sigma_y^2$); we optimize these using grid search combined with cross validation (which in general is slower than gradient based optimization). The resulting model fits are shown in Figure 18.8, and are very similar, as is to be expected.

18.4 GPs with non-Gaussian likelihoods

So far, we have focused on GPs for regression using Gaussian likelihoods. In this case, the posterior is also a GP, and all computation can be performed analytically. However, if the likelihood is non-Gaussian, we can no longer compute the posterior exactly. In the sections below, we briefly discuss some approximate inference methods.

18.4.1 Binary classification

In this section, we consider binary classification using GPs. If we use the sigmoid link function, we have $p(y_n = 1 | \mathbf{x}_n) = \sigma(y_n f(\mathbf{x}_n))$. If we assume $y_n \in \{-1, +1\}$, then we have $p(y_n | \mathbf{x}_n) = \sigma(y_n f_n)$, since $\sigma(-z) = 1 - \sigma(z)$. If we use the probit link, we have $p(y_n = 1 | \mathbf{x}_n) = \Phi(y_n f(\mathbf{x}_n))$, where $\Phi(z)$ is the cdf of the standard normal. More generally, let $p(y_n | \mathbf{x}_n) = \text{Ber}(y_n | \varphi(f_n))$. The overall log

<u>1</u>	$\log p(y_i f_i)$	$\frac{\partial}{\partial f_i} \log p(y_i f_i)$	$\frac{\partial^2}{\partial f_i^2} \log p(y_i f_i)$
<u>2</u>	$\log \sigma(y_i f_i)$	$t_i - \pi_i$	$-\pi_i(1 - \pi_i)$
<u>3</u>	$\log \Phi(y_i f_i)$	$\frac{y_i \phi(f_i)}{\Phi(y_i f_i)}$	$-\frac{\phi_i^2}{\Phi(y_i f_i)^2} - \frac{y_i f_i \phi(f_i)}{\Phi(y_i f_i)}$
<u>4</u>			
<u>5</u>			

6 Table 18.1: Likelihood, gradient and Hessian for binary logistic/ probit GP regression. We assume $y_i \in$
7 $\{-1, +1\}$ and define $t_i = (y_i + 1)/2 \in \{0, 1\}$ and $\pi_i = \sigma(f_i)$ for logistic regression, and $\pi_i = \Phi(f_i)$ for probit
8 regression. Also, ϕ and Φ are the pdf and cdf of $\mathcal{N}(0, 1)$. From [RW06, p43].

9
10 joint has the form
11

12
$$\mathcal{L}(\mathbf{f}_X) = \log p(\mathbf{y}|\mathbf{f}_X) + \log p(\mathbf{f}_X|\mathbf{X}) \quad (18.98)$$

13
$$= \log p(\mathbf{y}|\mathbf{f}_X) - \frac{1}{2} \mathbf{f}_X^\top \mathbf{K}_{X,X}^{-1} \mathbf{f}_X - \frac{1}{2} \log |\mathbf{K}_{X,X}| - \frac{N}{2} \log 2\pi \quad (18.99)$$

14 The simplest approach to approximate inference is to use a Laplace approximation (Section 7.4.3).
15 The gradient and Hessian of the log joint are given by

16
$$\nabla \mathcal{L} = \nabla \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} \mathbf{f}_X \quad (18.100)$$

17
$$\nabla^2 \mathcal{L} = \nabla^2 \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} = -\boldsymbol{\Lambda} - \mathbf{K}_{X,X}^{-1} \quad (18.101)$$

18 where $\boldsymbol{\Lambda} \triangleq -\nabla^2 \log p(\mathbf{y}|\mathbf{f}_X)$ is a diagonal matrix, since the likelihood factorizes across examples.
19 Expressions for the gradient and Hessian of the log likelihood for the logit and probit case are shown
20 in Table 18.1. At convergence, the Laplace approximation of the posterior takes the following form:

21
$$p(\mathbf{f}_X|\mathcal{D}) \approx q(\mathbf{f}_X) = \mathcal{N}(\hat{\mathbf{f}}, (\mathbf{K}_{X,X}^{-1} + \boldsymbol{\Lambda})^{-1}) \quad (18.102)$$

22 where $\hat{\mathbf{f}}$ is the MAP estimate. See [RW06, Sec 3.4] for further details.

23 For improved accuracy, we can use variational inference, in which we assume $q(\mathbf{f}_X) = \mathcal{N}(\mathbf{f}_X|\mathbf{m}, \mathbf{S})$;
24 we then optimize \mathbf{m} and \mathbf{S} using (stochastic) gradient descent, rather than assuming \mathbf{S} is the Hessian
25 at the mode. See Section 18.5.4 for the details.

26 Once we have a Gaussian posterior $q(\mathbf{f}_X|\mathcal{D})$, we can then use standard GP prediction to compute
27 $q(f_*|\mathbf{x}_*, \mathcal{D})$. Finally, we can approximate the posterior predictive distribution over binary labels
28 using

29
$$\pi_* = p(y_* = 1|\mathbf{x}_*, \mathcal{D}) = \int p(y_* = 1|f_*) q(f_*|\mathbf{x}_*, \mathcal{D}) df_* \quad (18.103)$$

30 This 1d integral can be computed using the probit approximation from Section 15.3.5. In this case
31 we have $\pi_* \approx \sigma(\kappa(v)\mathbb{E}[f_*])$, where $v = \mathbb{V}[f_*]$ and $\kappa^2(v) = (1 + \pi v/8)^{-1}$.

32 In Figure 18.9, we show a synthetic binary classification problem in 2d. We use an SE kernel.
33 On the left, we show predictions using hyper-parameters set by hand; we use a short length scale,
34 hence the very sharp turns in the decision boundary. On the right, we show the predictions using the
35 learned hyper-parameters; the model favors more parsimonious explanation of the data.

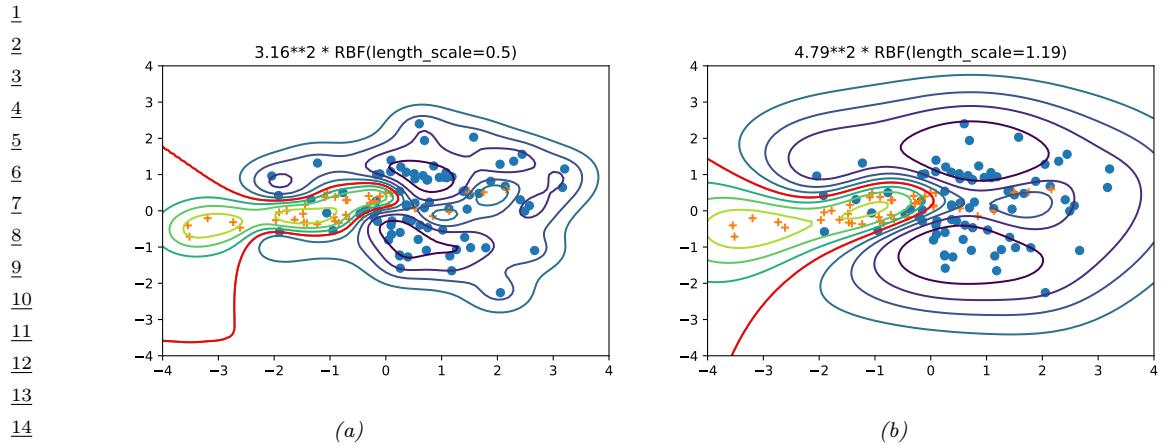


Figure 18.9: Contours of the posterior predictive probability for a binary classifier generated by a GP with an SE kernel. (a) Manual kernel parameters: short length scale, $\ell = 0.5$, variance $3.16^2 \approx 9.98$. (b) Learned kernel parameters: long length scale, $\ell = 1.19$, variance $4.79^2 \approx 22.9$. Generated by [gpc_demo_2d_sklearn.py](#).

18.4.2 Multi-class classification

The multi-class case is somewhat harder, since the function now needs to return a vector of C logits to get $p(y_n|\mathbf{x}_n) = \text{Cat}(y_n|\sigma(\mathbf{f}_n))$, where $\mathbf{f}_n = (f_n^1, \dots, f_n^C)$. It is standard to assume that $f^c \sim \text{GP}(0, \mathcal{K}_c)$. Thus we have one latent function per class, which are a priori independent, and which may use different kernels.

We can derive a Laplace approximation for this model as discussed in [RW06, Sec 3.5]. Alternatively, we can use a variational approach, using the local variational bound to the multinomial softmax in [Cha12]. An alternative variational method, based on data augmentation with auxiliary variables, is described in [Wen+19b; Liu+19a; GFWO20].

31

18.4.3 GPs for Poisson regression (Cox process)

In this section, we illustrate Poisson regression where the underlying log rate function is modeled by a GP. This is known as a **Cox process**. We can perform approximate posterior inference in this model using Laplace, MCMC or SVI (stochastic variational inference). In Figure 18.10 we give a 1d example, where we use a Matern $\frac{5}{2}$ kernel. We apply MCMC and SVI. In the VI case, we additionally have to specify the form of the posterior; we use a Gaussian approximation for the variational GP posterior $p(\mathbf{f}|\mathbf{X}, \mathbf{y})$, and a point estimate for the kernel parameters.

An interesting application of this is to spatial **disease mapping**. For example, [VPV10] discuss the problem of modeling the relative risk of heart attack in different regions in Finland. The data consists of the heart attacks in Finland from 1996-2000 aggregated into 20km x 20km lattice cells. The likelihood has the following form: $y_n \sim \text{Poi}(e_n r_n)$, where e_n is the known expected number of deaths (related to the population of cell n and the overall death rate), and r_n is the **relative risk** of cell n which we want to infer. Since the data counts are small, we regularize the problem by sharing information with spatial neighbors. Hence we assume $f \triangleq \log(r) \sim \text{GP}(0, \mathcal{K})$. We use a Matern

47

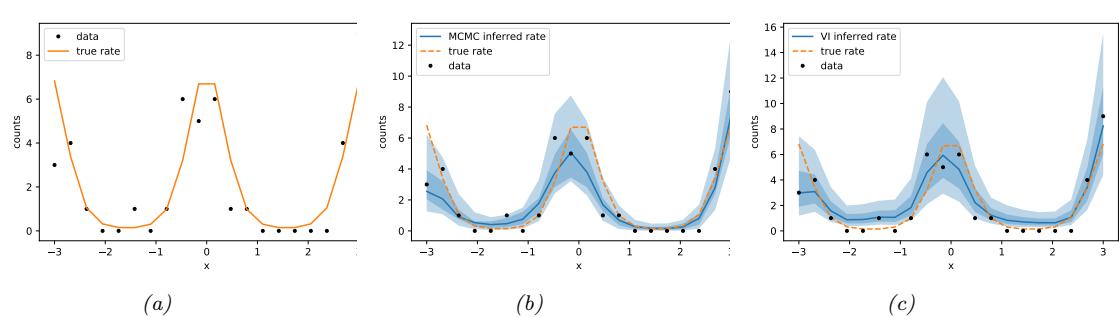


Figure 18.10: Poisson regression with a GP. (a) Observed data (black dots) and true log rate function (yellow line). (b) Posterior predictive distribution (shading shows 1 and 2 σ bands) from MCMC. (c) Posterior predictive distribution from SVI. Generated by [gp_poisson_1d.ipynb](#).

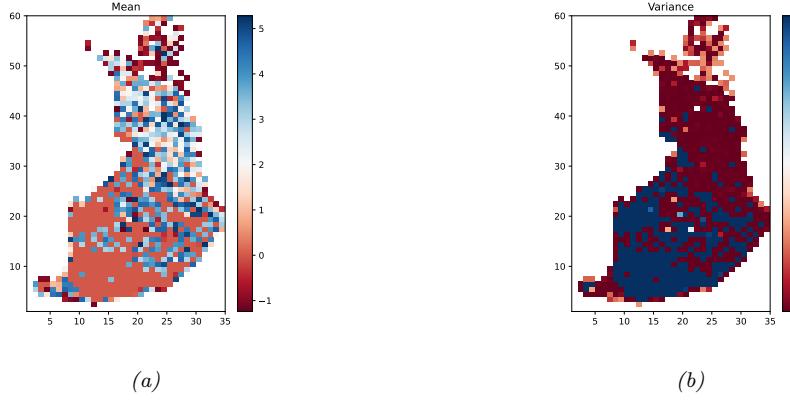


Figure 18.11: We show the relative risk of heart disease in Finland using a Poisson GP fit to 911 data points. Left: posterior mean. Right: posterior variance. Generated by [gp_spatial_demo.py](#).

kernel (Section 18.2.1.1) with $\nu = 3/2$, and a length scale and magnitude that are estimated from data.

Figure 18.11 gives an example of this method in action (using Laplace approximation). On the left we plot the posterior mean relative risk (RR), and on the right, the posterior variance. We see that the RR is higher in Eastern Finland, which is consistent with other studies. We also see that the variance in the North is higher, since there are fewer people living there.

18.5 Scaling GP inference to large datasets

In Section 18.3.6, we saw that the best way to perform GP inference and training is to compute a Cholesky decomposition of the $N \times N$ Gram matrix. Unfortunately, this takes $O(N^3)$ time. In this

Method	Cost	Section
Cholesky	$O(N^3)$	Section 18.3.6
Conj. Grad.	$O(CN^2)$	Section 18.5.5
Inducing	$O(NM^2 + M^3 + DNM)$	Section 18.5.3
Variational	$O(NM^2 + M^3 + DNM)$	Section 18.5.4
SVGP	$O(BM^2 + M^3 + DNM)$	Section 18.5.4.3
KISS-GP	$O(CN + CDM^D \log M)$	Section 18.5.5.3
SKIP	$O(DLN + DLM \log M + L^3 N \log D + CL^2 N)$	Section 18.5.5.3

Table 18.2: Summary of time to compute the log marginal likelihood of a GP regression model. Notation: N is number of training examples, M is number of inducing points, B is size of minibatch, D is dimensionality of input vectors (assuming $\mathcal{X} = \mathbb{R}^D$), C is number of conjugate gradient iterations. L is number of Lanczos iterations. Based on Table 2 of [Gar+18a].

section, we discuss methods to scale up GPs to handle large N . See Table 18.2 for a summary, and [Liu+20c] for more details.¹

18.5.1 Subset of data

The simplest approach to speeding up GP inference is to throw away some of the data. Suppose we keep a subset of M examples. In this case, exact inference will take $O(M^3)$ time. This is called the **subset-of-data** approach.

The key question is: how should we choose the subset? The simplest approach is to pick random examples (this method was recently analysed in [HIY19]). However, intuitively it makes more sense to try to pick a subset that in some sense “covers” the original data, so it contains approximately the same information (up to some tolerance) without the redundancy. Clustering algorithms are one heuristic approach, but we can also use coresets methods, which can provably find such an information-preserving subset (see e.g., [Hug+19] for an application of this idea to GPs).

18.5.1.1 Informative vector machine

Clustering and coresets methods are unsupervised, in that they only look at the features \mathbf{x}_i and not the labels y_i , which can be suboptimal. The **informative vector machine** [HLS03] uses a greedy strategy to iteratively add the labeled example (\mathbf{x}_j, y_j) that maximally reduces the entropy of the function’s posterior, $\Delta_j = \mathbb{H}(p(f_j)) - \mathbb{H}(p^{\text{new}}(f_j))$, where $p^{\text{new}}(f_j)$ is the posterior of f at \mathbf{x}_j after conditioning on y_j . (This is very similar to active learning.) To compute Δ_j , let $p(f_j) = \mathcal{N}(\mu_j, v_j)$, and $p(f_j|y_j) \propto p(f_j)\mathcal{N}(y_j|f_j, \sigma^2) = \mathcal{N}(f_j|\mu_j^{\text{new}}, v_j^{\text{new}})$, where $(v_j^{\text{new}})^{-1} = v_j^{-1} + \sigma^{-2}$. Since $\mathbb{H}(\mathcal{N}(\mu, v)) = \log(2\pi ev)/2$, we have $\Delta_j = 0.5 \log(1 + v_j/\sigma^2)$. Since this is a monotonic function of v_j , we can maximize it by choosing the site with the largest variance. (In fact, entropy is a submodular function, so we can use submodular optimization algorithms to improve on the IVM, as shown in [Kra+08].)

1. We focus on efficient methods for evaluating the marginal likelihood and the posterior predictive distribution. For an efficient method for sampling a function from the posterior, see [Wil+20a].

1

18.5.1.2 Discussion

2

3 The main problem with the subset of data approach is that it ignores some of the data, which can
4 reduce predictive accuracy and increase uncertainty about the true function. Fortunately there
5 are other scalable methods that avoid this problem, essentially by approximately representing (or
6 compressing) the training data, as we discuss below.

7

8

18.5.2 Nyström approximation

9

10 Suppose we had a rank M approximation to the $N \times N$ matrix gram matrix of the following form:

11 $\mathbf{K}_{X,X} \approx \mathbf{U}\Lambda\mathbf{U}^T$ (18.104)

12

13 where Λ is a diagonal matrix of the M leading eigenvalues, and \mathbf{U} is the matrix of the corresponding
14 M eigenvectors, each of size N . In this case, we can use the matrix inversion lemma to write

15

16 $\mathbf{K}_\sigma^{-1} = (\mathbf{K}_{X,X} + \sigma^2 \mathbf{I}_N)^{-1} \approx \sigma^{-2} \mathbf{I}_N + \sigma^{-2} \mathbf{U}(\sigma^2 \Lambda^{-1} + \mathbf{U}^T \mathbf{U})^{-1} \mathbf{U}^T$ (18.105)

17

18 which takes $O(NM^2)$ time. Similarly, one can show (using the Sylvester determinant lemma) that

19 $|\mathbf{K}_\sigma| \approx |\Lambda| |\sigma^2 \Lambda^{-1} + \mathbf{U}^T \mathbf{U}|$ (18.106)

20

21 which also takes $O(NM^2)$ time.

22 Unfortunately, directly computing such an eigendecomposition takes $O(N^3)$ time, which does not
23 help. However, suppose we pick a subset Z of $M < N$ points. We can partition the Gram matrix as
24 follows (where we assume the chosen points come first, and then the remaining points):

25 $\mathbf{K}_{X,X} = \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,X-Z} \\ \mathbf{K}_{X-Z,Z} & \mathbf{K}_{X-Z,X-Z} \end{pmatrix}$ (18.107)

26

27 Let $\mathbf{K}_{Z,X}$ denote the top $M \times N$ block, and $\mathbf{K}_{X,Z}$ denote its transpose. We now compute an
28 eigendecomposition of $\mathbf{K}_{Z,Z}$ to get the eigenvalues $\{\lambda_i\}_{i=1}^M$ and eigenvectors $\{\mathbf{u}_i\}_{i=1}^M$. We now use
29 these to approximate the full matrix as shown below, where the scaling constants are chosen so that
30 $\|\tilde{\mathbf{u}}_i\| \approx 1$:

31 $\tilde{\lambda}_i \triangleq \frac{N}{M} \lambda_i$ (18.108)

32

33 $\tilde{\mathbf{u}} \triangleq \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{X,Z} \mathbf{u}_i$ (18.109)

34

35 $\mathbf{K}_{X,X} \approx \sum_{i=1}^M \tilde{\lambda}_i \tilde{\mathbf{u}}_i \tilde{\mathbf{u}}_i^T$ (18.110)

36

37 $= \sum_{i=1}^M \frac{N}{M} \lambda_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{X,Z} \mathbf{u}_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{u}_i^T \mathbf{K}_{X,Z}^T$ (18.111)

38

39 $= \mathbf{K}_{X,Z} \left(\sum_{i=1}^M \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^T \right) \mathbf{K}_{Z,X}$ (18.112)

40

41 $= \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}$ (18.113)

42

¹ This is known as the **Nyström approximation** [WS01]. If we define
²

$$\mathbf{Q}_{A,B} \triangleq \mathbf{K}_{A,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,B} \quad (18.114)$$

⁵ then we can write the approximate Gram matrix as $\mathbf{Q}_{X,X}$. We can then replace \mathbf{K}_σ with $\hat{\mathbf{Q}}_{X,X} =$
⁶ $\mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$. Computing the eigendecomposition takes $O(M^3)$ time, and computing $\hat{\mathbf{Q}}_{X,X}^{-1}$ takes
⁷ $O(NM^2)$ time. Thus complexity is now linear in N instead of cubic.
⁸

⁹ If we are approximating *only* $\hat{\mathbf{K}}_{X,X}$ in $\boldsymbol{\mu}_{*|X}$ in Equation (18.50) and $\boldsymbol{\Sigma}_{*|X}$ in Equation (18.51)
¹⁰ $\hat{\mathbf{Q}}_{X,X}$, then this is inconsistent with the other un-approximated kernel function evaluations in these
¹¹ formulae, and can result in the predictive variance being negative. One solution to this is to use the
¹² same \mathbf{Q} approximation for all terms.

¹³ 18.5.3 Inducing point methods

¹⁵ In this section, we discuss an approximation method based on **inducing points**, also called **pseudo**
¹⁶ **inputs**, which are like a learned summary of the training data that we can condition on, rather than
¹⁷ conditioning on all of it.
¹⁸

¹⁹ Let \mathbf{X} be the observed inputs, and $\mathbf{f}_X = f(\mathbf{X})$ be the unknown vector of function values (for which
²⁰ we have noisy observations \mathbf{y}). Let \mathbf{f}_* be the unknown function values at one or more test points
²¹ \mathbf{X}_* . Finally, let us assume we have M additional inputs, \mathbf{Z} , with unknown function values \mathbf{f}_Z (often
²² denoted by \mathbf{u}). The exact joint prior has the form

$$\frac{23}{24} p(\mathbf{f}_X, \mathbf{f}_*) = \int p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) d\mathbf{f}_Z = \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) d\mathbf{f}_Z = \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}\right) \quad (18.115)$$

²⁵ (We write $p(\mathbf{f}_X, \mathbf{f}_*)$ instead of $p(\mathbf{f}_X, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$, since the inputs can be thought of as just indices
²⁶ into the random function f .)

²⁸ We will choose \mathbf{f}_Z in such a way that it acts as a sufficient statistic for the data, so that we can
²⁹ predict \mathbf{f}_* just using \mathbf{f}_Z instead of \mathbf{f}_X , i.e., we assume $\mathbf{f}_* \perp \mathbf{f}_X | \mathbf{f}_Z$. Thus we approximate the prior
³⁰ as follows:

$$\frac{31}{32} p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) = p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) \approx p(\mathbf{f}_* | \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) \quad (18.116)$$

³³ See Figure 18.12 for an illustration of this assumption, and Section 18.5.3.4 for details on how to
³⁴ choose the inducing set \mathbf{Z} . (Note that this method is often called a “**sparse GP**”, because it makes
³⁵ predictions for \mathbf{f}_* using a subset of the training data, namely \mathbf{f}_Z , instead of all of it, \mathbf{f}_X .)

³⁶ From this, we can derive the following train and test conditionals

$$\frac{38}{39} p(\mathbf{f}_X | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{f}_Z, \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.117)$$

$$\frac{39}{40} p(\mathbf{f}_* | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_* | \mathbf{K}_{*,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{f}_Z, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}) \quad (18.118)$$

⁴¹ The above equations can be seen as exact inference on noise-free observations \mathbf{f}_Z . To gain
⁴² computational speedups, we will make further approximations of the form $\tilde{\mathbf{Q}}_{X,X} \approx \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}$
⁴³ and $\tilde{\mathbf{Q}}_{*,*} \approx \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$. We consider various choices for these values below. All of these choices
⁴⁴ result in an initial training cost of $O(M^3 + NM^2)$, and then take $O(M)$ time for the predictive mean
⁴⁵ for each test case, and $O(M^2)$ time for the predictive variance. (Compare this to $O(N^3)$ training
⁴⁶ time and $O(N)$ and $O(N^2)$ testing time for exact inference.)

⁴⁷

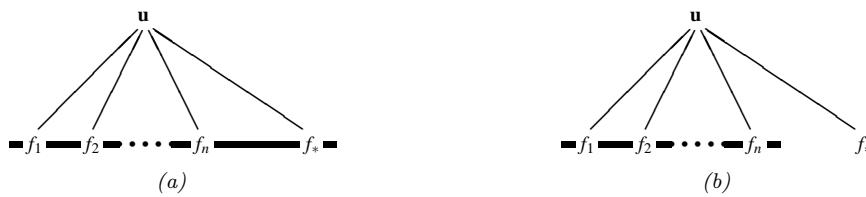


Figure 18.12: Illustration of the graphical model for a GP on n observations, $\mathbf{f}_{1:n}$, and one test case, f_* , with inducing variables \mathbf{u} . The thick lines indicate that all variables are fully interconnected. The observations y_i (not shown) are locally connected to each f_i . (a) no approximations are made. (b) we assume f_* is conditionally independent of \mathbf{f}_X given \mathbf{u} . From Figure 1 of [QCR05]. Used with kind permission of Joaquin Quinonero-Candela.

18.5.3.1 SOR/ DIC

Suppose we assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$ and $\tilde{\mathbf{Q}}_{*,*} = \mathbf{0}$, so the conditionals are deterministic. This is called the **deterministic inducing conditional (DIC)** approximation [QCR05], or the **subset of regressors (SOR)** approximation [Sil85; SB01]. The corresponding joint prior has the form

$$q_{\text{SOR}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{Q}_{*,*} \end{pmatrix}) \quad (18.119)$$

Consequently the predictive distribution is

$$q_{\text{SOR}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{Q}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.120)$$

$$= \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,*}) \quad (18.121)$$

where we have defined $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$, and $\Sigma = (\sigma^{-2} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} + \mathbf{K}_{Z,Z})^{-1}$.

This predictive distribution is equivalent to the usual one for GPs except we have replaced $\mathbf{K}_{X,X}$ by $\mathbf{Q}_{X,X}$. This is equivalent to performing GP inference with the following kernel function

$$\mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) = \mathcal{K}(\mathbf{x}_i, \mathbf{Z}) \mathbf{K}_{Z,Z}^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_j) \quad (18.122)$$

The kernel matrix has rank M , so the GP is degenerate. Furthermore, the kernel will be near 0 when \mathbf{x}_i or \mathbf{x}_j is far from one of the chosen points \mathbf{Z} , which can result in an underestimate of the predictive variance.

18.5.3.2 DTC

One way to overcome the overconfidence of DIC is to only assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$, but let $\tilde{\mathbf{Q}}_{*,*} = \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ be exact. This is called the **deterministic training conditional** or **DTC** method [SWL03].

The corresponding joint prior has the form

$$q_{\text{dtc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.123)$$

1 Hence the predictive distribution becomes
2

$$\underline{3} \quad q_{\text{dtc}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.124)$$

$$\underline{4} \quad = \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*} + \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,*}) \quad (18.125)$$

5 The predictive mean is the same as in SOR, but the variance is larger (since $\mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ is positive
6 definite) due to the uncertainty of \mathbf{f}_* given \mathbf{f}_Z .
7

8 18.5.3.3 FITC

9 A widely used approximation assumes $q(\mathbf{f}_X | \mathbf{f}_Z)$ is fully factorized, i.e,
10

$$\underline{11} \quad q(\mathbf{f}_X | \mathbf{f}_Z) = \prod_{n=1}^N p(f_n | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{f}_Z, \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X})) \quad (18.126)$$

12 This is called the **fully independent training conditional** or **FITC** assumption, and was first
13 proposed in [SG06a]. This throws away less uncertainty than the SOR and DTC methods, since it
14 does not make any deterministic assumptions about the relationship between \mathbf{f}_X and \mathbf{f}_Z .
15

The joint prior has the form
16

$$\underline{17} \quad q_{\text{fitc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} - \text{diag}(\mathbf{Q}_{X,X} - \mathbf{K}_{X,X}) & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.127)$$

The predictive distribution for a single test case is given by
18

$$\underline{19} \quad q_{\text{fitc}}(f_* | \mathbf{y}) = \mathcal{N}(f_* | \mathbf{k}_{*,Z} \Sigma \mathbf{K}_{Z,X} \Lambda^{-1} \mathbf{y}, k_{**} - q_{**} + \mathbf{k}_{*,Z} \Sigma \mathbf{k}_{Z,*}) \quad (18.128)$$

20 where $\Lambda \triangleq \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N)$, and $\Sigma \triangleq (\mathbf{K}_{Z,Z} + \mathbf{K}_{Z,X} \Lambda^{-1} \mathbf{K}_{X,Z})^{-1}$. If we have a batch
21 of test cases, we can assume they are conditionally independent (an approach known as **fully
22 independent conditional** or **FIC**), and multiply the above equation.
23

24 The computational cost is the same as for SOR and DTC, but the approach avoids some of the
25 pathologies due to a non-degenerate kernel. In particular, one can show that the FIC method is
26 equivalent to exact GP inference with the following non-degenerate kernel:
27

$$\underline{28} \quad \mathcal{K}_{\text{fic}}(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i = j \\ \mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i \neq j \end{cases} \quad (18.129)$$

29 18.5.3.4 Learning the inducing points

30 So far, we have not specified how to choose the inducing points or pseudo inputs \mathbf{Z} . We can treat
31 these like kernel hyperparameters, and choose them so as to maximize the log marginal likelihood,
32 given by
33

$$\underline{34} \quad \log q(\mathbf{y} | \mathbf{X}, \mathbf{Z}) = \log \int \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{f}_Z) p(\mathbf{f}_Z | \mathbf{Z}) d\mathbf{f}_Z d\mathbf{f} \quad (18.130)$$

$$\underline{35} \quad = \log \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{Z}) d\mathbf{f}_X \quad (18.131)$$

$$\underline{36} \quad = -\frac{1}{2} \log |\mathbf{Q}_{X,X} + \Lambda| - \frac{1}{2} \mathbf{y}^\top (\mathbf{Q}_{X,X} + \Lambda)^{-1} \mathbf{y} - \frac{n}{2} \log(2\pi) \quad (18.132)$$

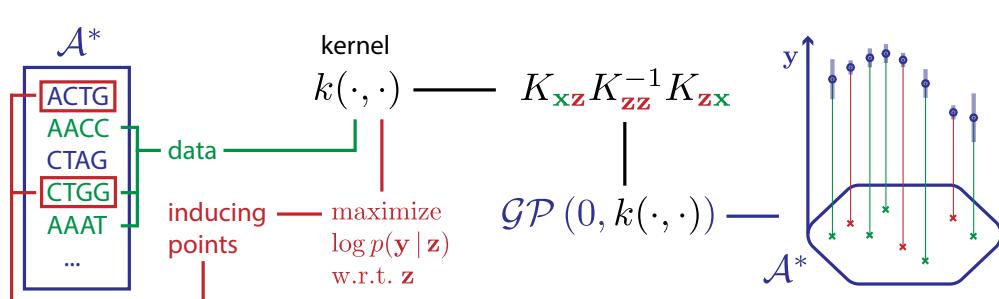


Figure 18.13: Illustration of how to choose inducing points from a discrete input domain (here DNA sequences of length 4) to maximize the log marginal likelihood. From Figure 1 of [For+18a]. Used with kind permission of Vincent Fortuin.

where the definition of Λ depends on the method, namely $\Lambda_{\text{SOR}} = \Lambda_{\text{dtc}} = \sigma^2 \mathbf{I}_N$, and $\Lambda_{\text{fitc}} = \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) + \sigma^2 \mathbf{I}_N$.

If the input domain is \mathbb{R}^d , we can optimize $\mathbf{Z} \in \mathbb{R}^{Md}$ using gradient methods. However, one of the appeals of kernel methods is that they can handle structured inputs, such as strings and graphs (see Section 18.2.1.4). In this case, we cannot use gradient methods to select the inducing points. A simple approach is to select the inducing points from the training set, as in the subset of data approach in Section 18.5.1, or using the efficient selection mechanism in [Cao+15]. However, we can also use discrete optimization methods, such as simulated annealing (Section 12.2.5), as discussed in [For+18a]. See Figure 18.13 for an illustration.

18.5.4 Sparse variational methods

In this section, we discuss a variational approach to GP inference that is similar to the inducing point methods in Section 18.5.3, but which generalizes it to also handle non-conjugate likelihoods. This is called the **sparse variational GP** or **SVGP** approximation. For more details, see [Lei+20]. See also [WKS21] for connections between SVGP and the Nyström method.)

To explain the idea behind SVGP, let us assume, for simplicity, that the function f is defined over a finite set \mathcal{X} of possible inputs, which we partition into three subsets: the training set \mathbf{X} , a set of inducing points \mathbf{Z} , and all other points (which we can think of as the test set), \mathbf{X}_* . (We assume these sets are disjoint.) Let \mathbf{f}_X , \mathbf{f}_Z and \mathbf{f}_* represent the corresponding unknown function values on these points, and let $\mathbf{f} = [\mathbf{f}_X, \mathbf{f}_Z, \mathbf{f}_*]$ be all the unknowns. (Here we work with a fixed-length vector \mathbf{f} , but the result generalizes to Gaussian processes, as explained in [Mat+16].) We assume the function is sampled from a GP, so $p(\mathbf{f}) = \mathcal{N}(\mathbf{m}(\mathcal{X}), \mathcal{K}(\mathcal{X}, \mathcal{X}))$.

The inducing point methods in Section 18.5.3 approximates the GP prior by assuming $p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \approx p(\mathbf{f}_* | \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z)$. The inducing points \mathbf{f}_Z are chosen to maximize the likelihood of the observed data. We then perform exact inference in this approximate model. By contrast, in this section, we will keep the model unchanged, but we will instead approximate the posterior $p(\mathbf{f} | \mathbf{y})$ using variational inference. This approach is known as the **variational free energy (VFE)** method for GPs [Tit09; Mat+16]; it is also called SVGP.

In the VFE view, the inducing points \mathbf{Z} and inducing variables \mathbf{f}_Z are variational parameters,

¹ rather than model parameters, which avoids the risk of overfitting. Furthermore, one can show that
² as the number of inducing points m increases, the quality of the posterior consistently improves,
³ eventually recovering exact inference. By contrast, in the classical inducing point method, increasing
⁴ m does not always result in better performance [BWR16].
⁵

⁶ In more detail, the VFE approach tries to find an approximate posterior $q(\mathbf{f})$ to minimize
⁷ $D_{\text{KL}}(q(\mathbf{f})\|p(\mathbf{f}|\mathbf{y}))$. The key assumption is that $q(\mathbf{f}) = q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) = p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_Z)$, where
⁸ $p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z)$ is computed exactly using the GP prior, and $q(\mathbf{f}_Z)$ is learned, by minimizing $\mathcal{K}(q) =$
⁹ $D_{\text{KL}}(q(\mathbf{f})\|p(\mathbf{f}|\mathbf{y}))$.² Intuitively, $q(\mathbf{f}_Z)$ acts as a “bottleneck” which “absorbs” all the observations
¹⁰ from \mathbf{y} ; posterior predictions for elements of \mathbf{f}_X or \mathbf{f}_* are then made via their dependence on \mathbf{f}_Z ,
¹¹ rather than their dependence on each other.

¹² We can derive the form of the loss, which is used to compute the posterior $q(\mathbf{f}_Z)$, as follows:

$$\mathcal{K}(q) = D_{\text{KL}}(q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z)\|p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z|\mathbf{y})) \quad (18.133)$$

$$= \int q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \log \frac{q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z)}{p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z|\mathbf{y})} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.134)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_Z) \log \frac{\cancel{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)}q(\mathbf{f}_Z)p(\mathbf{y})}{\cancel{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z)p(\mathbf{y} | \mathbf{f}_X)}} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.135)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z)p(\mathbf{y})}{p(\mathbf{f}_Z)p(\mathbf{y} | \mathbf{f}_X)} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.136)$$

$$= \int q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z)}{p(\mathbf{f}_Z)} d\mathbf{f}_Z - \int p(\mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_Z) \log p(\mathbf{y} | \mathbf{f}_X) d\mathbf{f}_X d\mathbf{f}_Z + C \quad (18.137)$$

$$= D_{\text{KL}}(q(\mathbf{f}_Z)\|p(\mathbf{f}_Z)) - \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] + C \quad (18.138)$$

²⁶ where $C = \log p(\mathbf{y})$ is an irrelevant constant.

²⁷ We can alternatively write the objective as an evidence lower bound that we want to maximize:

$$\log p(\mathbf{y}) = \mathcal{K}(q) + \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z)\|p(\mathbf{f}_Z)) \quad (18.139)$$

$$\geq \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z)\|p(\mathbf{f}_Z)) \triangleq \mathcal{L}(q) \quad (18.140)$$

³² Now suppose we choose a Gaussian posterior approximation, $q(\mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_Z | \mathbf{m}, \mathbf{S})$. Since $p(\mathbf{f}_Z) =$
³³ $\mathcal{N}(\mathbf{f}_Z | \mathbf{0}, \mathcal{K}(\mathbf{Z}, \mathbf{Z}))$, we can compute the KL term in closed form using the formula for KL divergence
³⁴ between Gaussians (Equation (5.66)).
³⁵

³⁶ As for the expected log-likelihood term, we need to compute $q(\mathbf{f}_X)$. Since $q(\mathbf{f}_Z)$ is Gaussian, this
³⁷ can be done in closed form as follows:

$$q(\mathbf{f}_X | \mathbf{m}, \mathbf{S}) = \int p(\mathbf{f}_X | \mathbf{f}_Z, \mathbf{X}, \mathbf{Z})q(\mathbf{f}_Z | \mathbf{m}, \mathbf{S}) d\mathbf{f}_Z = \mathcal{N}(\mathbf{f}_X | \tilde{\mathbf{\mu}}, \tilde{\mathbf{\Sigma}}) \quad (18.141)$$

$$\tilde{\mathbf{\mu}}_i = \mathbf{m}(\mathbf{x}_i) + \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathbf{m} - \mathbf{m}(\mathbf{Z})) \quad (18.142)$$

$$\tilde{\mathbf{\Sigma}}_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) - \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathcal{K}(\mathbf{Z}, \mathbf{Z}) - \mathbf{S}) \boldsymbol{\alpha}(\mathbf{x}_j) \quad (18.143)$$

$$\boldsymbol{\alpha}(\mathbf{x}_i) = \mathcal{K}(\mathbf{Z}, \mathbf{Z})^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_i) \quad (18.144)$$

⁴⁵
⁴⁶ 2. One can show that $D_{\text{KL}}(q(\mathbf{f})\|p(\mathbf{f}|\mathbf{y})) = D_{\text{KL}}(q(\mathbf{f}_X, \mathbf{f}_Z)\|p(\mathbf{f}_X, \mathbf{f}_Z|\mathbf{y}))$, which is the original objective from [Tit09].
⁴⁷

Hence $q(f_n) = \mathcal{N}(f_n | \tilde{\mu}_n, \tilde{\Sigma}_{nn})$, which we can use to compute the expected log likelihood:

$$\mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] = \sum_{n=1}^N \mathbb{E}_{q(f_n)} [\log p(y_n | f_n)] \quad (18.145)$$

We discuss how to compute the expected loglikelihood below.

18.5.4.1 Gaussian likelihood

If we have a Gaussian observation model, we can compute the expected log likelihood in closed form. In particular, if we assume $m(\mathbf{x}) = \mathbf{0}$, we have

$$\mathbb{E}_{q(f_n)} [\log \mathcal{N}(y_n | f_n, \beta^{-1})] = \log \mathcal{N}(y_n | \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1}) - \frac{1}{2} \beta \tilde{k}_{nn} - \frac{1}{2} \text{tr}(\mathbf{S} \boldsymbol{\Lambda}_n) \quad (18.146)$$

where $\tilde{k}_{nn} = k_{nn} - \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1} \mathbf{k}_n$, \mathbf{k}_n is the n 'th column of $\mathbf{K}_{Z,X}$ and $\boldsymbol{\Lambda}_n = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{k}_n \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1}$.

Hence the overall ELBO has the form

$$\mathcal{L}(q) = \log \mathcal{N}(\mathbf{y} | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1} \mathbf{I}_N) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{S} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}) \quad (18.147)$$

$$- \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.148)$$

To compute the gradients of this, we leverage the following result [OA09]:

$$\frac{\partial}{\partial \mu} \mathbb{E}_{\mathcal{N}(x | \mu, \sigma^2)} [h(x)] = \mathbb{E}_{\mathcal{N}(x | \mu, \sigma^2)} \left[\frac{\partial}{\partial x} h(x) \right] \quad (18.149)$$

$$\frac{\partial}{\partial \sigma^2} \mathbb{E}_{\mathcal{N}(x | \mu, \sigma^2)} [h(x)] = \frac{1}{2} \mathbb{E}_{\mathcal{N}(x | \mu, \sigma^2)} \left[\frac{\partial^2}{\partial x^2} h(x) \right] \quad (18.150)$$

We then substitute $h(x)$ with $\log p(y_n | f_n)$. Using this, one can show

$$\nabla_{\mathbf{m}} \mathcal{L}(q) = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} - \boldsymbol{\Lambda} \mathbf{m} \quad (18.151)$$

$$\nabla_{\mathbf{S}} \mathcal{L}(q) = \frac{1}{2} \mathbf{S}^{-1} - \frac{1}{2} \boldsymbol{\Lambda} \quad (18.152)$$

Setting the derivatives to zero gives the optimal solution:

$$\mathbf{S} = \boldsymbol{\Lambda}^{-1} \quad (18.153)$$

$$\boldsymbol{\Lambda} = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} + \mathbf{K}_{Z,Z}^{-1} \quad (18.154)$$

$$\mathbf{m} = \beta \boldsymbol{\Lambda}^{-1} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} \quad (18.155)$$

This is called **sparse GP regression** or **SGPR** [Tit09].

With these parameters, the lower bound on the log marginal likelihood is given by

$$\log p(\mathbf{y}) \geq \log \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} + \beta^{-1} \mathbf{I}) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.156)$$

where $\mathbf{Q}_{X,X} = \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}$. (This is called the “collapsed” lower bound, since we have marginalized out \mathbf{f}_Z .) If $Z = X$, then $\mathbf{K}_{Z,Z} = \mathbf{K}_{Z,X} = \mathbf{K}_{X,X}$, so the bound becomes tight, and we have $\log p(\mathbf{y}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_{X,X} + \beta^{-1}\mathbf{I})$.

Equation (18.156) is almost the same as the log marginal likelihood for the DTC model in Equation (18.132), except for the trace term; it is this latter term that prevents overfitting, due to the fact that we treat \mathbf{f}_Z as variational parameters of the posterior rather than model parameters of the prior.

18.5.4.2 Non-Gaussian likelihood

In this section, we briefly consider the case of non-Gaussian likelihoods, which arise when using GPs for classification or for count data (see Section 18.4). We can compute the gradients of the expected log likelihood by defining $h(f_n) = \log p(y_n|f_n)$ and then using a Monte Carlo approximation to Equation (18.149) and Equation (18.150). In the case of a binary classifier, we can use the results in Table 18.1 to compute the inner $\frac{\partial}{\partial f_n} h(f_n)$ and $\frac{\partial^2}{\partial f_n^2} h(f_n)$ terms.

18.5.4.3 Minibatch SVI

Computing the optimal variational solution in Section 18.5.4.1 requires solving a batch optimization problem, which takes $O(M^3 + NM^2)$ time. This may still be too slow if N is large, unless M is small, which compromises accuracy.

An alternative approach is to perform stochastic optimization of the VFE objective, instead of batch optimization. This is known as stochastic variational inference (see Section 10.3.2). The key observation is that the log likelihood in Equation (18.145) is a sum of N terms, which we can approximate with minibatch sampling to compute noisy estimates of the gradient, as proposed in [HFL13].

In more detail, the objective becomes

$$\mathcal{L}(q) = \left[\frac{N}{B} \sum_{b=1}^B \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \mathbb{E}_{q(f_n)} [\log p(y_n|f_n)] \right] - D_{\text{KL}}(q(\mathbf{f}_Z) \parallel p(\mathbf{f}_Z)) \quad (18.157)$$

where \mathcal{B}_b is the b 'th batch, and B is the number of batches. Since the GP model (with Gaussian likelihoods) is in the exponential family, we can efficiently compute the natural gradient (Section 6.4) of Equation (18.157) wrt the canonical parameters of $q(\mathbf{f}_Z)$; this converges much faster than following the standard gradient. See [HFL13] for details.

18.5.5 Exploiting parallelization and structure via kernel matrix multiplies

It takes $O(N^3)$ time to compute the Cholesky decomposition of $\mathbf{K}_{X,X}$, which is needed to solve the linear system $\mathbf{K}_\sigma \boldsymbol{\alpha} = \mathbf{y}$ and to compute $|\mathbf{K}_{X,X}|$. An alternative to Cholesky decomposition is to use linear algebra methods, often called **Krylov subspace methods** based just on **matrix vector multiplication** or **MVM**. These approaches are often much faster.

In short, if the kernel matrix $\mathbf{K}_{X,X}$ has special algebraic structure, which is often the case through either the choice of kernel or the structure of the inputs, then it is typically easier to exploit this structure in performing fast matrix multiplies. Moreover, even if the kernel matrix **does not** have

special structure, matrix multiplies are trivial to parallelize, and can thus be greatly accelerated by GPUs, unlike Cholesky based methods which are largely sequential. Algorithms based on matrix multiplies are in harmony with modern hardware advances, which enable significant parallelization.

18.5.5.1 Using conjugate gradient and Lanczos methods

We can solve the linear system $\mathbf{K}_\sigma \boldsymbol{\alpha} = \mathbf{y}$ using conjugate gradients (CG). The key computational step in CG is the ability to perform MVMs. Let $\tau(\mathbf{K}_\sigma)$ be the time complexity of a single MVM with \mathbf{K}_σ . For a dense $n \times n$ matrix, we have $\tau(\mathbf{K}_\sigma) = n^2$; however, we can speed this up if \mathbf{K}_σ is sparse or structured, as we discuss below.

Even if \mathbf{K}_σ is dense, we may still be able to save time by solving the linear system approximately. In particular, if we perform C iterations, CG will take $O(C\tau(\mathbf{K}_\sigma))$ time. If we run for $C = n$, and $\tau(\mathbf{K}_\sigma) = n^2$, it gives the exact solution in $O(n^3)$ time. However, often we can use fewer iterations and still get good accuracy, depending on the condition number of \mathbf{K}_σ .

We can compute the log determinant of a matrix using the MVM primitive with a similar iterative method known as **stochastic Lanczos quadrature** [UCS17; Don+17a]. This takes $O(L\tau(\mathbf{K}_\sigma))$ time for L iterations.

These methods have been used in the **blackbox matrix-matrix multiplication (BBMM)** inference procedure of [Gar+18a], which formulates a batch approach to CG that can be effectively parallelized on GPUs. Using 8 GPUs, this enabled the authors of [Wan+19a] to perform exact inference for a GP regression model on $N \sim 10^4$ datapoints in seconds, $N \sim 10^5$ datapoints in minutes, and $N \sim 10^6$ datapoints in hours.

Interestingly, Figure 18.14 shows that exact GP inference on a subset of the data can often outperform approximate inference on the full data. We also see that performance of exact GPs continues to significantly improve as we increase the size of the data, suggesting that GPs are not only useful in the small-sample setting. In particular, the BBMM is an exact method, and so will preserve the non-parametric representation of a GP with a non-degenerate kernel. By contrast, standard scalable approximations typically operate by replacing the exact kernel with an approximation that corresponds to a parametric model. The non-parametric GPs are able to grow their capacity with more data, benefiting more significantly from the structure present in large datasets.

18.5.5.2 Kernels with compact support

Suppose we use a kernel with **compact support**, where $\mathcal{K}(\mathbf{x}, \mathbf{x}') = 0$ if $\|\mathbf{x} - \mathbf{x}'\| > \epsilon$ for some threshold ϵ (see e.g., [MR09]), then \mathbf{K}_σ will be sparse, so $\tau(\mathbf{K}_\sigma)$ will be $O(N)$. We can also induce sparsity and structure in other ways, as we discuss in Section 18.5.5.3.

18.5.5.3 KISS

One way to ensure that MVMs are fast is to force the kernel matrix to have structure. The **structured kernel interpolation (SKI)** method of [WN15] does this as follows. First it assumes we have a set of inducing points, with Gram matrix $\mathbf{K}_{Z,Z}$. It then interpolates these values to predict the entries of the full kernel matrix using

$$\mathbf{K}_{X,X} \approx \mathbf{W}_X \mathbf{K}_{Z,Z} \mathbf{W}_X^\top \quad (18.158)$$

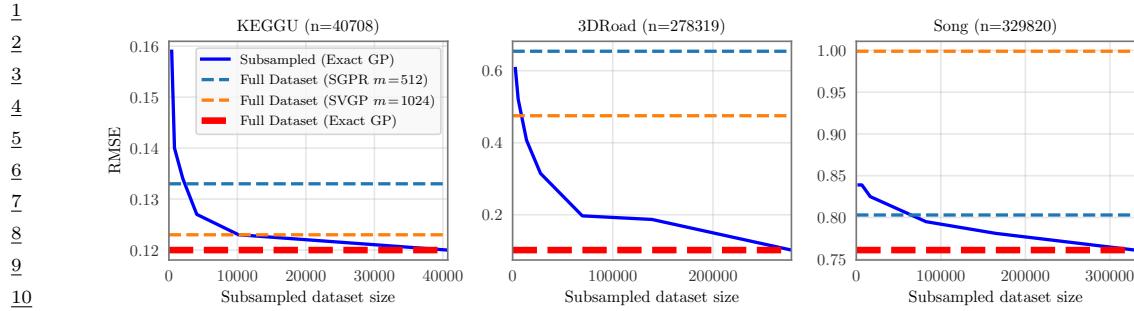


Figure 18.14: RMSE on test set as a function of training set size using a GP with Matern 3/2 kernel with shared lengthscale across all dimensions. Solid lines: exact inference. Dashed blue: SGPR method (closed-form batch solution to the Gaussian variational approximation) of Section 18.5.4.1 with $M = 512$ inducing points. Dashed orange: SVGP method (SGD on Gaussian variational approximation) of Section 18.5.4.3 with $M = 1024$ inducing points. Number of input dimensions: KEGGU $D = 27$, 3DRoad $D = 3$, Song $D = 90$. From Figure 4 of [Wan+19a]. Used with kind permission of Andrew Wilson.

where \mathbf{W}_X is a sparse matrix containing interpolation weights. If we use cubic interpolation, each row only has 4 nonzeros. Thus we can compute $(\mathbf{W}_X \mathbf{K}_{Z,Z} \mathbf{W}_X^\top) \mathbf{v}$ for any vector \mathbf{v} in $O(N + M^2)$ time.

Note that the **SKI** approach generalizes all inducing point methods. For example, we can recover the subset of regressors method (SOR) method by setting the interpolation weights to $\mathbf{W} = \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1}$. We can identify this procedure as performing a global Gaussian process interpolation strategy on the user specified kernel. See [WN15] and [WDN15] for more details.

In 1d, we can further reduce the running time by choosing the inducing points to be on a regular grid, so that $\mathbf{K}_{Z,Z}$ is a Toeplitz matrix. In higher dimensions, we need to use a multidimensional grid of points, resulting in $\mathbf{K}_{Z,Z}$ being a Kronecker product of Toeplitz matrices. This enables matrix vector multiplication in $O(N + M \log M)$ time and $O(N + M)$ space. The resulting method is called **KISS-GP** [WN15], which stands for ‘kernel interpolation for scalable, structured GPs’.

Unfortunately, the KISS method can take exponential time in the input dimensions D when exploiting Kronecker structure in $\mathbf{K}_{Z,Z}$, due to the need to create a fully connected multidimensional lattice. In [Gar+18b], they propose a method called **SKIP**, which stands for ‘SKI for products’. The idea is to leverage the fact that many kernels (including ARD) can be written as a product of 1d kernels: $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \prod_{d=1}^D \mathcal{K}^d(\mathbf{x}, \mathbf{x}')$. This can be combined with the 1d SKI method to enable fast MVMs. The overall running time to compute the log marginal likelihood (which is the bottleneck for kernel learning) using C iterations of CG and a Lanczos decomposition of rank L , becomes $O(DL(N + M \log M) + L^3 N \log D + CL^2 N)$. Typical values are $L \sim 10^1$ and $C \sim 10^2$.

40

41 18.5.5.4 Tensor train methods

42 Consider the Gaussian VFE approach in Section 18.5.4. We have to estimate the covariance \mathbf{S} and
43 the mean \mathbf{m} . We can represent \mathbf{S} efficiently using Kronecker structure, as used by KISS. Additionally,
44 we can represent \mathbf{m} efficiently using the **tensor train decomposition** [Ose11] in combination with
45 **SKI** [WN15]. The resulting **TT-GP** method can scale efficiently to billions of inducing points, as
46

47

1 explained in [INK18].
2

3 18.5.6 Converting a GP to a SSM

4 Consider a function defined on a 1d scalar input, such as a time index. For many kernels, the
5 corresponding GP can be modeled using a stochastic differential equation. This induces a block
6 tri-diagonal precision matrix for the posterior $p(\mathbf{f}_{1:T}|\mathbf{y}_{1:T})$. We can therefore convert this to a
7 linear-Gaussian state space model (Section 31.1), and perform exact inference in $O(T)$ time using
8 Kalman smoothing, as explained in [SSH13; Ada+20]. This conversion can be done exactly for
9 Matern kernels and approximately for Gaussian (RBF) kernels (see [SS19, Ch. 12]). In [SGF21], they
10 describe how to reduce the linear dependence on T to $\log(T)$ time using a **parallel prefix scan**
11 operator, that can be run efficiently on GPUs.
12

13 18.6 Learning the kernel

14 In [Mac98], David MacKay asked: “How can Gaussian processes replace neural networks? Have we
15 thrown the baby out with the bathwater?” This remark was made in the late 1990s, at the end of
16 the second wave of neural networks. Researchers and practitioners had grown weary of the design
17 decisions associated with neural networks — such as activation functions, optimization procedures,
18 architecture design — and the lack of a principled framework to make these decisions. Gaussian
19 processes, by contrast, were perceived as flexible and principled probabilistic models, which naturally
20 followed from Radford Neal’s results on infinite neural networks [Nea96], which we discuss in more
21 depth in Section 18.7.

22 However, MacKay [Mac98] noted that neural networks could discover rich representations of data
23 through adaptive hidden basis functions, while Gaussian processes with standard kernel functions,
24 such as the RBF kernel, are essentially just smoothing devices. Indeed, the generalization properties
25 of Gaussian processes hinge on the suitability of the kernel function. *Learning* the kernel is how we
26 do **representation learning** with Gaussian processes, and in many cases will be crucial for good
27 performance — especially when we wish to perform **extrapolation**, making predictions far away
28 from the data [WA13; Wil+14].

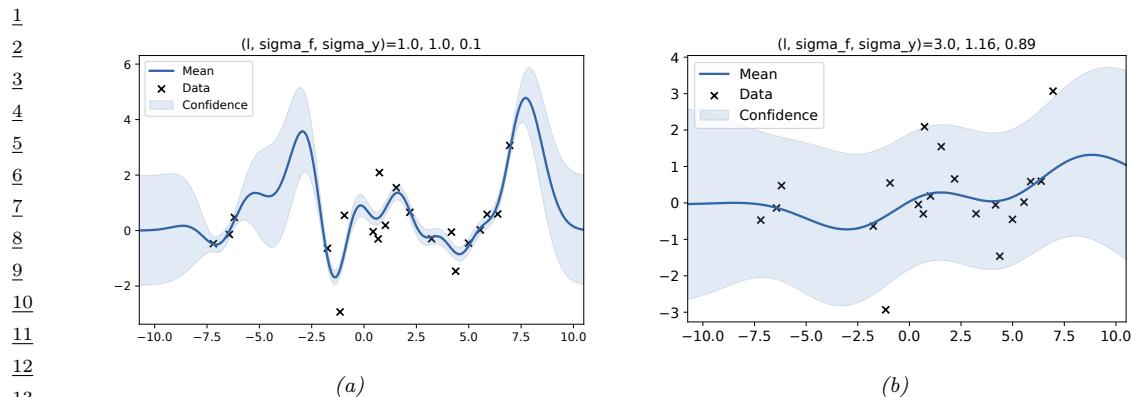
29 As we will see, learning a kernel is in many ways analogous to training a neural network. Moreover,
30 neural networks and Gaussian processes can be synergistically combined through approaches such as
31 deep kernel learning (see Section 18.6.6) and deep GPs (Section 18.7.3).

32 18.6.1 Empirical Bayes for the kernel parameters

33 Suppose, as in Section 18.3.2, we are performing 1d regression using a GP with an RBF kernel. Since
34 the data has observation noise, the kernel has the following form:
35

$$\mathcal{K}_y(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x_p - x_q)^2\right) + \sigma_y^2 \delta_{pq} \quad (18.159)$$

36 Here ℓ is the horizontal scale over which the function changes, σ_f^2 controls the vertical scale of
37 the function, and σ_y^2 is the noise variance. Figure 18.15 illustrates the effects of changing these
38 parameters. We sampled 20 noisy data points from the SE kernel using $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$,
39 and then made predictions various parameters, conditional on the data. In Figure 18.15(a), we use
40



¹⁴ Figure 18.15: Some 1d GPs with RBF kernels but different hyper-parameters fit to 20 noisy observations.
¹⁵ The hyper-parameters $(\ell, \sigma_f, \sigma_y)$ are as follows: (a) $(1, 1, 0.1)$ (b) $(3.0, 1.16, 0.89)$. Adapted from Figure 2.5
¹⁶ of [RW06]. Generated by `gprDemoChangeHparams.py`.

²¹ $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and the result is a good fit. In Figure 18.15(b), we increase the length scale ²² to $\ell = 3$; now the function looks smoother, but we are arguably underfitting.

²³ To estimate the kernel parameters θ (sometimes called hyperparameters), we could use exhaustive
²⁴ search over a discrete grid of values, with validation loss as an objective, but this can be quite slow.
²⁵ (This is the approach used by nonprobabilistic methods, such as SVMs, to tune kernels.) Here we
²⁶ consider an empirical Bayes approach, which will allow us to use continuous optimization methods,
²⁷ which are much faster. In particular, we will maximize the marginal likelihood

$$\frac{29}{30} \quad p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X})p(\mathbf{f}|\mathbf{X}, \boldsymbol{\theta})d\mathbf{f} \quad (18.160)$$

³² (The reason it is called the marginal likelihood, rather than just likelihood, is because we have marginalized out the latent Gaussian vector \mathbf{f}_* .) Since $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K})$, and $p(\mathbf{y}|\mathbf{f}) = \prod_{n=1}^N \mathcal{N}(y_n|f_n, \sigma_y^2)$, the marginal likelihood is given by

$$\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_\sigma) = -\frac{1}{2}\mathbf{y}\mathbf{K}_\sigma^{-1}\mathbf{y} - \frac{1}{2}\log |\mathbf{K}_\sigma| - \frac{N}{2}\log(2\pi) \quad (18.161)$$

where the dependence of \mathbf{K}_σ on θ is implicit. The first term is a data fit term, the second term is a model complexity term, and the third term is just a constant. To understand the tradeoff between the first two terms, consider a SE kernel in 1D, as we vary the length scale ℓ and hold σ_y^2 fixed. Let $J(\ell) = -\log p(\mathbf{y}|\mathbf{X}, \ell)$. For short length scales, the fit will be good, so $\mathbf{y}^\top \mathbf{K}_\sigma^{-1} \mathbf{y}$ will be small. However, the model complexity will be high: \mathbf{K} will be almost diagonal, since most points will not be considered “near” any others, so the $\log |\mathbf{K}_\sigma|$ will be large. For long length scales, the fit will be poor but the model complexity will be low: \mathbf{K} will be almost all 1’s, so $\log |\mathbf{K}_\sigma|$ will be small.

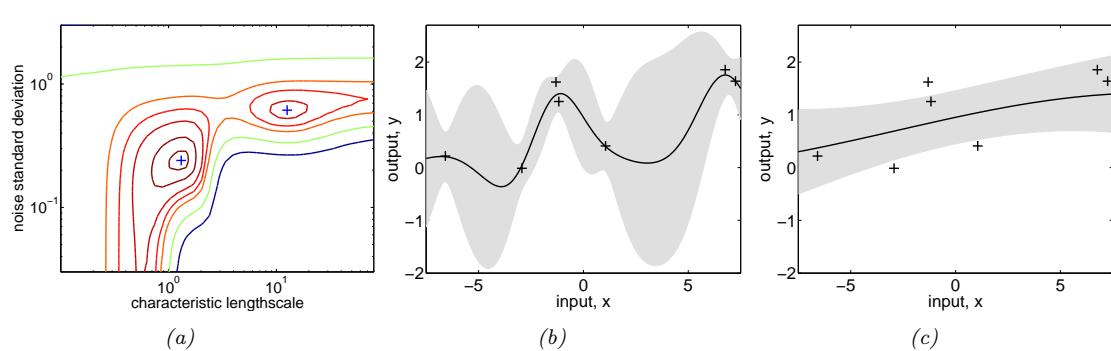


Figure 18.16: Illustration of local minima in the marginal likelihood surface. (a) We plot the log marginal likelihood vs σ_y^2 and ℓ , for fixed $\sigma_f^2 = 1$, using the 7 data points shown in panels b and c. (b) The function corresponding to the lower left local minimum, $(\ell, \sigma_n^2) \approx (1, 0.2)$. This is quite “wiggly” and has low noise. (c) The function corresponding to the top right local minimum, $(\ell, \sigma_n^2) \approx (10, 0.8)$. This is quite smooth and has high noise. The data was generated using $(\ell, \sigma_n^2) = (1, 0.1)$. Adapted from Figure 5.5 of [RW06]. Generated by `qpr demo marqlik.py`.

We now discuss how to maximize the marginal likelihood. One can show that

$$\frac{\partial}{\partial \theta_i} \log p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta}) = \frac{1}{2} \mathbf{y}^\top \mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_i} \mathbf{K}_\sigma^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_i}) \quad (18.162)$$

$$= \frac{1}{2} \text{tr} \left((\boldsymbol{\alpha}\boldsymbol{\alpha}^\top - \mathbf{K}_\sigma^{-1}) \frac{\partial \mathbf{K}_\sigma}{\partial \theta_i} \right) \quad (18.163)$$

where $\alpha = \mathbf{K}_\sigma^{-1} \mathbf{y}$. It takes $O(N^3)$ time to compute \mathbf{K}_σ^{-1} , and then $O(N^2)$ time per hyper-parameter to compute the gradient.

The form of $\frac{\partial \mathbf{K}_\sigma}{\partial \theta_j}$ depends on the form of the kernel, and which parameter we are taking derivatives with respect to. Often we have constraints on the hyper-parameters, such as $\sigma_y^2 \geq 0$. In this case, we can define $\theta = \log(\sigma_y^2)$, and then use the chain rule.

Given an expression for the log marginal likelihood and its derivative, we can estimate the kernel parameters using any standard gradient-based optimizer. However, since the objective is not convex, local minima can be a problem, as we illustrate below, so we may need to use multiple restarts.

18.6.1.1 Example

Consider Figure 18.16. We use the SE kernel in Equation (18.159) with $\sigma_f^2 = 1$, and plot $\log p(\mathbf{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where \mathbf{X} and \mathbf{y} are the 7 data points shown in panels b and c as we vary ℓ and σ_y^2). The two local optima are indicated by + in panel (a). The bottom left optimum corresponds to a low-noise, short-length scale solution (shown in panel b). The top right optimum corresponds to a high-noise, long-length scale solution (shown in panel c). With only 7 data points, there is not enough evidence to confidently decide which is more reasonable, although the more complex model (panel b) has a marginal likelihood that is about 60% higher than the simpler model (panel c). With more data, the more complex model would become even more preferred.

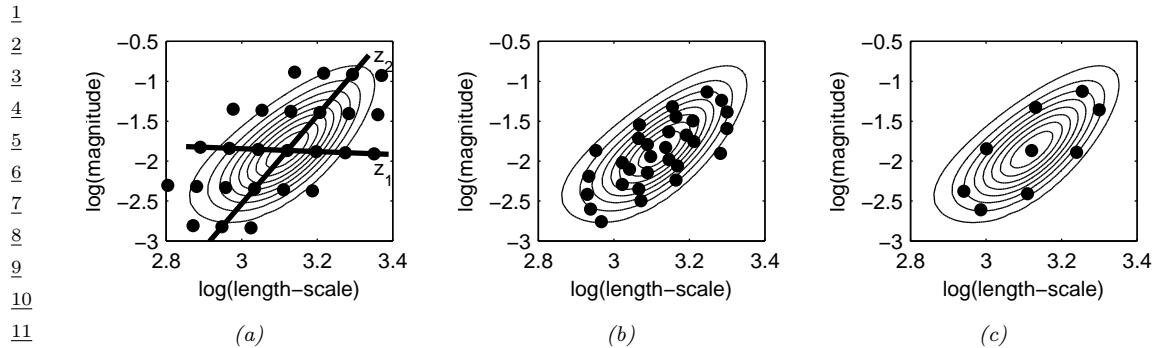


Figure 18.17: Three different approximations to the posterior over hyper-parameters: grid-based, Monte Carlo, and central composite design. From Figure 3.2 of [Van10]. Used with kind permission of Jarno Vanhatalo.

Figure 18.16 illustrates some other interesting (and typical) features. The region where $\sigma_y^2 \approx 1$ (top of panel a) corresponds to the case where the noise is very high; in this regime, the marginal likelihood is insensitive to the length scale (indicated by the horizontal contours), since all the data is explained as noise. The region where $\ell \approx 0.5$ (left hand side of panel a) corresponds to the case where the length scale is very short; in this regime, the marginal likelihood is insensitive to the noise level (indicated by the vertical contours), since the data is perfectly interpolated. Neither of these regions would be chosen by a good optimizer.

18.6.2 Bayesian inference for the kernel parameters

When we have a small number of datapoints (e.g., when using GPs for blackbox optimization, as we discuss in Section 6.9), using a point estimate of the kernel parameters can give poor results [Bul11; WF14]. As a simple example, if the function values that have been observed so far are all very similar, then we may estimate $\hat{\sigma} \approx 0$, which will result in overly confident predictions.³

To overcome such overconfidence, we can compute a posterior over the kernel parameters. If the dimensionality of $\boldsymbol{\theta}$ is small, we can compute a discrete grid of possible values, centered on the MAP estimate $\hat{\boldsymbol{\theta}}$ (computed as above). We can then approximate the posterior using

$$p(\mathbf{f}|\mathcal{D}) = \sum_{s=1}^S p(\mathbf{f}|\mathcal{D}, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s|\mathcal{D}) w_s \quad (18.164)$$

where w_s denotes the weight for grid point s .

In higher dimensions, a regular grid suffers from the curse of dimensionality. One alternative is to place grid points at the mode, and at a distance $\pm 1\text{sd}$ from the mode along each dimension, for a total of $2|\boldsymbol{\theta}| + 1$ points. This is called a **central composite design** [RMC09]. See Figure 18.17 for an illustration.

In higher dimensions, we can use Monte Carlo inference for the kernel parameters when computing

³ In [WSN00; BBV11b], they show how we can put a conjugate prior on σ^2 and integrate it out, to generate a Student version of the GP, which is more robust.

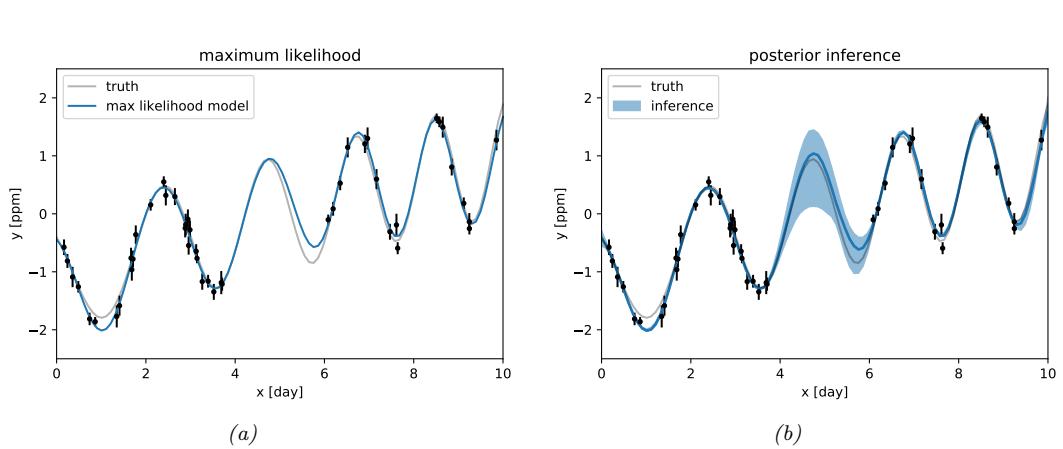


Figure 18.18: Difference between estimation and inference for kernel hyper-parameters. (a) Empirical Bayes approach based on optimization. We plot the posterior predicted mean given a plug-in estimate, $\mathbb{E}[f(x)|\mathcal{D}, \hat{\theta}]$. (b) Bayesian approach based on HMC. We plot the posterior predicted mean, marginalizing over hyper-parameters, $\mathbb{E}[f(x)|\mathcal{D}]$. Generated by `gp_kernel_opt.ipynb`.

Equation (18.164). For example, [MA10] shows how to use slice sampling (Section 12.4.1) for this task, [Hen+15] shows how to use HMC (Section 12.5), and [BBV11a] shows how to use SMC (Chapter 13).

In Figure 18.18, we illustrate the difference between kernel optimization vs kernel inference. We fit a 1d dataset using a kernel of the form

$$\mathcal{K}(r) = \sigma_2^2 \mathcal{K}_{\text{SE}}(r; \tau) \mathcal{K}_{\text{cos}}(r; \rho_1) + \sigma_3^2 \mathcal{K}_{\text{3D}}(r; \rho_2) \quad (18.165)$$

where $\mathcal{K}_{\text{SE}}(r; \ell)$ is the squared exponential kernel (Equation (18.11)), $\mathcal{K}_{\cos}(r; \rho_1)$ is the cosine kernel (Equation (18.18)), and $\mathcal{K}_{32}(r; \rho_2)$ is the Matern $\frac{3}{2}$ kernel (Equation (18.14)). We then compute a point-estimate of the kernel parameters using empirical Bayes, and posterior samples using HMC. We then predicting the posterior mean of f on a 1d test set by plugging in the MLE or averaging over samples. We see that the latter captures more uncertainty (beyond the uncertainty captured by the Gaussian itself).

18.6.3 Multiple kernel learning for additive kernels

A special case of kernel learning arises when the kernel is a sum of B base kernels

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{x}, \mathbf{x}') \quad (18.166)$$

Optimizing the weights $w_b > 0$ using structural risk minimization is known as **multiple kernel learning**; see e.g. [Bak+08] for details.

Now suppose we constrain the base kernels to depend on a subset of the variables. Furthermore, suppose we enforce a hierarchical inclusion property (e.g., including the kernel k_{123} means we must

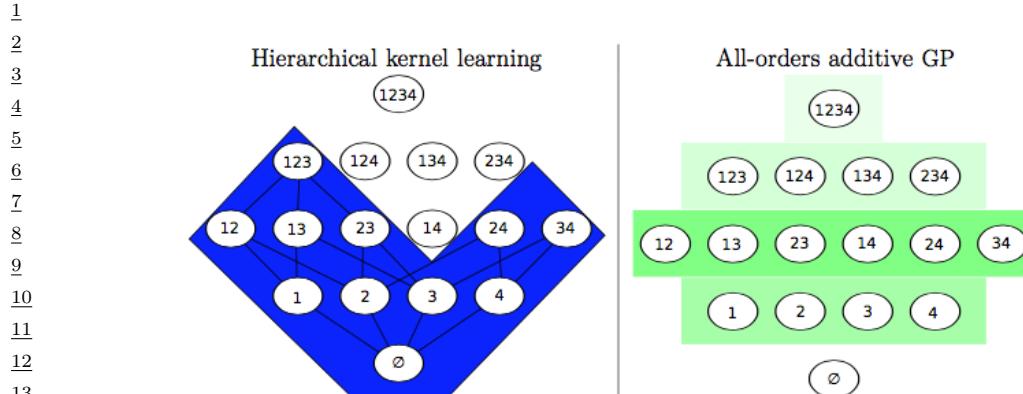


Figure 18.19: Comparison of different additive model classes for a 4d function. Circles represent different interaction terms, ranging from first-order to fourth-order. Color shades represent different weighting terms. Left: hierarchical kernel learning uses a nested hierarchy of terms. Right: additive GPs use a weighted sum of additive kernels of different orders. From Figure 6.2 of [Duv14]. Used with kind permission of David Duvenaud.

also include k_{12} , k_{13} and k_{23}), as illustrated in Figure 18.19(left). This is called **hierarchical kernel learning**. We can find a good subset from this model class using convex optimization [Bac09]; however, this requires the use of cross validation to estimate the weights. A more efficient approach is to use the empirical Bayes approach described in [DNR11].

In many cases, it is common to restrict attention to first order additive kernels, i.e., $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^D \mathcal{K}_d(x_d, x'_d)$. The resulting function has the form

$$f(\mathbf{x}) = f_1(x_1) + \dots + f_D(x_D) \quad (18.167)$$

This is called a **generalized additive model** or **GAM**.

Figure 18.20 shows an example of this, where each base kernel has the form $\mathcal{K}_d(x_d, x'_d) = \sigma_d^2 \text{SE}(x_d, x'_d | \ell_d)$. In Figure 18.20, we see that the σ_d^2 terms for the coarse and fine features are set to zero, indicating that these inputs have no impact on the response variable.

[DBW20] considers additive kernels operating on different linear projections of the inputs:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{P}_b \mathbf{x}, \mathbf{P}_b \mathbf{x}') \quad (18.168)$$

Surprisingly, they show that these models can match or exceed the performance of kernels operating on the original space, even when the projections are into a **single** dimension, and not learned. In other words, it is possible to reduce many regression problems to a single dimension without loss in performance. This finding is particularly promising for scalable inference, such as KISS (see Section 18.5.5.3), and active learning, which are greatly simplified in a low dimensional setting.

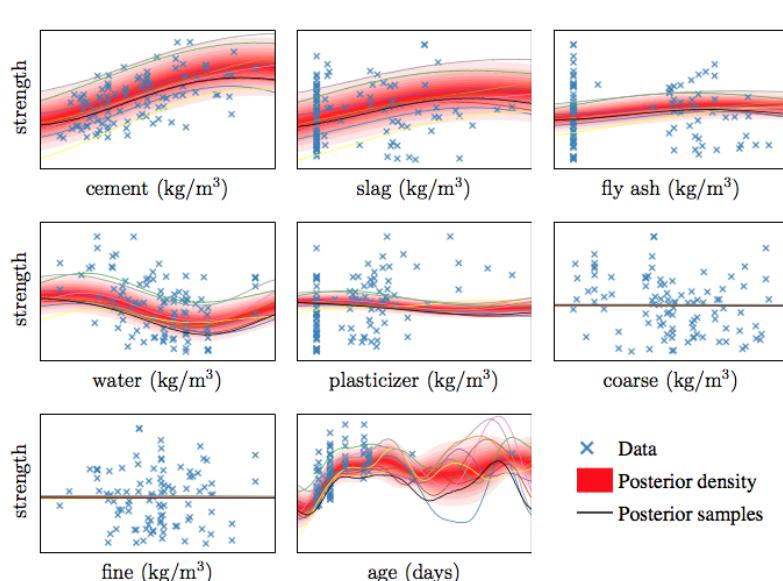


Figure 18.20: Predictive distribution of each term in a GP-GAM model applied to a dataset with 8 continuous inputs and 1 continuous output, representing the strength of some concrete. From Figure 2.7 of [Duv14]. Used with kind permission of David Duvenaud.

18.6.4 Automatic search for compositional kernels

Although the above methods can estimate the hyperparameters of a specified set of kernels, they do not choose the kernels themselves (other than the special case of selecting a subset of kernels from a set). In this section, we describe a method, based on [Duv+13], for sequentially searching through the space of increasingly complex GP models so as to find a parsimonious description of the data.

We start with a simple kernel, such as the white noise kernel, and then consider replacing it with a set of possible alternative kernels, such as an SE kernel, RQ kernel, etc. We use the BIC score (Section 3.7.5.2) to evaluate each candidate model (choice of kernel) m . This has the form $BIC(m) = \log p(\mathcal{D}|m) - \frac{1}{2}|m|\log N$, where $p(\mathcal{D}|m)$ is the marginal likelihood, and $|m|$ is the number of parameters. The first term measures fit to the data, and the second term is a complexity penalty. We can also consider replacing a kernel by the addition of two kernels, $k \rightarrow (k + k')$, or the multiplication of two kernels, $k \rightarrow (k \times k')$. See Figure 18.21 for an illustration of the search space.

Searching through this space is similar to what a human expert would do. In particular, if we find structure in the residuals, such as periodicity, we can propose a certain “move” through the space. We can also start with some structure that is assumed to hold globally, such as linearity, but if we find this only holds locally, we can multiply the kernel by an SE kernel. We can also add input dimensions incrementally, to capture higher order interactions.

Figure 18.22 shows the output of this process applied to a dataset of monthly totals of international airline passengers. The input to the GP is the set of time stamps, $\mathbf{x} = 1 : t$; there are no other features.

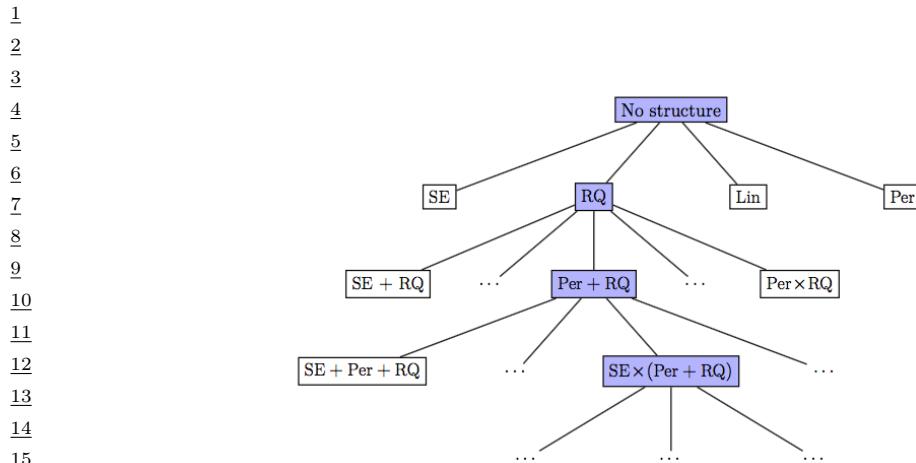


Figure 18.21: Example of a search tree over kernel expressions. From Figure 3.2 of [Duv14]. Used with kind permission of David Duvenaud.

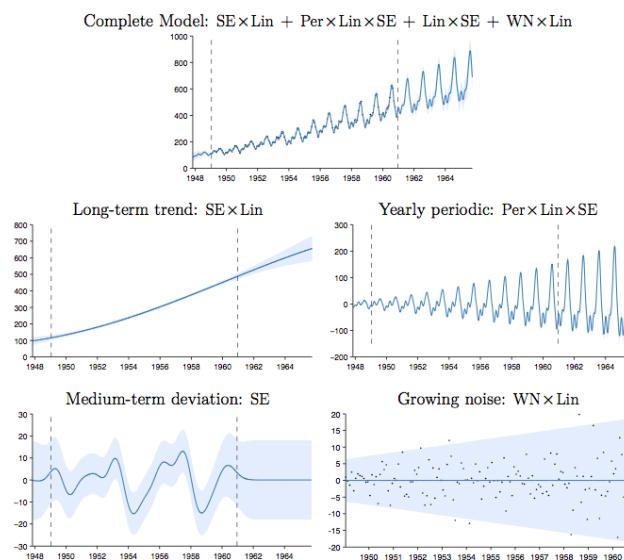


Figure 18.22: Top row: airline dataset and posterior distribution of the model discovered after a search of depth 10. Subsequent rows: predictions of the individual components. From Figure 3.5 of [Duv14], based on [Llo+14]. Used with kind permission of David Duvenaud.

The observed data lies in between the dotted vertical lines; curves outside of this region are extrapolations. We see that the system has discovered a fairly interpretable set of patterns in the data. Indeed, it is possible to devise an algorithm to automatically convert the output of this search process to a natural language summary, as shown in [Llo+14]. In this example, it summarizes the data as being generated by the addition of 4 underlying trends: a linearly increasing function; an approximately periodic function with a period of 1.0 years, and with linearly increasing amplitude; a smooth function; and uncorrelated noise with linearly increasing standard deviation.

Recently, [Sun+18] showed how to create a DNN which learns the kernel given two input vectors. The hidden units are defined as sums and products of elementary kernels, as in the above search based approach. However, the DNN can be trained in a differentiable way, so is much faster.

18.6.5 Spectral mixture kernel learning

Any shift-invariant (stationary) kernel can be converted via the Fourier transform to its dual form, known as its **spectral density**. This means that learning the spectral density is equivalent to learning any shift-invariant kernel. For example, if we take the Fourier transform of an RBF kernel, we get a Gaussian spectral density centered at the origin. If we take the Fourier transform of a Matern kernel, we get a Student- t spectral density centred at the origin. Thus standard approaches to multiple kernel learning, which typically involve additive compositions of RBF and Matern kernels with different length-scale parameters, amount to density estimation with a scale mixture of Gaussian or Student- t distributions at the origin. Such models are very inflexible for density estimation, and thus also very limited in being able to perform kernel learning.

On the other hand, *scale-location* mixture of Gaussians can model any density to arbitrary precision. Moreover, with even a small number of components these mixtures of Gaussians are highly flexible. Thus a spectral density corresponding to a scale-location mixture of Gaussians forms an expressive basis for all shift-invariant kernels. One can evaluate the inverse Fourier transform for a Gaussian mixture analytically, to derive the **spectral mixture kernel** [WA13], which we can express for one-dimensional inputs x as:

$$\mathcal{K}(x, x') = \sum_i w_i \cos((x - x')(2\pi\mu_i)) \exp(-2\pi^2(x - x')^2 v_i) \quad (18.169)$$

The mixture weights w_i , as well as the means μ_i and variances v_i of the Gaussians in the spectral density, can be learned by empirical Bayes optimization (Section 18.6.1) or in a fully-Bayesian procedure (Section 18.6.2) [Jan+17]. We illustrate the former approach in Figure 18.23.

By learning the parameters of the spectral mixture kernel, we can discover representations that enable extrapolation — to make reasonable predictions far away from the data. For example, in Section 19.3.3.1, compositions of kernels are carefully hand-crafted to extrapolate CO₂ concentrations. But in this instance, the human statistician is doing all of the interesting representation learning. Figure Figure 18.24 shows Gaussian processes with learned spectral mixture kernels instead automatically extrapolating on CO₂ and airline passenger problems.

These kernels can also be used to extrapolate higher dimensional large-scale spatio-temporal patterns. Large datasets can provide relatively more information for expressive kernel learning. However, scaling an expressive kernel learning approach poses different challenges than scaling a standard Gaussian process model. One faces additional computational constraints, and the need to retain significant model structure for expressing the rich information available in a large dataset.

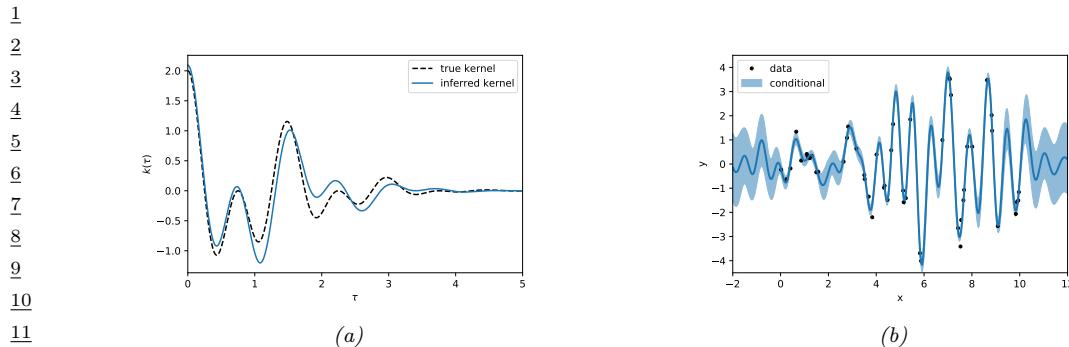


Figure 18.23: Illustration of a GP with a spectral mixture kernel in 1d. (a) Learned vs true kernel. (b) Predictions using learned kernel. Generated by [gp_spectral_mixture.ipynb](#).

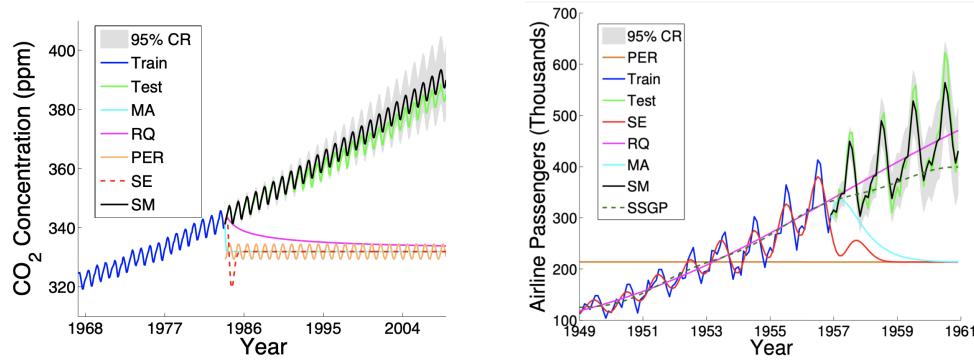


Figure 18.24: Extrapolations (point predictions and 95% credible set) on CO₂ and airline datasets using Gaussian processes with Matern, rational quadratic, periodic, RBF (SE), and spectral mixture kernels, each with hyperparameters learned using empirical Bayes. From Figure from [Wil14].

Indeed, in Figure 18.24 we can separately understand the effects of the kernel learning approach and scalable inference procedure, in being able to discover structure necessary to extrapolate textures. An expressive kernel model and a scalable inference approach that preserves a *non-parametric* representation are needed for good performance.

Structure exploiting inference procedures, such as Kronecker methods, as well as KISS-GP and conjugate gradient based approaches, are appropriate for these tasks — since they generally preserve or exploit existing structure, rather than introducing approximations that corrupt the structure. Spectral mixture kernels combined with these scalable inference techniques have been used to great effect for spatiotemporal extrapolation problems, including land-surface temperature forecasting, epidemiological modeling, and policy-relevant applications.

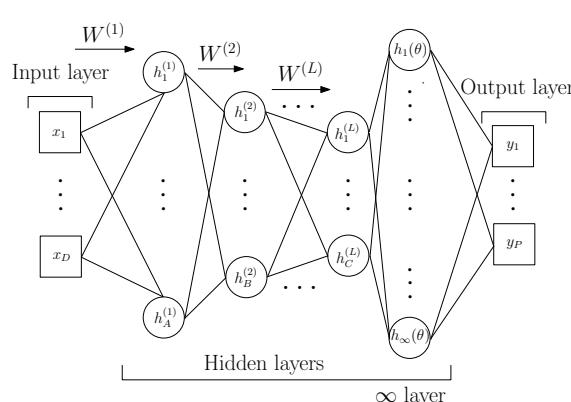


Figure 18.25: Deep Kernel Learning: A Gaussian process with a deep kernel maps D dimensional inputs \mathbf{x} through L parametric hidden layers followed by a hidden layer with an infinite number of basis functions, with base kernel hyperparameters $\boldsymbol{\theta}$. Overall, a Gaussian process with a deep kernel produces a probabilistic mapping with an infinite number of adaptive basis functions parametrized by $\gamma = \{\mathbf{w}, \boldsymbol{\theta}\}$. All parameters γ are learned through the marginal likelihood of the Gaussian process. From Figure 1 of [Wil+16b].

18.6.6 Deep kernel learning

Deep kernel learning [SH07; Wil+16b] combines the structural properties of neural networks with the non-parametric flexibility and uncertainty representation provided by Gaussian processes. For example, we can define a “deep RBF kernel” as follows:

$$\mathcal{K}_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}') = \exp \left[-\frac{1}{2\sigma^2} \|\mathbf{h}_{\boldsymbol{\theta}}^L(\mathbf{x}) - \mathbf{h}_{\boldsymbol{\theta}}^L(\mathbf{x}')\|^2 \right] \quad (18.170)$$

where $\mathbf{h}_{\boldsymbol{\theta}}^L(\mathbf{x})$ are the outputs of layer L from a DNN. We can then learn the parameters $\boldsymbol{\theta}$ by maximizing the marginal likelihood of the Gaussian processes.

This framework is illustrated in Figure 18.25. We can understand the neural network features as inputs into a base kernel. The neural network can either be (i) pre-trained, (ii) learned jointly with the base kernel parameters, or (iii) pre-trained and then fine-tuned through the marginal likelihood. This approach can be viewed as a “last-layer” Bayesian model, where a Gaussian process is applied to the final layer of a neural network. The base kernel often provides a good measure of distance in feature space, desirably encouraging predictions to have high uncertainty as we move far away from the data.

We can use deep kernel learning to help the GP learn discontinuous functions, as illustrated in Figure 18.26. On the left we show the results of a GP with a standard Matern $\frac{3}{2}$ kernel. It is clear that the out-of-sample predictions are poor. On the right we show the results of the same model where we first transform the input through a learned 2 layer MLP (with 15 and 10 hidden units). It is clear that the model is working much better.

As a more complex example, we consider a regression problem where we wish to map faces (vectors of pixel intensities) to a continuous valued orientation angle. In Figure 18.27, we evaluate the deep kernel matrix (with RBF and spectral mixture base kernels, discussed in Section 18.6.5) on data ordered by orientation angle. We can see that the learned deep kernels, in the left two panels, have a

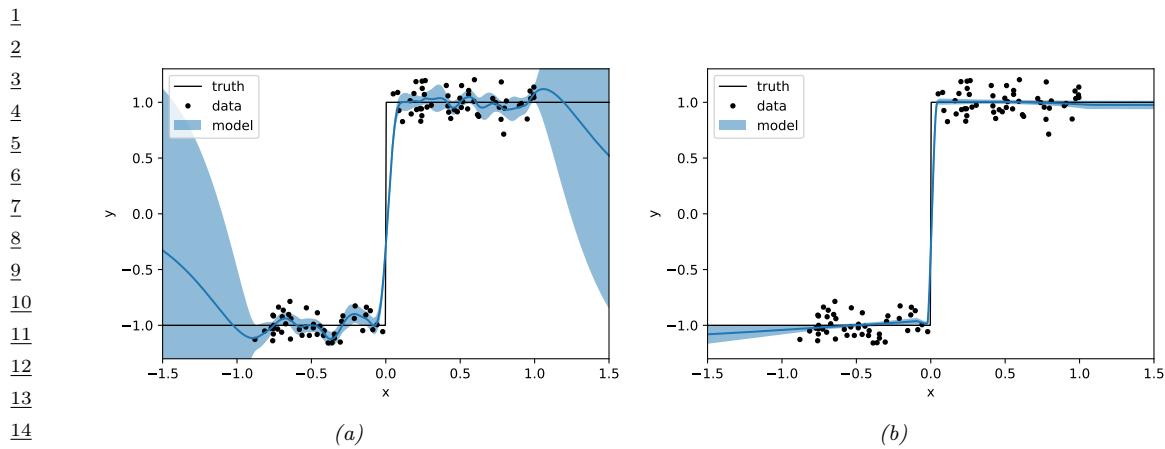


Figure 18.26: Modeling a discontinuous function with (a) a GP with a “shallow” Matern $\frac{3}{2}$ kernel, and (b) a GP with a “deep” MLP + Matern kernel. Generated by [gp_deep_kernel_learning.py](#).

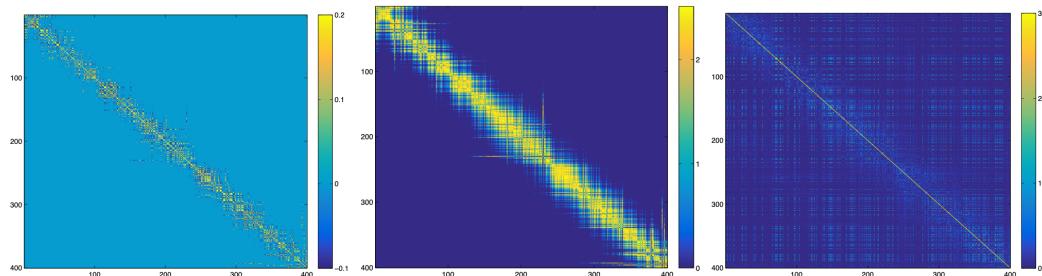


Figure 18.27: **Left:** The learned covariance matrix of a deep kernel with spectral mixture base kernel on a set of test cases for the Olivetti faces **dataset**, where the test samples are ordered according to the orientations of the input faces. **Middle:** The respective covariance matrix using a deep kernel with RBF base kernel. **Right:** The respective covariance matrix using a standard RBF kernel. From Figure 5 of [Wil+16b].

pronounced diagonal band, meaning that they have *discovered* that faces with similar orientation angles are correlated. On the other hand, in the right panel we see that the entries even for a learned RBF kernel are highly diffuse. Since the RBF kernel essentially uses Euclidean distance as a metric for similarity, it is unable to learn a representation that effectively solves this problem. In this case, one must do highly non-Euclidean metric learning.

40

18.6.7 Functional kernel learning

Gaussian processes naturally give rise to a function-space view of modelling, whereby we place a prior distribution directly over functions that could model our data. Since the kernel crucially influences the generalization properties of the Gaussian process, it is natural to take this perspective one step further in a hierarchical model — so that we can represent the prior belief that the kernel does not

47

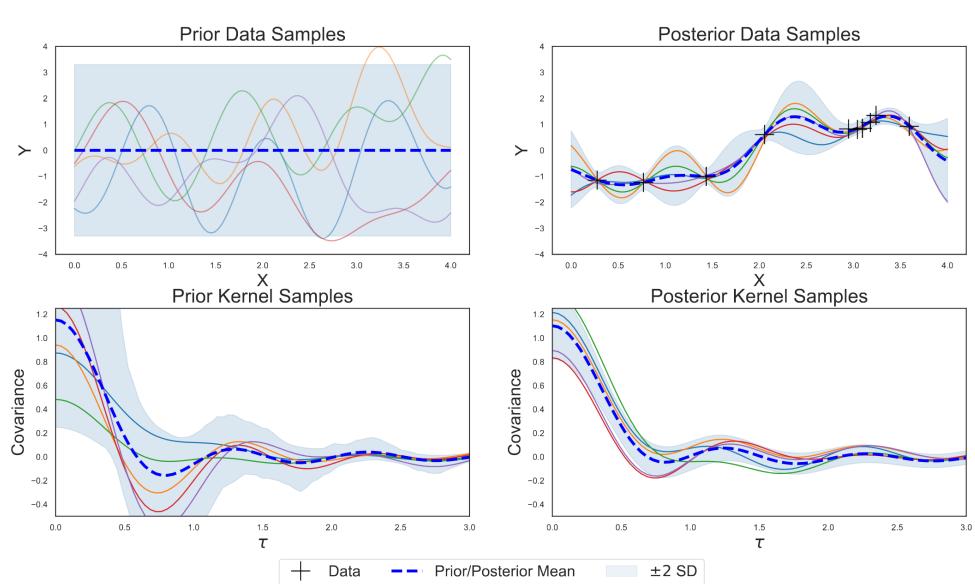


Figure 18.28: In the above two panels we show the conventional function-space approach to modeling data with GPs, with prior and posterior draws. In the bottom two rows we apply this perspective to kernels, with priors and posterior draws from a distribution over kernels, in a functional kernel learning model. From Figure 1 of [Ben+19].

have a simple functional form and that we have uncertainty over its values. This approach is referred to as **functional kernel learning** (FKL) [Ben+19].

FKL provides a function-space distribution over kernels by modeling a spectral density with a transformed Gaussian process. It is then possible to specify the prior expected kernel to be whatever one would have used otherwise, such as an RBF kernel, while enabling a non-parametric relaxation around its values. We can now imagine drawing prior kernel functions, observing data, and inferring posterior kernels. When we wish to form predictions, we can then marginalize this posterior over kernel functions.

This approach is shown in [Ben+19] to provide strong extrapolation performance on a number of spatiotemporal problems, without requiring much manual intervention.

18.7 GPs and DNNs

In this section, we discuss various connections between Gaussian processes (GPs) and deep neural networks (DNNs). Note that this is a rapidly changing field, so we just present a few highlights, rather than going into depth.

1 **18.7.1 Kernels derived from random DNNs (NN-GP)**

2

3

4 In Section 17.2.1, we showed that the prior over parameters of the DNN indirectly induces a prior
5 over functions. If we use infinitely wide MLPs with randomly initialized weights (drawn from a
6 Gaussian), the resulting distribution over functions is the same as a GP with a certain kernel. This
7 result was first shown for one layer MLPs in [Nea96; Wil98], and was later extended to deep MLPs
8 in [DFS16; Lee+18]. The resulting kernel is called the **NNGP** kernel [Lee+18] (also called the
9 **compositional kernel** [DFS16]).

10 A similar result holds for CNNs, where we let the number of channels per layer go to infinity, as
11 shown in [Nov+19]. In fact, one can show this phenomenon for a large class of neural networks, such
12 as RNNs and models with attention [Yan19]. Below we derive the result for the MLP case.

13

14

15

16

17

18

19

20

21 **18.7.1.1 Result for MLP with one hidden layer**

22

23 Let us start by considering an MLP with a single hidden layer with inputs $\mathbf{x} \in \mathbb{R}^{D_0}$ and outputs
24 $f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{z}^2 \in \mathbb{R}^{D_2}$, defined as follows:

25

26

27

28

29

$$\text{z}_i^2(\mathbf{x}) = b_i^2 + \sum_{j=1}^{D_1} W_{ij}^2 h_j^1(\mathbf{x}), \quad h_j^1(\mathbf{x}) = \varphi \left(b_j^1 + \sum_{k=1}^{D_0} W_{jk}^1 x_k \right) \quad (18.171)$$

30

31

32

33

34

35 We call z_i^ℓ the pre-activation for unit i at layer ℓ , and h_i^ℓ the post-activation.

36 We assume $W_{ij}^\ell \sim \mathcal{N}(0, \sigma_w^2/D_\ell)$ and $b_i^\ell \sim \mathcal{N}(0, \sigma_b^2)$. Because the weight and bias parameters are
37 iid, the post-activations $h_i^1, h_{i'}^1$ are independent for different units, $i \neq i'$. Therefore we can consider
38 a single unit i in our analysis.

39

40 Since the post-activations $z_i^2(\mathbf{x})$ is a sum of iid terms, it follows from the central limit theorem
41 (CLT) that, as $D_1 \rightarrow \infty$, the distribution of $z_i^2(\mathbf{x})$ will tend towards a Gaussian. Furthermore,
42 from the multidimensional CLT, any finite collection $\{z_i^2(\mathbf{x}_1), \dots, z_i^2(\mathbf{x}_N)\}$ will tend towards a joint
43 Gaussian, which is the definition of a GP. We have one GP for each output dimension i , but the
44 parameters will be the same for each i .

45 The mean of this process is given by $\mu(\mathbf{x}) = \mathbb{E}[f_i(\mathbf{x})] = \mathbb{E}[z_i^2(\mathbf{x})] = 0$, since the parameters have
46 zero mean. Using the independence of the parameters, we can derive the covariance of this process

47

1
2 as follows:⁴

3
4 $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbb{E}[f_i(\mathbf{x})f_i(\mathbf{x}')] = \mathbb{E}[z_i^2(\mathbf{x})z_i^2(\mathbf{x}')] \quad (18.172)$

5
6 $= \mathbb{E}\left[\left(b_i^2 + \sum_{j=1}^{D_1} W_{ij}^2 h_j^1(\mathbf{x})\right) \left(b_i^2 + \sum_{j=1}^{D_1} W_{ij}^2 h_j^1(\mathbf{x}')\right)\right] \quad (18.173)$

7
8 $= \mathbb{E}[(b_i^2)(b_i^2)] + \mathbb{E}\left[\sum_{j=1}^{D_1} (W_{ij}^2)(W_{ij}^2) h_j^1(\mathbf{x})h_j^1(\mathbf{x}')\right] \quad (18.174)$

9
10 $= \sigma_b^2 + \sum_{j=1}^{D_1} \frac{\sigma_w^2}{D_1} \mathbb{E}[h_j^1(\mathbf{x})h_j^1(\mathbf{x}')] \quad (18.175)$

11
12 $= \sigma_b^2 + \sigma_w^2 C(\mathbf{x}, \mathbf{x}') \quad (18.176)$

13 where we have defined

14
15 $C(\mathbf{x}, \mathbf{x}') \triangleq \mathbb{E}[\varphi(b + \mathbf{w}^\top \mathbf{x})\varphi(b + \mathbf{w}^\top \mathbf{x}')] \quad (18.177)$

16 where expectations are wrt b, \mathbf{w} .

17 In some cases we can compute the above Gaussian integral analytically. For example, for the erf activation function, [Wil98] derive the following result:

18
19 $C(\mathbf{x}, \mathbf{x}') = \frac{2}{\pi} \sin^{-1}\left(\frac{2\mathbf{x} \cdot \mathbf{x}}{\sqrt{(1+2\mathbf{x} \cdot \mathbf{x})(1+2\mathbf{x}' \cdot \mathbf{x}')}}\right) \quad (18.178)$

20 This is sometimes called the **neural net kernel**. Note that this is a **nonstationary kernel** (i.e.,
21 not a function of $\|\mathbf{x} - \mathbf{x}'\|$), and sample paths from it are nearly discontinuous and tend to constant
22 values for large positive or negative inputs. See Figure 18.29 for an illustration.

23 [CS09] extend the above result to the case of ReLU activations. However, in general we need to
24 use Monte Carlo approximation, or numerical integration, to compute the kernel function.

33 18.7.1.2 Result for deep MLPs

34 We can extend the above result by induction to show that the activations in every layer are distributed
35 as a GP, if we let each of the hidden layer widths go to infinity. More precisely, let us assume that
36 the output from the previous layer, $\{z_j^{\ell-1}(\mathbf{x})\}$, is a GP for each j . Now consider layer ℓ :

37
38 $z_i^\ell(\mathbf{x}) = b_i^\ell + \sum_{j=1}^{D_{\ell-1}} W_{ij}^\ell h_j^{\ell-1}(\mathbf{x}), \quad h_j^{\ell-1}(\mathbf{x}) = \varphi(z_j^{\ell-1}(\mathbf{x})) \quad (18.179)$

39 The $\{h_j^{\ell-1}(\mathbf{x})\}_j$ are iid random variables, since the inputs $\{z_j^{\ell-1}(\mathbf{x})\}_j$ are also iid (being jointly
40 Gaussian but with zero covariance between units j). Thus as $D_\ell \rightarrow \infty$, we have that $z_i^\ell \sim \text{GP}(0, \mathcal{K}^\ell)$,

41 42 43 4. We are using the fact that $u \sim \mathcal{N}(0, \sigma^2)$ implies $\mathbb{E}[u^2] = \mathbb{V}[u] = \sigma^2$.

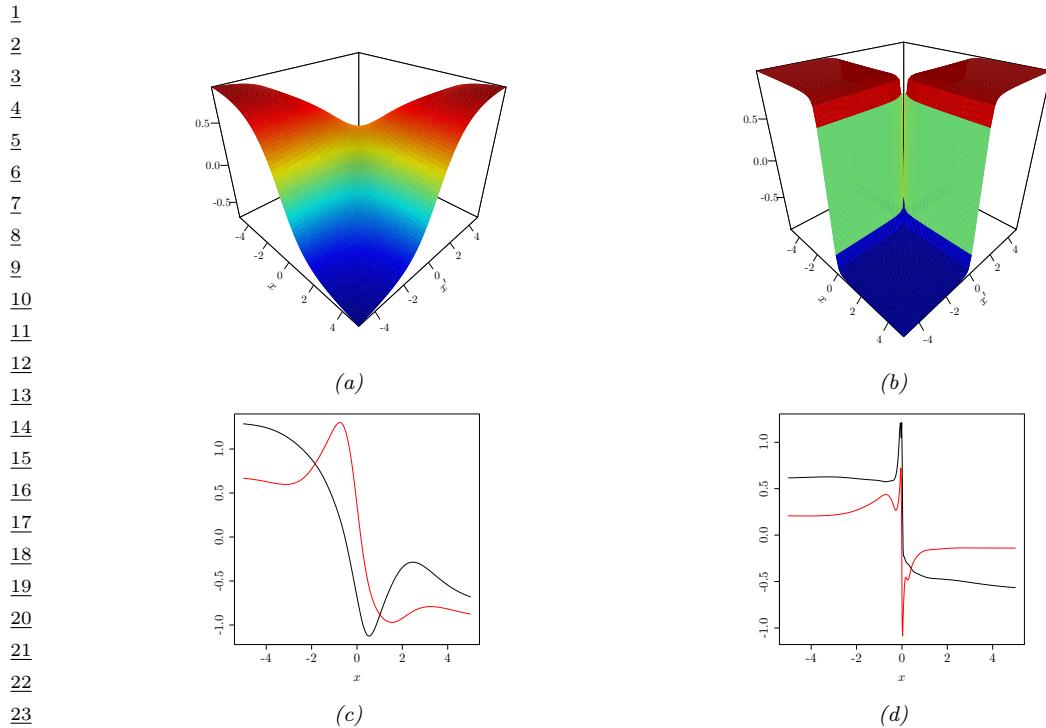


Figure 18.29: Illustration of the neural net kernel, corresponding to $\mathcal{K}(x, x') = \mathbb{E}[\text{erf}(w_0 + w_1 x)\text{erf}(w_0 + w_1 x')]$, where $w_0 \sim \mathcal{N}(0, \sigma_0^2)$ and $w_1 \sim \mathcal{N}(0, \sigma_1^2)$. (a) We plot $\mathcal{K}(x, x')$ for $\sigma_1 = 1$ (b) $\sigma_1 = 50$. In both cases, $\sigma_0 = 1$. (c-d). We plot samples from $f \sim \text{GP}(0, \mathcal{K})$. From Figure 2 of [Moh+19b]. Used with kind permission of Hossein Mohammadi.

30 where

$$_{32} \quad \mathcal{K}^\ell(\mathbf{x}, \mathbf{x}') = \mathbb{E} [z_i^\ell(\mathbf{x}) z_i^\ell(\mathbf{x}')] \quad (18.180)$$

$$= \sigma_w^2 + \sigma_w^2 \mathbb{E}_{z_i^{\ell-1} \sim \text{GP}(0, \kappa^{\ell-1})} [\varphi(z_i^{\ell-1}(\mathbf{x})) \varphi(z_i^{\ell-1}(\mathbf{x}'))] \quad (18.181)$$

$$= \sigma_b^2 + \sigma_w^2 \mathcal{C} \begin{pmatrix} \mathcal{K}^{l-1}(x, x) & \mathcal{K}^{l-1}(x, x') \\ \mathcal{K}^{l-1}(x', x) & \mathcal{K}^{l-1}(x', x') \end{pmatrix} \quad (18.182)$$

$$37 \quad \mathcal{C}(\Sigma) \triangleq \mathbb{E} [\varphi(u)\varphi(v)], \quad (u, v) \sim \mathcal{N}(\mathbf{0}, \Sigma) \quad (18.183)$$

For the base case \mathcal{K}^1 , we have

$$\frac{40}{\mathcal{K}} \mathcal{K}^1(\mathbf{x}, \mathbf{x}') = \mathbb{E}[z_i^1(\mathbf{x}) z_i^1(\mathbf{x}')] = \mathbb{E}[(b + \mathbf{w}^\top \mathbf{x})(b + \mathbf{w}^\top \mathbf{x}')] \quad (18.184)$$

$$= \mathbb{E}[b^2] + \mathbb{E}[(\mathbf{x}^\top \mathbf{w})(\mathbf{w}^\top \mathbf{x}')]=\sigma_b^2 + \frac{\sigma_w^2}{D_1}\mathbf{x}^\top \mathbf{x}' \quad (18.185)$$

⁴⁴ See Figure 18.30 for an illustration of how the $N \times N$ covariance matrix between examples in a minibatch evolves across layers, and how this compares to the $D_\ell \times N$ matrix of activations produced by a parametric model.

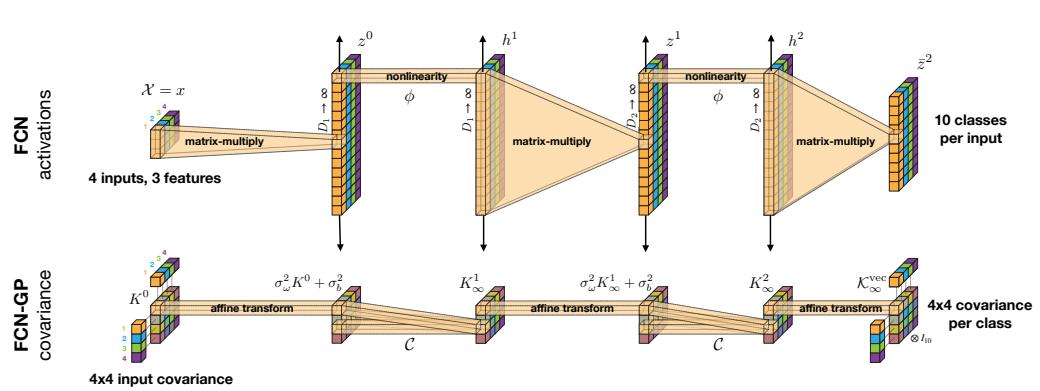


Figure 18.30: Comparison of a parametric fully connected network (FCN) and its GP equivalent. Here there are $N = 4$ input examples, each of which are $D_0 = 3$ dimensional. From Figure 3 of [Nov+19]. Used with kind permission of Roman Novak.

18.7.2 Kernels derived from trained DNNs (neural tangent kernel)

Although it is useful to convert a random, untrained DNN to a GP in order to gain insight into the corresponding implicit prior over functions, it is even more useful to derive the posterior over functions that result from training a DNN to minimize $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D})$.

Suppose we just train the final layer weights to compute $\hat{\boldsymbol{\theta}}^L$; let the resulting parameters be $\boldsymbol{\theta} = (\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^{L-1}, \hat{\boldsymbol{\theta}}^L)$ where all are random except the last. The resulting prediction $f(\mathbf{x}; \hat{\boldsymbol{\theta}})$ will be equivalent to exact GP inference $\mathbb{E}[f(\mathbf{x})|\mathcal{D}]$ using the NN-GP kernel.

Now suppose we train all the weights. Let $\mathbf{f} = [f(\mathbf{x}_n; \boldsymbol{\theta})]_{n=1}^N$ be the $N \times 1$ prediction vector, let $\nabla_f \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial f(\mathbf{x}_n)}]_{n=1}^N$ be the $N \times 1$ loss gradient vector, let $\boldsymbol{\theta} = [\theta_p]_{p=1}^P$ be the $P \times 1$ vector of parameters, and let $\nabla_{\boldsymbol{\theta}} \mathbf{f} = [\frac{\partial f(\mathbf{x}_n)}{\partial \theta_p}]$ be the $P \times N$ matrix of partials. Suppose we perform continuous time gradient descent with fixed learning rate η . The parameters evolve over time as follows:

$$\partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{f}_t) = -\eta \nabla_{\boldsymbol{\theta}} \mathbf{f}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) \quad (18.186)$$

Thus the function evolves over time as follows:

$$\partial_t \mathbf{f}_t = \nabla_{\boldsymbol{\theta}} \mathbf{f}_t^\top \partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \mathbf{f}_t^\top \nabla_{\boldsymbol{\theta}} \mathbf{f}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) = -\eta \mathcal{T}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) \quad (18.187)$$

where \mathcal{T}_t is the $N \times N$ kernel matrix

$$\mathcal{T}_t(\mathbf{x}, \mathbf{x}') \triangleq \nabla_{\boldsymbol{\theta}} f_t(\mathbf{x}) \cdot \nabla_{\boldsymbol{\theta}} f_t(\mathbf{x}') = \sum_{p=1}^P \frac{\partial f(\mathbf{x}; \boldsymbol{\theta})}{\partial \theta_p} \Big|_{\boldsymbol{\theta}_t} \frac{\partial f(\mathbf{x}'; \boldsymbol{\theta})}{\partial \theta_p} \Big|_{\boldsymbol{\theta}_t} \quad (18.188)$$

If we let the learning rate η become infinitesimally small, and the widths go to infinity, one can show that this kernel converges to a constant matrix, this is known as the **neural tangent kernel** or **NTK** [JGH18]:

$$\mathcal{T}(\mathbf{x}, \mathbf{x}') \triangleq \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_\infty) \cdot \nabla_{\boldsymbol{\theta}} f(\mathbf{x}'; \boldsymbol{\theta}_\infty) \quad (18.189)$$

1 2 **18.7.2.1 Computing the NTK**

3 We can compute the NTK in a recursive fashion, similar to the computation of the NN-GP kernel in
4 Section 18.7.1.1. In particular, let us define

5 6 $\dot{\mathcal{C}}(\Sigma) \triangleq \mathbb{E}[\varphi'(u)\varphi'(v)], \quad (u, v) \sim \mathcal{N}(\mathbf{0}, \Sigma)$ (18.190)

7 Then one can show, by induction, that

8 9 10 $\mathcal{T}^l(\mathbf{x}, \mathbf{x}') = \mathcal{K}^l(\mathbf{x}, \mathbf{x}') + \sigma_w^2 \mathcal{T}^{l-1}(\mathbf{x}, \mathbf{x}') \dot{\mathcal{C}} \begin{pmatrix} \mathcal{K}^{l-1}(\mathbf{x}, \mathbf{x}) & \mathcal{K}^{l-1}(\mathbf{x}, \mathbf{x}') \\ \mathcal{K}^{l-1}(\mathbf{x}', \mathbf{x}) & \mathcal{K}^{l-1}(\mathbf{x}', \mathbf{x}') \end{pmatrix}$ (18.191)

11 where \mathcal{K} is the NN-GP kernel.

12 Details on how to compute the $\dot{\mathcal{C}}$ function for various models, such as CNNs, graph neural nets,
13 and general neural nets, can be found in [Aro+19; Du+19; Yan19]. A software library to compute the
14 NNGP and NTK is available in [Ano19].

15 16 **18.7.2.2 Connections with GPs**

17 Suppose \mathcal{L} is the square loss, and \mathbf{y} is the true prediction targets, so

18 19 20 21 $\nabla_{\mathbf{f}} \mathcal{L}(\mathbf{f}_t) = \frac{1}{2N} \nabla_{\mathbf{f}} \|\mathbf{f}_t - \mathbf{y}\|_2^2 = \frac{1}{N} (\mathbf{f}_t - \mathbf{y})$ (18.192)

22 From Equation (18.187), and since the NTK kernel converges to a constant matrix, the function
23 evolves linearly over time as follows:

24 25 26 $\partial_t \mathbf{f}_t = -\eta \mathcal{T} \nabla_{\mathbf{f}} \mathcal{L}(\mathbf{f}_t) = -\eta \mathcal{T}(\mathbf{f}_t - \mathbf{y})$ (18.193)

27 We can solve this differential equation to get $f_t(\mathbf{x}) = \mathcal{T}(\mathbf{x}, \mathbf{X}) \mathcal{T}(\mathbf{X}, \mathbf{X})^{-1} \mathbf{H}_t \mathbf{y}$, where $\mathbf{H}_t \triangleq \mathbf{I} - e^{-\eta t \mathcal{T}(\mathbf{X}, \mathbf{X})}$. Hence this converges to

28 29 30 $f_\infty(\mathbf{x}) = \mathcal{T}(\mathbf{x}, \mathbf{X}) \mathcal{T}(\mathbf{X}, \mathbf{X})^{-1} \mathbf{y} = \mathcal{T}(\mathbf{x}, \mathbf{X})^\top \boldsymbol{\alpha}$ (18.194)

31 where $\boldsymbol{\alpha} = \mathcal{T}(\mathbf{X}, \mathbf{X})^{-1} \mathbf{y}$. Thus we see that this is a linear model, $f(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\alpha}$, using a fixed set of
32 basis functions, $\boldsymbol{\phi}(\mathbf{x}) = [\mathcal{T}(\mathbf{x}, \mathbf{x}_1), \dots, \mathcal{T}(\mathbf{x}, \mathbf{x}_N)]$. This has the same form as kernel ridge regression
33 (Section 18.3.7) without a ridge penalty term. This is known as “ridgeless regression” [LR18], and
34 corresponds to a model that perfectly interpolates the training data. This also matches the mean of
35 a GP with no observation noise. However, the variance of $f_\infty(\mathbf{x})$ is different from a GP.
36

37 38 **18.7.2.3 Discussion**

39 The assumptions behind the NTK results in the parameters barely changing from their initial values
40 (which is why a linear approximation around the starting parameters is valid). This can still lead
41 to a change in the final predictions (and zero final training error), because the final layer weights
42 can learn to use the random features just like in kernel regression. However, this phenomenon —
43 which has been called “lazy training” [COB18] — is not representative of DNN behavior in practice
44 [Woo+19], where parameters often change a lot.

45 A theoretical analysis that more closely matches practice is currently being developed by various
46 authors, but is beyond the scope of this chapter (see e.g., [Fan+19] for one of many recent works).

47

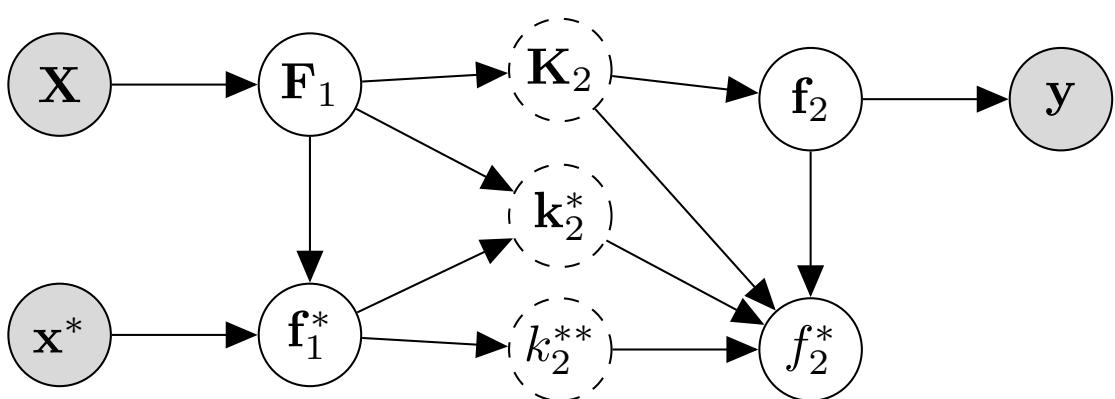


Figure 18.31: Graphical model corresponding to a deep GP with 2 layers. The dashed nodes are deterministic functions of their parents, and represent kernel matrices. The shaded nodes are observed, the unshaded nodes are hidden. From Figure 5 of [PC21]. Used with kind permission of Geoff Pleis.

18.7.3 Deep GPs

A **deep Gaussian process** or **DGP** is a composition of GPs [DL13]. (See [Jak21] for a recent survey.) More formally, a DGP of L layers is a hierarchical model of the form

$$\text{DGP}(\mathbf{x}) = f_L \circ \dots \circ f_1(\mathbf{x}), \quad f_i(\cdot) = [f_i^{(1)}(\cdot), \dots, f_i^{(H_i)}(\cdot)], \quad f_i^{(j)} \sim \text{GP}(0, \mathcal{K}_i(\cdot, \cdot)) \quad (18.195)$$

This is similar to a deep neural network, except the hidden nodes are now hidden functions.

A natural question is: what is gained by this approach compared to a standard GP? Although conventional single-layer GPs are nonparametric, and can model any function (assuming the use of a non-degenerate kernel) with enough data, in practice their performance is limited by the choice of kernel. This can be partially overcome by using a DGP, as we show in Section 18.7.3.2. Unfortunately, posterior inference in DGPs is challenging, as we discuss in Section 18.7.3.3.

In Section 18.7.3.4, we discuss the expressive power of infinitely wide DGPs, and in Section 18.7.3.5 we discuss connections with DNNs.

18.7.3.1 Construction of a deep GP

In this section we give an example of a 2 layer DGP, following the presentation in [PC21]. Let $f_1^{(j)} \sim \text{GP}(0, \mathcal{K}_1)$ for $j = 1 : H_1$, where H_1 is the number of hidden units, and $f_2 \sim \text{GP}(0, \mathcal{K}_2)$. Assume we have labeled training data $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$ and $\mathbf{y} = (y_1, \dots, y_N)$. Define $\mathbf{F}_1 = [f_1(\mathbf{x}_1), \dots, f_1(\mathbf{x}_N)]$ and $\mathbf{f}_2 = [f_2(f_1(\mathbf{x}_1)), \dots, f_2(f_1(\mathbf{x}_N))]$. Let \mathbf{x}^* be a test input and define $\mathbf{f}_1^* = f_1(\mathbf{x}^*)$ and $\mathbf{f}_2^* = f_2(\mathbf{f}_1^*)$. The corresponding joint distribution over all the random variables is given by

$$p(\mathbf{f}_2^*, \mathbf{f}_2, \mathbf{F}_1, \mathbf{f}_1, \mathbf{y}) = p(\mathbf{f}_2^* | \mathbf{f}_2, \mathbf{f}_1^*, \mathbf{F}_1) p(\mathbf{f}_2 | \mathbf{F}_1, \mathbf{f}_1^*) p(\mathbf{f}_1^*, \mathbf{F}_1) p(\mathbf{y} | \mathbf{f}_2) \quad (18.196)$$

where we drop the dependence on \mathbf{X} and \mathbf{x}^* for brevity. This is illustrated by the graphical model in Figure 18.31, where we define $\mathbf{K}_2 = \mathcal{K}_2(\mathbf{F}_1, \mathbf{F}_1)$, $\mathbf{k}_{2*} = \mathcal{K}_2(\mathbf{F}_1, \mathbf{f}_1^*)$, and $\mathbf{k}_{2**} = \mathcal{K}_2(\mathbf{f}_1^*, \mathbf{f}_1^*)$.

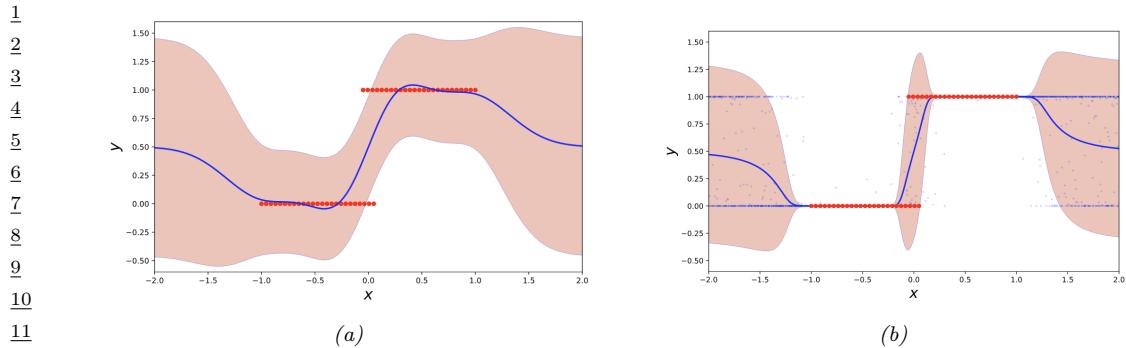


Figure 18.32: Some data (red points) sampled from a step function fit with (a) a standard GP with RBF kernel and (b) a deep GP with 4 layers of RBF kernels. The solid blue line is the posterior mean. The pink shaded area represents the posterior variance ($\mu(x) \pm 2\sigma(x)$). The thin blue dots in (b) represent posterior samples. From [Law19]. Used with kind permission of Neil Lawrence.

18.7.3.2 Example: 1d step function

Suppose we have data from a piecewise constant function. (This can often happen when modeling certain physical processes, which can exhibit saturation effects.) Figure 18.32a shows what happens if we fit data from such a step function using a standard GP with an RBF (Gaussian) kernel. Obviously this method oversmooths the function and does not “pick up on” the underlying discontinuity. It is possible to learn kernels that can capture such discontinuous (non-stationary) behavior by learning to warp the input with a neural net before passing into the RBF kernel (see Figure 18.26).

Another approach is to learn a sequence of smooth mappings which together capture the overall complex behavior, analogous to the approach in deep learning. Suppose we fit a 4 layer DGP with a single hidden unit at each layer; we will use an RBF kernel. Thus the kernel at level 1 is $\mathcal{K}_1(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2/(2D))$, the kernel at level 2 is $\mathcal{K}_2(\mathbf{f}_1(\mathbf{x}), \mathbf{f}_1(\mathbf{x}')) = \exp(-\|\mathbf{f}_1(\mathbf{x}) - \mathbf{f}_1(\mathbf{x}')\|^2/(2H_1))$, etc.

We can perform posterior inference in this model to compute $p(\mathbf{f}_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y})$ for a set of test points \mathbf{x}_* (see Section 18.7.3.3 for the details). Figure 18.32b shows the resulting posterior predictive distribution. We see that the predictions away from the data capture two plausible modes: either the signal continues at the level $y = 0$ or at $y = 1$. (The posterior mean, shown by the solid blue line, is a poor summary of the predictive distribution in this case, since it lies between these two modes.) This is an example of non-trivial extrapolation behavior outside of the support of the data.

Figure 18.33 shows the individual functions learned at each layer (these are all maps from 1d to 1d). We see that the functions are individually smooth (since they are derived from an RBF kernel), but collectively they define non-smooth behavior.

47

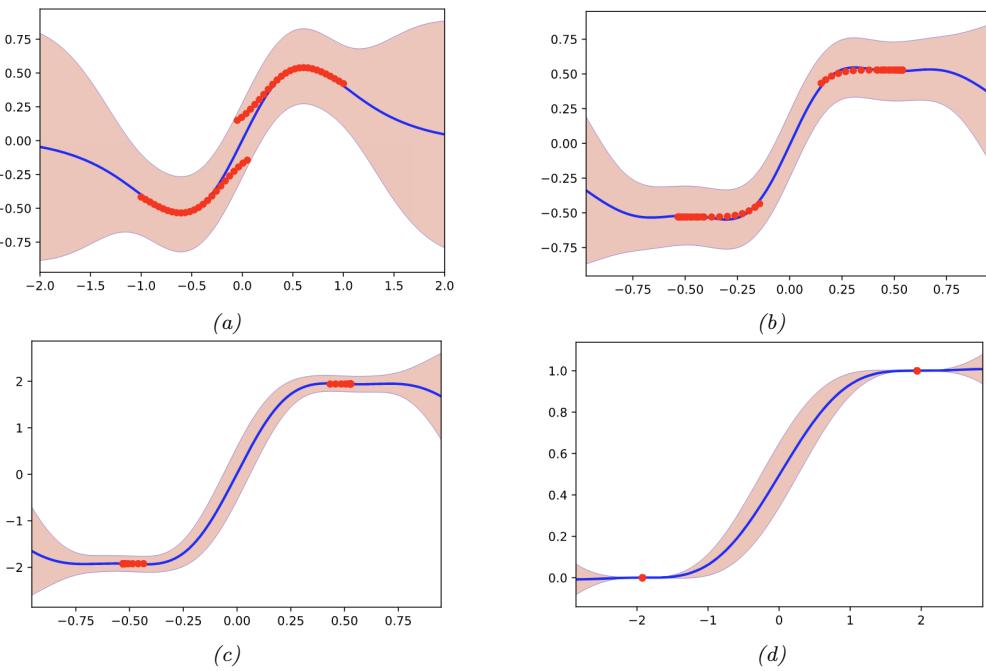


Figure 18.33: Illustration of the functions learned at each layer of the DGP. (a) Input to layer 1. (b) Layer 2 to layer 1. (c) Layer 2 to layer 3. (d) Layer 3 to output. From [Law19]. Used with kind permission of Neil Lawrence.

18.7.3.3 Posterior inference

In Equation (18.196), we defined the joint distribution defined by a (2 layer) DGP. We can condition on \mathbf{y} to convert this into a joint posterior, as follows:

$$p(f_2^*, f_2, \mathbf{F}_1, f_1 | \mathbf{y}) = p(f_2^* | f_2, f_1^*, \mathbf{F}_1, \mathbf{y}) p(f_2 | \mathbf{F}_1, f_1^*, \mathbf{y}) p(f_1^*, \mathbf{F}_1 | \mathbf{y}) \quad (18.197)$$

$$= p(f_2^* | f_2, f_1^*, \mathbf{F}_1) p(f_2 | \mathbf{F}_1, \mathbf{y}) p(f_1^*, \mathbf{F}_1 | \mathbf{y}) \quad (18.198)$$

where the simplifications in the second line follow from the conditional independencies encoded in Figure 18.31. Note that f_2 and f_2^* depend on \mathbf{F}_1 and f_1^* only through \mathbf{K}_2 , \mathbf{k}_2^* and k_2^{**} , where

$$p(f_2 | \mathbf{K}_2) \sim \mathcal{N}(\mathbf{0}, \mathbf{K}_2), \quad p(f_2^* | k_2^{**}, \mathbf{k}_2^*, \mathbf{K}_2, f_2) \sim \mathcal{N}((\mathbf{k}_2^*)^\top \mathbf{K}_2^{-1} f_2, k_2^{**} - (\mathbf{k}_2^*)^\top \mathbf{K}_2^{-1} \mathbf{k}_2^*) \quad (18.199)$$

Hence

$$p(f_2^*, f_2, \mathbf{F}_1, f_1 | \mathbf{y}) = p(f_2^* | f_2, \mathbf{K}_2, \mathbf{k}_2^*, k_2^{**}) p(f_2 | \mathbf{K}_2, \mathbf{y}) p(\mathbf{K}_2, \mathbf{k}_2^*, k_2^{**} | \mathbf{y}) \quad (18.200)$$

For prediction we only care about f_2^* , so we marginalize out the other variables. The posterior

1 mean is given by
2

$$\mathbb{E}_{f_2^*|\mathbf{y}}[f_2^*] = \mathbb{E}_{\mathbf{K}_2, \mathbf{k}_2^*, k_2^{**}|\mathbf{y}} [\mathbb{E}_{f_2|\mathbf{K}_2, \mathbf{y}} [\mathbb{E}_{f_2^*|f_2, \mathbf{K}_2, \mathbf{k}_2^*, k_2^{**}} [f_2^*]]] \quad (18.201)$$

$$= \mathbb{E}_{\mathbf{K}_2, \mathbf{k}_2^*, k_2^{**}|\mathbf{y}} [\mathbb{E}_{f_2|\mathbf{K}_2, \mathbf{y}} [(k_2^*)^\top \mathbf{K}_2^{-1} f_2]] \quad (18.202)$$

$$= \mathbb{E}_{\mathbf{K}_2, \mathbf{k}_2^*|\mathbf{y}} \left[\underbrace{\mathbf{k}_2^* \mathbf{K}_2^{-1} \mathbb{E}_{f_2|\mathbf{K}_2, \mathbf{y}} [f_2]}_{\alpha} \right] \quad (18.203)$$

10 Since \mathbf{K}_2 and \mathbf{k}_2^* are deterministic transformations of $\mathbf{f}_1(\mathbf{x}^*), \mathbf{f}_1(\mathbf{x}_1), \dots, \mathbf{f}_1(\mathbf{x}_N)$, we can rewrite this
11 as

$$\mathbb{E}_{f_2^*|\mathbf{y}}[f_2^*] = \mathbb{E}_{\mathbf{f}_1(\mathbf{x}^*), \mathbf{f}_1(\mathbf{x}_1), \dots, \mathbf{f}_1(\mathbf{x}_N)|\mathbf{y}} \left[\sum_{i=1}^N \alpha_i \mathcal{K}_2(\mathbf{f}_1(\mathbf{x}_i), \mathbf{f}_1(\mathbf{x}^*)) \right] \quad (18.204)$$

16 We see from the above that inference in a DGP is, in general, very expensive, due to the need to
17 marginalize over a lot of variables, corresponding to all the hidden function values at each layer at
18 each data point. In [SD17], they propose an approach to approximate inference in DGPs based on
19 the sparse variational method of Section 10.1.1. The key assumption is that each layer has a set of
20 inducing points, along with corresponding inducing values, that simplifies the dependence between
21 unknown function values within each layer. However, the dependence between layers is modeled
22 exactly. In [Dut+21] they show that the posterior mean of such a sparse variational approximation
23 can be computed by performing a forwards pass through a ReLU DNN.
24

25 18.7.3.4 Behavior in the limit of infinite width 26

27 Consider the case of a DGP where the depth is 2. The posterior mean of the predicted output
28 at a test point is given by Equation (18.204). We see that this is a mixture of data-dependent
29 kernel functions, since both \mathbf{K}_2 and \mathbf{k}_2 depend on the data \mathbf{y} . This is what makes deep GPs more
30 expressive than single layer GPs, where the kernel is fixed. However, [PC21] show that, in the limit
31 $H_1 \rightarrow \infty$, the posterior over the kernels for the layer 2 features becomes independent of the data,
32 i.e., $p(\mathbf{K}_2, \mathbf{k}_2^*|\mathbf{y}) = \delta(\mathbf{K}_2 - \mathbf{K}_{\lim}) \delta(\mathbf{k}_2^* - \mathbf{k}_{\lim}^*)$, where $\mathbf{K}_{\lim} = \mathbb{E}[f_2 f_2^\top]$ and $\mathbf{k}_{\lim}^* = \mathbb{E}[f_2 f_2^*]$, where the
33 expectations depend on \mathbf{X} but not \mathbf{y} . Consequently the posterior predictive mean reduces to

$$\lim_{H_1 \rightarrow \infty} \mathbb{E}_{f_2^*|\mathbf{y}}[f_2^*] = \sum_{i=1}^N \alpha_i \mathcal{K}_{\lim}(\mathbf{x}_i, \mathbf{x}_*) \quad (18.205)$$

38 which is the same form as a single layer GP.

39 As a concrete example, consider a 2 layer DGP with an RBF kernel at each layer. Thus the kernel
40 at level 1 is $\mathcal{K}_1(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2/(2D))$, and the kernel at level 2 is $\mathcal{K}_2(\mathbf{f}_1(\mathbf{x}), \mathbf{f}_1(\mathbf{x}')) =$
41 $\exp(-\|\mathbf{f}_1(\mathbf{x}) - \mathbf{f}_1(\mathbf{x}')\|^2/(2H_1))$. Let us fit this model to a noisy step function. In Figure 18.34 we
42 show the results as we increase the width of the hidden layer. When the width is 1, we see that
43 the covariance of the resulting DGP, $\mathcal{K}_2(\mathbf{f}_1(\mathbf{x}), \mathbf{f}_1(\mathbf{x}'))$, is nonstationary. In particular, there are
44 long-range correlations near $\mathbf{x} = \pm 1$ (since the function is constant in this region), but short range
45 correlations near $\mathbf{x} = 0$ (since the function is changing rapidly in this region). However, as the width
46 increases, we lose this nonstationarity, as shown by the constant diagonals of the kernel matrix. Indeed,
47

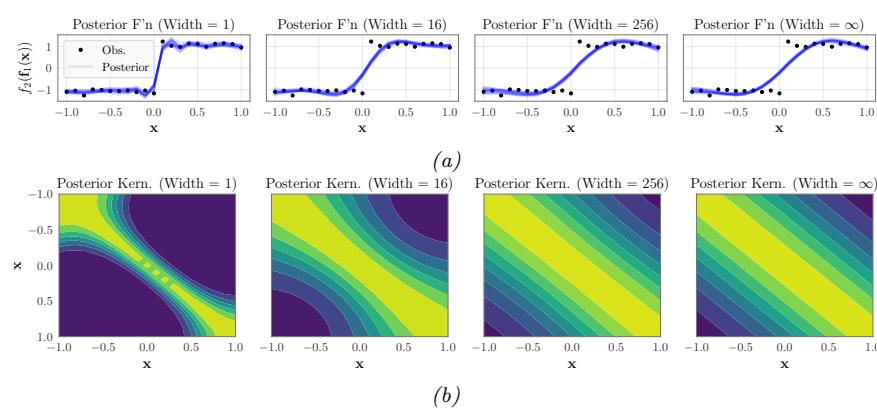


Figure 18.34: (a) Posterior of 2-layer RBF deep GP fit to a noisy step function. Columns represent width of 1, 16, 256 and infinity. (b) Average posterior covariance of the DGP, given by $\mathbb{E}_{\mathbf{f}_1(\mathbf{x}), \mathbf{f}_1(\mathbf{x}') | \mathbf{y}} [\mathcal{K}_2(\mathbf{f}_1(\mathbf{x}), \mathbf{f}_1(\mathbf{x}'))]$. As the width increases, the covariance becomes stationary, as shown by the kernel's constant diagonals. From Figure 1 of [PC21]. Used with kind permission of Geoff Pleiss.

in [PC21, App. G] they prove that the limiting kernel is $\mathcal{K}_{\lim}(\mathbf{x}, \mathbf{x}') = \exp(\exp(-\|\mathbf{x} - \mathbf{x}'\|^2/(2D)) - 1)$, which is stationary.

In [PC21], they also show that increasing the width makes the marginals more Gaussian, due to central-limit like behavior. However, increasing the depth makes the marginals less Gaussian, and causes them to have sharper peaks and heavier tails. Thus one often gets best results with a deep GP if it is deep but narrow.

18.7.3.5 Connection with Bayesian neural networks

A Bayesian neural network (BNN) is a DNN in which we place priors over the parameters (see Section 17.1). One can show (see e.g., [OA21]) that BNNs are a degenerate form of deep GPs. For example, consider a 2 layer MLP, $f_2(\mathbf{f}_1(\mathbf{x}))$, with $\mathbf{f}_1 : \mathbb{R}^D \rightarrow \mathbb{R}^{H_1}$ and $f_2 : \mathbb{R}^{H_1} \rightarrow \mathbb{R}$, defined by

$$\mathbf{f}_1^{(i)}(\mathbf{x}) = (\mathbf{w}_1^{(i)})^\top \mathbf{x} + \beta \mathbf{b}_1, \quad f_2(\mathbf{z}) = \frac{1}{\sqrt{H_1}} \mathbf{w}_2^\top \varphi(\mathbf{z}) + \beta b_2 \quad (18.206)$$

where $\beta > 0$ is a scaling constant, and $\mathbf{W}_1, \mathbf{b}_1, \mathbf{w}_2, b_2$ are Gaussian. The first layer is a linear regression model, and hence (from the results in Section 18.3.3) corresponds to a GP with a linear kernel of the form $\mathcal{K}_1(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}'$. The second layer is also a linear regression model but applied to features $\varphi(\mathbf{z})$. Hence (from the results in Section 18.3.3) this corresponds to a GP with a linear kernel of the form $\mathcal{K}_2(\mathbf{z}, \mathbf{z}') = \varphi(\mathbf{z})^\top \varphi(\mathbf{z}')$. Thus each layer of the model corresponds to a (degenerate) GP, and hence the overall model is a (degenerate) DGP. (The term “degenerate” refers to the fact that the covariance matrices only have a finite number of non-zero eigenvalues, due to the use of a finite set of basis functions.) Consequently we can use the results from Section 18.7.3.4 to conclude that infinitely wide DNNs also reduce to a single layer GP, as we already established in Section 18.7.1.

In practice we use finite-width DNNs. The width should be wide enough to approximate a standard GP at each layer, but should not be too wide, otherwise the corresponding kernels of the resulting

1 deep GP will no longer be adapted to the data, i.e., there will not be any “feature learning”. See e.g.,
2 [[Ait20](#); [PC21](#); [ZVP21](#)] for details.
3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

19 Structured prediction

19.1 Introduction

In **structured prediction**, we want to learn a conditional model $p(\mathbf{y}|\mathbf{x})$, where the output $\mathbf{y} \in \mathcal{Y}$ lives in some structured space, such as the set of sequences, or the set of labels of nodes on a graph. In contrast to other kinds of conditional generative model, such as text-to-image generation, in structured prediction there are often constraints on the set of valid values of the output \mathbf{y} . For example, if we want to perform sentence parsing, the output set of tags should satisfy the rules of grammar. In some cases, the “constraints” are “soft”, rather than “hard”. For example, if we want to perform pixel labeling, we usually want to encourage the label at one location to be the same as its neighbors, unless the visual input strongly suggests a change in semantic content at this location (e.g., at the edge of an object). We can capture both of these scenarios by using conditional graphical models, as we discuss in Section 19.2.

Structured prediction can also be used for temporal prediction problems, where we condition on the past and generate the future, as we discuss in Section 19.3.

19.2 Conditional random fields (CRFs)

A **conditional random field** or **CRF** [LMP01] is a version of an MRF where all the clique potentials are conditioned on input features:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{x}, \boldsymbol{\theta})} \prod_c \psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) \quad (19.1)$$

(Note how the partition function now depends on the inputs \mathbf{x} as well as the parameters $\boldsymbol{\theta}$.)

If the potential functions are log-linear and have the form

$$\psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) = \exp(\boldsymbol{\theta}_c^\top \phi_c(\mathbf{x}, \mathbf{y}_c)) \quad (19.2)$$

then the result model is called a **conditional maxent model**. This can be thought of as an extension of logistic regression where we model the correlation amongst the output labels, conditioned on the input features. Of course, we can also use nonlinear potential functions, such as DNNs.

19.2.1 1d CRFs

In this section, we focus on 1d CRFs defined on chain-structured graphical models.

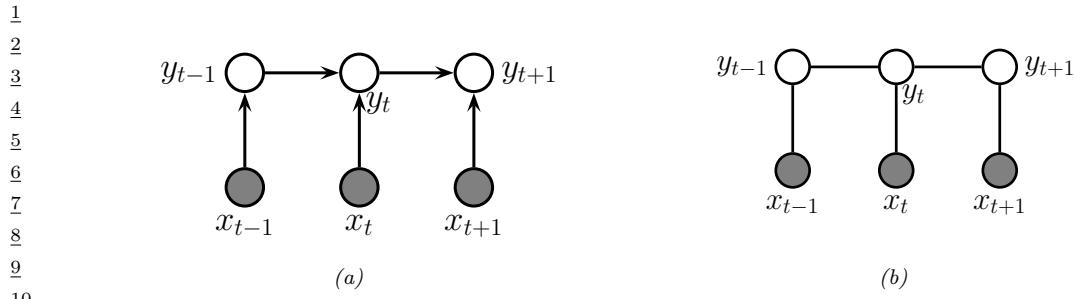


Figure 19.1: Two discriminative models for sequential data. (a) A directed model (MEMM). (b) An undirected model (CRF).

13

14

15 19.2.1.1 The label bias problem

16 We start by contrasting CRFs with **maximum entropy Markov models** (MEMMs), which is
 17 a directed Markov chain in which the state transition probabilities are conditioned on the input
 18 features, as shown in Figure 19.1(a). An MEMM seems like the natural generalization of classifiers to
 19 the structured-output setting, but it suffers from a subtle problem known (rather obscurely) as the
 20 **label bias** problem [LMP01]. The problem is that local features at time t do not influence states
 21 prior to time t . This follows by examining the DAG, which shows that x_t is d-separated from y_{t-1}
 22 (and all earlier time points) by the v-structure at y_t , thus blocking the information flow backwards
 23 in time.

24 To understand what this means in practice, consider the **part of speech tagging** task, in which
 25 we must label every word in the sentence with a part of speech (POS), such as noun, verb, etc.
 26 Suppose we see the word “banks”; this could be a verb (as in “he banks at Chase”), or a noun (as
 27 in “the river banks were overflowing”). Locally the POS tag for the word is ambiguous. However,
 28 suppose that later in the sentence, we see the word “fishing”; this gives us enough context to infer
 29 that the sense of “banks” is “river banks”. However, in an MEMM the “fishing” evidence will not flow
 30 backwards, so we will not be able to infer the correct label for “banks”.

31 Now consider a chain-structured CRF. This model has the form

$$32 \quad p(\mathbf{y}_{1:T} | \mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{x}, \boldsymbol{\theta})} \prod_{t=1}^T \psi(y_t; \mathbf{x}, \boldsymbol{\theta}) \prod_{t=1}^{T-1} \psi(y_t, y_{t+1}; \mathbf{x}, \boldsymbol{\theta}) \quad (19.3)$$

33 From the graph in Figure 19.1(b) we see that the label bias problem no longer exists, since y_t does
 34 not block the information from x_t from reaching other $y_{t'}$ nodes, due to the lack of directed edges.

35 The label bias problem in MEMMs occurs because directed models are **locally normalized**,
 36 meaning each CPD sums to 1. By contrast, MRFs and CRFs are **globally normalized**, which
 37 means that local factors do not need to sum to 1, since the partition function Z , which sums over all
 38 joint configurations, will ensure the model defines a valid distribution.

39 However, this solution comes at a price: we do not get a valid probability distribution over $\mathbf{y}_{1:T}$
 40 until we have seen the whole sentence, since only then can we normalize over all configurations.
 41 Consequently, CRFs are not as useful as directed probabilistic graphical models (PGM-D) for online
 42 or real-time inference. Furthermore, the fact that Z is a function of all the parameters makes CRFs
 43 less modular and much slower to train than PGM-D’s, as we discuss in Section 19.2.3.

44

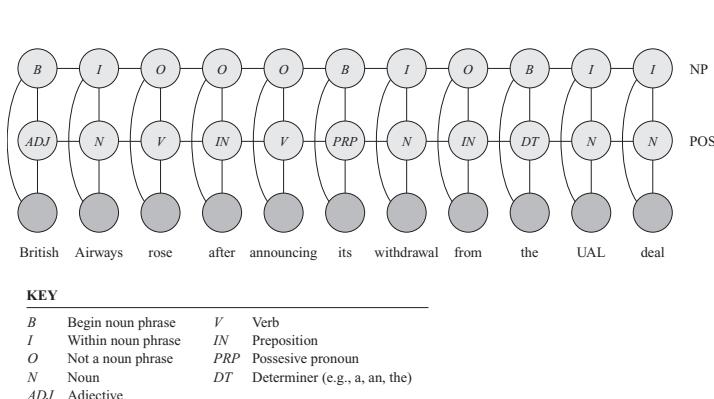


Figure 19.2: A CRF for joint POS (part of speech) tagging and NP (noun phrase) segmentation. From Figure 4.E.1 of [KF09a]. Used with kind permission of Daphne Koller.

19.2.1.2 Applications to NLP

CRFs used to be widely used in the natural language processing (NLP) community, before the advent of RNNs and transformers. Fortunately, we can get the best of both worlds by combining CRFs with DNNs. In particular, instead of using local potentials defined in terms of linear classifiers on pre-defined features $\phi(y_t, \mathbf{x})$,

$$\psi(y_t|x) = \exp(\mathbf{w}^\top \phi(y_t, x)) \quad (19.4)$$

we can use DNN classifiers instead. This is called a **neural CRF** [DK15b]. The pairwise edge potentials $\psi(y_t, y_{t+1}; \boldsymbol{x}, \boldsymbol{\theta})$, can often be manually designed to encode prior knowledge or constraints, as we illustrate below.

19.2.1.3 Noun phrase chunking

³⁴ One common NLP task is **noun phrase chunking**, which refers to the task of segmenting a sentence
³⁵ into its distinct noun phrases (NPs). This can be implemented as a seq2seq problem, in which each
³⁶ word is labeled with **BIO** tags (begin / inside / outside). A standard approach to this problem
³⁷ would first convert the string of words into a string of **POS** (part of speech tags), and then convert
³⁸ the POS tags to a string of BIOS. However, such a **pipeline** method can propagate errors. A more
³⁹ robust approach is to build a joint probabilistic model of the form $p(\text{BIO}_{1:T}, \text{POS}_{1:T} | \text{words}_{1:T})$. One
⁴⁰ way to do this is to use the CRF in Figure 19.2. The connections between adjacent labels encode the
⁴¹ probability of transitioning between the B, I and O states, and can enforce constraints such as the
⁴² fact that B (begin) must precede I (inside).

Inference in the model in Figure 19.2 is tractable, since it is essentially a “fat chain”, so we can use the forwards-backwards algorithm (Section 8.3.3) for exact inference in $O(T|\text{POS}|^2|\text{NP}|^2)$ time, where $|\text{POS}|$ is the number of POS tags, and $|\text{NP}|$ is the number of NP tags. However, the seemingly similar graph in Figure 19.3, to be explained below, is computationally intractable.

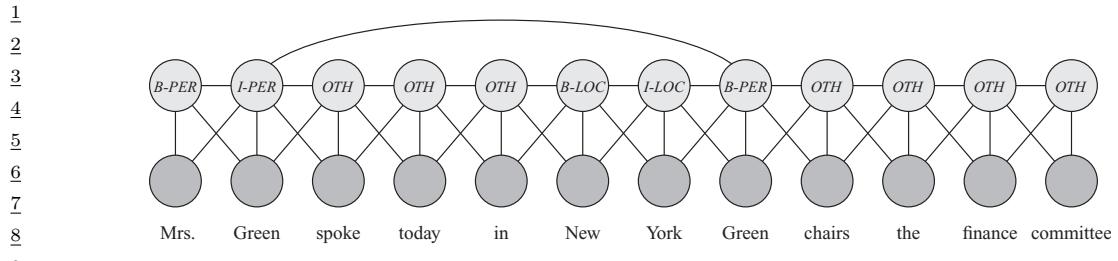


Figure 19.3: A skip-chain CRF for named entity recognition. From Figure 4.E.1 of [KF09a]. Used with kind permission of Daphne Koller.

19.2.1.4 Named entity recognition

A task that is related to NP chunking is **named entity extraction**. A simple approach to this is to extend the BIO notation to {B-Per, I-Per, B-Loc, I-Loc, B-Org, I-Org, Other }. We can then apply a linear CRF.

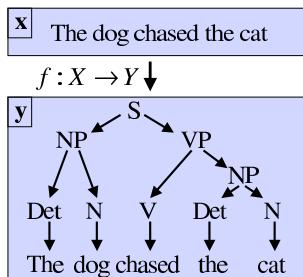
However, sometimes it is ambiguous whether a word is a person, location, or something else. (Proper nouns are particularly difficult to deal with because they belong to an **open class**, that is, there is an unbounded number of possible names, unlike the set of nouns and verbs, which is large but essentially fixed.) We can get better performance by considering long-range correlations between words. For example, we might add a link between all occurrences of the same word, and force the word to have the same tag in each occurrence. (The same technique can also be helpful for resolving the identity of pronouns.) This is known as a **skip-chain CRF**. See Figure 19.3 for an illustration, where we show that the word “Green” is interpreted as a person in both occurrences within the same sentence.

We see that the graph structure itself changes depending on the input, which is an additional advantage of CRFs over generative models. Unfortunately, inference in this model is generally more expensive than in a simple chain with local connections because of the larger treewidth (see Section 9.4.2).

19.2.1.5 Natural language parsing

A generalization of chain-structured models for language is to use probabilistic grammars. In particular, a probabilistic **context free grammar** or **PCFG** in Chomsky normal form is a set of re-write or production rules of the form $\sigma \rightarrow \sigma' \sigma''$ or $\sigma \rightarrow x$, where $\sigma, \sigma', \sigma'' \in \Sigma$ are non-terminals (analogous to parts of speech), and $x \in \mathcal{X}$ are terminals, i.e., words. Each such rule has an associated probability. The resulting model defines a probability distribution over sequences of words. We can compute the probability of observing a particular sequence $x = x_1 \dots x_T$ by summing over all trees that generate it. This can be done in $O(T^3)$ time using the **inside-outside algorithm**; see e.g., [JM08; MS99; Eis16] for details.

1 2 3 4 5 6 7 8 9 10



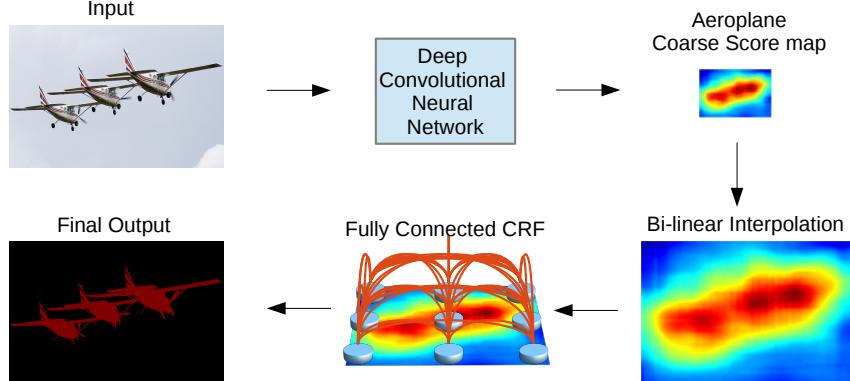
$$\Psi(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} 1 & S \rightarrow NPVP \\ 0 & S \rightarrow NP \\ 2 & NP \rightarrow Det N \\ 1 & VP \rightarrow V NP \\ \vdots & \\ 0 & Det \rightarrow dog \\ 2 & Det \rightarrow the \\ 1 & N \rightarrow dog \\ 1 & V \rightarrow chased \\ 1 & N \rightarrow cat \end{pmatrix}$$

11
12
13
14
15

Figure 19.4: Illustration of a simple parse tree based on a context free grammar in Chomsky normal form. The feature vector $\Psi(\mathbf{x}, \mathbf{y})$ counts the number of times each production rule was used, and is used to define the energy of a particular tree structure, $E(\mathbf{y}|\mathbf{x}) = -\mathbf{w}^\top \Psi(\mathbf{x}, \mathbf{y})$. The probability distribution over trees is given by $p(\mathbf{y}|\mathbf{x}) \propto \exp(-E(\mathbf{y}|\mathbf{x}))$. From Figure 5.2 of [AHT07]. Used with kind permission of Yasemin Altun.

16

17
18
19
20
21
22



29
30
31

Figure 19.5: A fully connected CRF is added to the output of a CNN, in order to increase the sharpness of the segmentation boundaries. From Figure 3 of [Che+15]. Used with kind permission of Jay Chen.

32

35
36
37
38
39
40
41

PCFGs are generative models. It is possible to make discriminative versions which encode the probability of a labeled tree, \mathbf{y} , given a sequence of words, \mathbf{x} , by using a CRF of the form $p(\mathbf{y}|\mathbf{x}) \propto \exp(\mathbf{w}^\top \Psi(\mathbf{x}, \mathbf{y}))$. For example, we might define $\Psi(\mathbf{x}, \mathbf{y})$ to count the number of times each production rule was used (which is analogous to the number of state transitions in a chain-structured model), as illustrated in Figure 19.4. We can also use a deep neural net to define the features, as in the **neural CRF parser** method of [DK15b].

48

42

43
44

45

19.2.2 2d CRFs

45

It is also possible to apply CRFs to image processing problems, which are usually defined on 2d grids. We give some examples below.

47

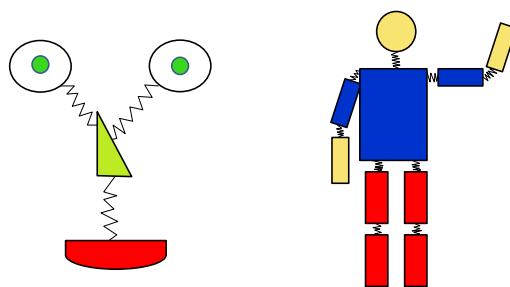


Figure 19.6: Pictorial structures model for a face and body. Each body part corresponds to a node in the CRF whose state space represents the location of that part. The edges (springs) represent pairwise spatial constraints. The local evidence nodes are not shown. Adapted from a figure by Pedro Felzenszwalb.

19.2.2.1 Semantic segmentation

The task of **semantic segmentation** is to assign a label to every pixel in an image. We can easily solve this problem using a CNN with one softmax output node per pixel. However, this may fail to capture long-range dependencies, since convolution is a local operation.

One way to get better results is to feed the output of the CNN into a CRF. Since the CNN already uses convolution, its outputs will usually already be locally smooth, so the benefits from using a CRF with a local grid structure may be quite small. However, we can sometimes get better results if we use a **fully connected CRF**, which has connections between all the pixels. This can capture long range connections which the grid-structured CRF cannot. See Figure 19.5 for an illustration, and [Che+17a] for details.

Unfortunately, exact inference in a fully connected CRF is intractable, but in the case of Gaussian potentials, it is possible to devise an efficient mean field algorithm, as described in [KK11]. Interestingly, [Zhe+15] showed how the mean field update equations can be implemented using a recurrent neural network (see Section 16.3.3), allowing end-to-end training. Alternatively, if we are willing to use a finite number of iterations, we can just “unroll” the computation graph and treat it as a fixed-sized feedforward circuit. The result is a graph-structured neural network, where the topology of the GNN is derived from the graphical model (c.f., Section 9.3.7). The advantage of this compared to standard CRF methods is that we can train this entire model end-to-end using standard gradient descent methods; we no longer have to worry about the partition function (see Section 19.2.3), or the lack of convergence that can arise when combining approximate inference with standard CRF learning.

19.2.2.2 Deformable parts models

Consider the problem of **object detection**, i.e., finding the location(s) of an object of a given class (e.g., a person or a car) in an image. One way to tackle this is to train a binary classifier that takes as input an image patch and specifies if the patch contains the object or not. We can then apply this to every image patch, and return the locations where the classifier has high confidence detections; this is known as a **sliding window detector**, and works quite well for rigid objects such as cars or frontal faces. Such an approach can be made efficient by using convolutional neural networks

1 (CNNs); see Section 16.3.2 for details.

2 However, such methods can work poorly when there is occlusion, or when the shape is deformable,
3 such as a person’s or animal’s body, because there is too much variation in the overall appearance.
4 A natural strategy to deal with such problems is break the object into parts, and then to detect
5 each part separately. But we still need to enforce spatial coherence of the parts. This can be done
6 using a pairwise CRF, where node y_i specifies the location of part i in the image (assuming it is
7 present), and where we connect adjacent parts by a potential function that encourages them to be
8 close together. For example, we can use a pairwise potential of the form $\psi(y_i, y_j) = \exp(-d(y_i, y_j))$,
9 where $y_i \in \{1, \dots, K\}$ is the location of part i (a discretization of the 2d image plane), and $d(y_i, y_j)$ is
10 the distance between parts i and j . In addition we will have a local evidence term of the form $p(y_i | \mathbf{x})$,
11 which can be any kind of discriminative classifier, such as a CNN, which predicts the distribution
12 over locations for part i given the image \mathbf{x} . The overall model has the form

$$\frac{15}{15} p(\mathbf{y} | \mathbf{x}) = \frac{1}{Z(\mathbf{x})} \left[\prod_i p(y_i | f(\mathbf{x})_i) \right] \left[\prod_{(i,j) \in E} \psi(y_i, y_j | \mathbf{x}) \right] \quad (19.5)$$

19 where E is the set of edges in the CRF, and $f(\mathbf{x})_i$ is the i ’th output of the CNN.

20 We can think of this CRF as a series of parts connected by springs, where the energy of the system
21 increases if the parts are moved too far from their expected relative distance. This is illustrated in
22 Figure 19.6. The resulting model is known as a **pictorial structure** [FE73], or **deformable parts**
23 **model** [Fel+10]. Furthermore, since this is a conditional model, we can make the spring strengths
24 be image dependent.

25 We can find the globally optimal joint configuration $\mathbf{y}^* = \operatorname{argmax}_{\mathbf{y}} p(\mathbf{y} | \mathbf{x}, \boldsymbol{\theta})$ using brute force
26 enumeration in $O(K^T)$ time, where T is the number of nodes and K is the number of states (locations)
27 per node. While T is often small, (e.g., just 10 body parts in Figure 19.6), K is often very large,
28 since there are millions of possible locations in an image. By using tree-structured graphs, exact
29 inference can be done in $O(TK^2)$ time, as we explain in Section 9.2.1. Furthermore, by exploiting
30 the fact that the discrete states are ordinal, inference time can be further reduced to $O(TK)$, as
31 explained in [Fel+10].

32 Note that by “augmenting” standard deep neural network libraries with a dynamic programming
33 inference “module”, we can represent DPMs as a kind of CNN, as shown in [Gir+15]. The key
34 property is that we can backpropagate gradients through the inference algorithm.

37 19.2.3 Parameter estimation

38 In this section, we discuss how to perform maximum likelihood estimation for CRFs. This is a small
39 extension of the MRF case in Section 4.3.6.1.

42 19.2.3.1 Log-linear potentials

44 In this section we assume the log potential functions are linear in the parameters, i.e.,

$$\frac{46}{46} \psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) = \exp(\boldsymbol{\theta}_c^\top \phi_c(\mathbf{x}, \mathbf{y}_c)) \quad (19.6)$$

1 Hence the log likelihood becomes
2

3

$$\ell(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_n \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}) = \frac{1}{N} \sum_n \left[\sum_c \boldsymbol{\theta}_c^\top \phi_c(\mathbf{y}_n, \mathbf{x}_n) - \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \right] \quad (19.7)$$

4

5 where
6

7

$$Z(\mathbf{x}_n; \boldsymbol{\theta}) = \sum_{\mathbf{y}} \exp(\boldsymbol{\theta}^\top \phi(\mathbf{y}, \mathbf{x}_n)) \quad (19.8)$$

8

9 is the partition function for example n .
10

11 We know from Section 2.5.3 that the derivative of the log partition function yields the expected
12 sufficient statistics, so the gradient of the log likelihood can be written as follows:
13

14

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N} \sum_n \left[\phi_c(\mathbf{y}_n, \mathbf{x}_n) - \frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \right] \quad (19.9)$$

15

16

$$= \frac{1}{N} \sum_n [\phi_c(\mathbf{y}_n, \mathbf{x}_n) - \mathbb{E}[\phi_c(\mathbf{y}, \mathbf{x}_n)]] \quad (19.10)$$

17

18 Since the objective is convex, we can use a variety of solvers to find the MLE, such as the
19 stochastic meta descent method of [Vis+06], which is a variant of SGD where the stepsize is adapted
20 automatically.
21

22 19.2.3.2 General case

23

24 In the general case, a CRF can be written as follows:

25

$$p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta}) = \frac{\exp(f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}))}{Z(\mathbf{x}; \boldsymbol{\theta})} = \frac{\exp(f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}))}{\sum_{\mathbf{y}'} \exp(f(\mathbf{x}, \mathbf{y}'; \boldsymbol{\theta}))} \quad (19.11)$$

26

27 where $f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta})$ is a scoring (negative energy) function, where high scores correspond to probable
28 configurations. The gradient of the log likelihood is
29

30

$$\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_n, \mathbf{y}_n; \boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \quad (19.12)$$

31

32 Computing derivatives of the log partition function is tractable provided we can compute the
33 corresponding expectations, as we discuss in Section 4.3.6.2. Note, however, that we need to compute
34 these derivatives of for every training example, which is slower than the MRF case.
35

36 19.2.4 Other approaches

37

38 Many other approaches to structured prediction have been proposed, going beyond CRFs. Some of
39 these methods are discussed in [Now+14]. More recently, [BYM17] proposed **Structured Prediction**
40 **Energy Networks**, which are a form of energy based model (Chapter 25), where we predict using
41 an optimization procedure, $\hat{\mathbf{y}}(\mathbf{x}) = \operatorname{argmin} \mathcal{E}(\mathbf{x}, \mathbf{y})$. In addition, it is common to use graph neural
42 networks (Section 16.3.5) and sequence-to-sequence models such as transformers (Section 16.3.4) for
43 this task.
44

45

19.3 Time series forecasting

In this section, we discuss **time series forecasting**, which is the problem of computing the predictive distribution over future observations $h \geq 1$ steps into the future, given the data up until the present, i.e., computing $p(\mathbf{y}_{t+h}|\mathbf{y}_{1:t})$. (The model may optionally be conditioned on known inputs, to get $p(\mathbf{y}_{t+h}|\mathbf{y}_{1:t}, \mathbf{x}_{1:t+h})$.) There are many approaches to this problem (see e.g., [HA21]). In the sections below, we summarize a few of the more popular methods.

But first, a note on notation. We assume \mathbf{y}_t are the outputs at time t , which we assume are observed up to the present but need to be predicted for the future; \mathbf{x}_t are optional observed inputs or covariates which are always observed; and \mathbf{z}_t are optional hidden variables which are never observed.

19.3.1 Structural time series models

In this section, we discuss a particular approach to time series forecasting known as the **structural time series (STS)** approach. The basic idea is to represent the observed data as a sum of C individual **components**:

$$f(t) = f_1(t) + f_2(t) + \cdots + f_C(t) + \epsilon_t \quad (19.13)$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. For example, we might have a seasonal component that causes the observed values to oscillate up and down, and a growth component, that causes the observed values to get larger over time. Each latent process $f_c(t)$ is modeled by a linear Gaussian state-space model (**SSM**, Section 31.1), which (in this context) is also called a **dynamic linear model (DLM)**. Since these are linear, we can combine them altogether into a single LG-SSM. In particular, in the case of scalar observations, the model has the form

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_t|\mathbf{A}\mathbf{z}_{t-1}, \mathbf{Q}) \quad (19.14)$$

$$p(y_t|\mathbf{z}_t, \boldsymbol{\theta}) = \mathcal{N}(y_t|\mathbf{C}\mathbf{z}_t + \boldsymbol{\beta}^\top \mathbf{x}_t, \sigma_y^2) \quad (19.15)$$

where \mathbf{A} and \mathbf{Q} are block structured matrices, with one block per component. The vector \mathbf{C} then adds up all the relevant pieces from each component to generate the overall mean. Note that the matrices \mathbf{A} and \mathbf{C} are fixed sparse matrices which can be derived from the form of the corresponding components of the model, as we discuss below. So the only model parameters are the variance terms, \mathbf{Q} and σ_y^2 , and the optional regression coefficients $\boldsymbol{\beta}$.¹

Once we have specified the model, we can infer the latent state distribution at each step, $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \boldsymbol{\theta})$, using the Kalman filter (Section 8.4.1). Given the current posterior, we can then “roll forward” in time to forecast future observations h steps ahead by computing $p(\mathbf{y}_{t+h}|\mathbf{y}_{1:t}, \boldsymbol{\theta})$, as in Section 8.4.1.6. To estimate the model parameters, $\boldsymbol{\theta} = (\mathbf{A}, \mathbf{C}, \sigma_y^2, \mathbf{Q})$, we can either use maximum likelihood estimation (see Section 31.2.5), or Bayesian inference (see Section 31.4.2). The latter approach is known as **Bayesian structural time series** or **BSTS** modeling and tends to give more reliable results (since small errors in estimating the noise variances can have a large impact on the forecast quality).

Many classical time series methods such as the **ARMA** (autoregressive moving average) method, can be represented as an STS model (see e.g., [Har90; Sim06; CK07; Fra08; DK12; Sar13; PFW21;

¹ In the statistics community, the notation is often slightly different, and often follow [DK12]. In particular, the dynamics are written as $\boldsymbol{\alpha}_t = \mathbf{T}_t \boldsymbol{\alpha}_{t-1} + \mathbf{c}_t \mathbf{R}_t \boldsymbol{\eta}_t$ and the observations are written as $y_t = \mathbf{Z}_t \boldsymbol{\alpha}_t + \boldsymbol{\beta}^\top \mathbf{x}_t + H_t \epsilon_t$, where $\boldsymbol{\eta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\epsilon_t \sim \mathcal{N}(0, 1)$.

¹ ² ³ ⁴ [Tri21]). However, the STS approach has much more flexibility. For example, we can replace the Gaussian likelihood with other kinds of distributions, and we can replace the linear observation and dynamics models with nonlinear versions.

⁵ ⁶ Below we illustrate some of the basic STS components, and then give some examples. We focus on forecasting individual scalar times series, as opposed to vector valued series, or multiple sets of related series.

⁸

⁹ 19.3.1.1 Local level model

¹⁰ The simplest latent dynamical process is known as the **local level model**. It assumes the observations ¹¹ $y_t \in \mathbb{R}$ are generated by a Gaussian with (latent) mean μ_t , which evolves over time according to a ¹² random walk:

$$\underline{\mu}_t = \mu_t + \epsilon_{y,t} \quad \epsilon_{y,t} \sim \mathcal{N}(0, \sigma_y^2) \quad (19.16)$$

$$\underline{\mu}_t = \mu_{t-1} + \epsilon_{\mu,t}, \quad \epsilon_{\mu,t} \sim \mathcal{N}(0, \sigma_\mu^2) \quad (19.17)$$

¹⁷ Under this model, the latent mean has distribution $\mu_t \sim \mathcal{N}(0, t\sigma_\mu^2)$, so the variance grows with ¹⁸ time. We can also use an autoregressive (AR) process, $\mu_t = \rho\mu_{t-1} + \epsilon_{\mu,t}$, where $|\rho| < 1$. This has the ¹⁹ stationary distribution $\mu_\infty \sim \mathcal{N}(0, \frac{\sigma_\mu^2}{1-\rho^2})$, so the uncertainty grows to a finite asymptote instead of ²⁰ unboundedly.

²¹

²² 19.3.1.2 Local linear model

²³ Many time series exhibit linear trends upwards or downwards, at least locally. We can model this ²⁴ by letting the level μ_t change by an amount δ_{t-1} (representing the slope of the line over an interval ²⁵ $\Delta t = 1$) at each step

$$\underline{\mu}_t = \mu_{t-1} + \delta_{t-1} + \epsilon_{\mu,t} \quad (19.18)$$

²⁸ The slope itself also follows a random walk,

$$\underline{\delta}_t = \delta_{t-1} + \epsilon_{\delta,t} \quad (19.19)$$

³¹ and $\epsilon_{\delta,t} \sim \mathcal{N}(0, \sigma_\delta^2)$. This is called a **local linear trend** model.

³² We can combine these two processes by defining the following dynamics model:

$$\begin{pmatrix} \mu_t \\ \delta_t \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} \mu_{t-1} \\ \delta_{t-1} \end{pmatrix}}_{\mathbf{z}_{t-1}} + \underbrace{\begin{pmatrix} \epsilon_{\mu,t} \\ \epsilon_{\delta,t} \end{pmatrix}}_{\boldsymbol{\epsilon}_t} \quad (19.20)$$

³⁷ For the emission model we have

$$\underline{y}_t = \underbrace{\begin{pmatrix} 1 & 0 \end{pmatrix}}_{\mathbf{C}} \underbrace{\begin{pmatrix} \mu_t \\ \delta_t \end{pmatrix}}_{\mathbf{z}_t} + \epsilon_{y,t} \quad (19.21)$$

⁴² We can also use an autoregressive model for the slope, i.e.,

$$\underline{\delta}_t = D + \rho(\delta_{t-1} - D) + \epsilon_{\delta,t} \quad (19.22)$$

⁴⁵ where D is the long run slope to which δ will revert. This is called a “**semilocal linear trend**” ⁴⁶ model, and is useful for longer term forecasts.

⁴⁷

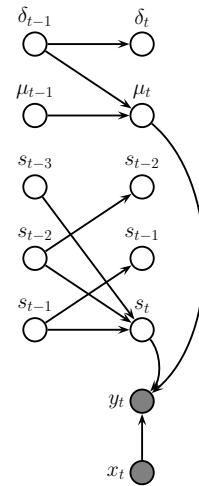
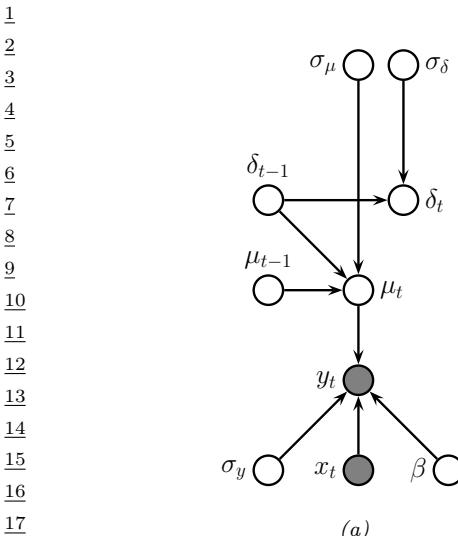


Figure 19.7: (a) A BSTS model with local linear trend and linear regression. The observed output is y_t . The latent state vector is defined by $\mathbf{z}_t = (\mu_t, \delta_t)$. The (static) parameters are $\theta = (\sigma_y, \sigma_\mu, \sigma_\delta, \beta)$. The covariates are \mathbf{x}_t . (b) Adding a latent seasonal process (with $S = 4$ seasons). Parameter nodes are omitted for clarity.

19.3.1.3 Adding covariates

We can also include covariates \mathbf{x}_t into the model, to increase prediction accuracy. If we use a linear model, we have

$$y_t = \mu_t + \beta^\top \mathbf{x}_t + \epsilon_{y,t} \quad (19.23)$$

See Figure 19.7a for an illustration of the local level model with covariates. Note that, when forecasting into the future, we will need some way to predict the input values of future \mathbf{x}_{t+h} ; a simple approach is just to assume future inputs are the same as the present, $\mathbf{x}_{t+h} = \mathbf{x}_t$.

19.3.1.4 Modelling seasonality

Many time series also exhibit **seasonality**, i.e., they fluctuate periodically. This can be modeled by creating a latent process consisting of a series offset terms, s_t , which sums to zero (on average) over a complete cycle of S steps:

$$s_t = - \sum_{k=1}^{S-1} s_{t-k} + \epsilon_{s,t}, \quad \epsilon_{s,t} \sim \mathcal{N}(0, \sigma_s^2) \quad (19.24)$$

For example, for $S = 4$, we have $s_t = -(s_{t-1} + s_{t-2} + s_{t-3})$. We can convert this to a first-order model by stacking the last $S - 1$ seasons into the state vector, as shown in Figure 19.7b.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

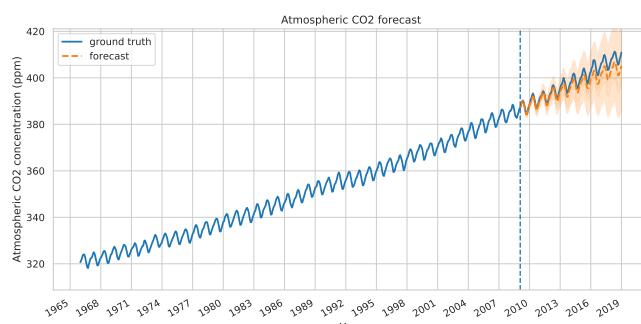


Figure 19.8: CO_2 levels from Mauna Loa. In orange plot we show predictions for the most recent 10 years.
Generated by [STS_TFP.ipynb](#).

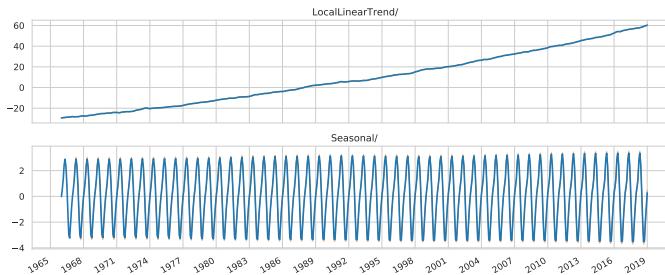


Figure 19.9: Underlying components for the STS mode which was fit to Figure 19.8. Generated by [STS_TFP.ipynb](#).

19.3.1.5 Adding it all up

We can combine the various latent processes (local level, linear trend, and seasonal cycles) into a single linear-Gaussian SSM, because the sparse graph structure can be encoded by sparse matrices. More precisely, the transition model becomes

$$\underbrace{\begin{pmatrix} s_t \\ s_{t-1} \\ s_{t-2} \\ \mu_t \\ \delta_t \end{pmatrix}}_{z_t} = \underbrace{\begin{pmatrix} -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_A \underbrace{\begin{pmatrix} s_{t-1} \\ s_{t-2} \\ s_{t-3} \\ \mu_{t-1} \\ \delta_{t-1} \end{pmatrix}}_{z_{t-1}} + \mathcal{N}(\mathbf{0}, \text{diag}([\sigma_s^2, 0, 0, \sigma_\mu^2, \sigma_\delta^2])) \quad (19.25)$$

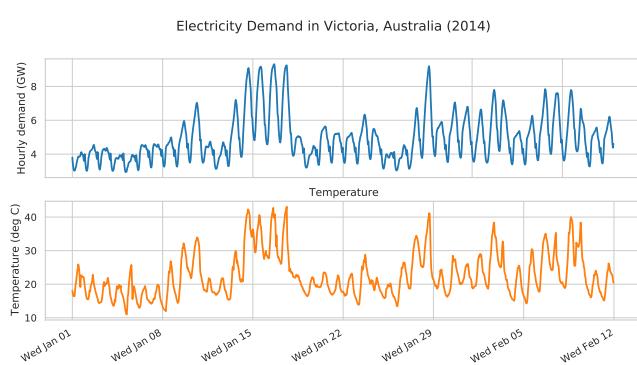


Figure 19.10: Hourly temperature and electricity demand in Victoria, Australia in 2014. Generated by [STS_TFP.ipynb](#).

19.3.1.6 Example: modeling CO₂ levels from Mauna Loa

In this section, we fit an STS model to the monthly atmospheric CO₂ readings from the Mauna Loa observatory in Hawaii.² The data is from January 1966 to February 2019. We combine a local linear trend model with a seasonal model, where we assume the periodicity is $S = 12$, since the data is monthly (see Figure 19.8). We fit the model to all the data except for the last 10 years using variational Bayes. The resulting posterior mean and standard deviations for the parameters are $\sigma_y = 0.169 \pm 0.008$, $\sigma_\mu = 0.159 \pm 0.017$, $\sigma_\delta = 0.009 \pm 0.003$, $\sigma_s = 0.038 \pm 0.008$. We can sample 10 parameter vectors from the posterior and then plug them into it to create a distribution over forecasts. The results are shown in orange in Figure 19.8. Finally, in Figure 19.9, we plot the posterior mean values of the two latent components (linear trend and current seasonal value) over time. We see how the model has successfully decomposed the observed signal into a sum of two simpler signals. (See also Section 19.3.3.1 where we model this data using a GP.)

19.3.1.7 Example: forecasting electricity demand

In this section, we consider a more complex example: forecasting electricity demand in Victoria, Australia, as a function of the previous value and the external temperature. (Remember that January is summer in Australia!) The hourly data from the first six weeks of 2014 is shown in Figure 19.10.³

We fit an STS to this using 4 components: a seasonal hourly effect (period 24), a seasonal daily effect (period 7, with 24 steps per season), a linear regression on the temperature, and an autoregressive term on the observations themselves. We fit the model with variational inference. (This takes about a minute on a GPU.) We then draw 10 posterior samples and show the posterior predictive forecasts in Figure 19.11. We see that the results are reasonable, but there is also considerable uncertainty.

We plot the individual components in Figure 19.12. Note that they have different vertical scales, reflecting their relative importance. We see that the regression on the external temperature is the

⁴⁵ 2. For details, see <https://blog.tensorflow.org/2019/03/structural-time-series-modeling-in.html>.

⁴⁶ 3. The data is from <https://github.com/robjhyndman/fpp2-package>.

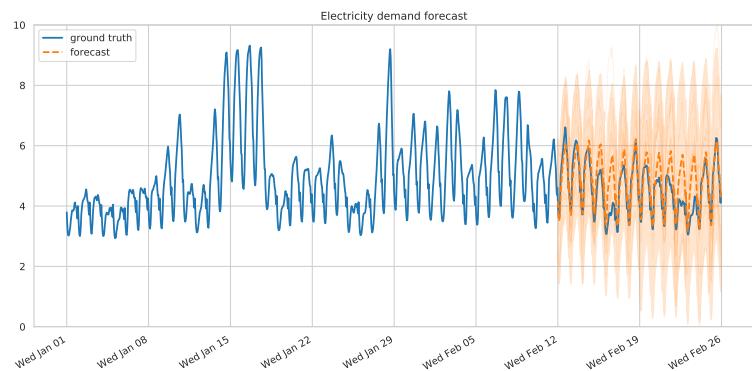


Figure 19.11: Electricity forecasts using an STS. Generated by `STS_TFP.ipynb`.

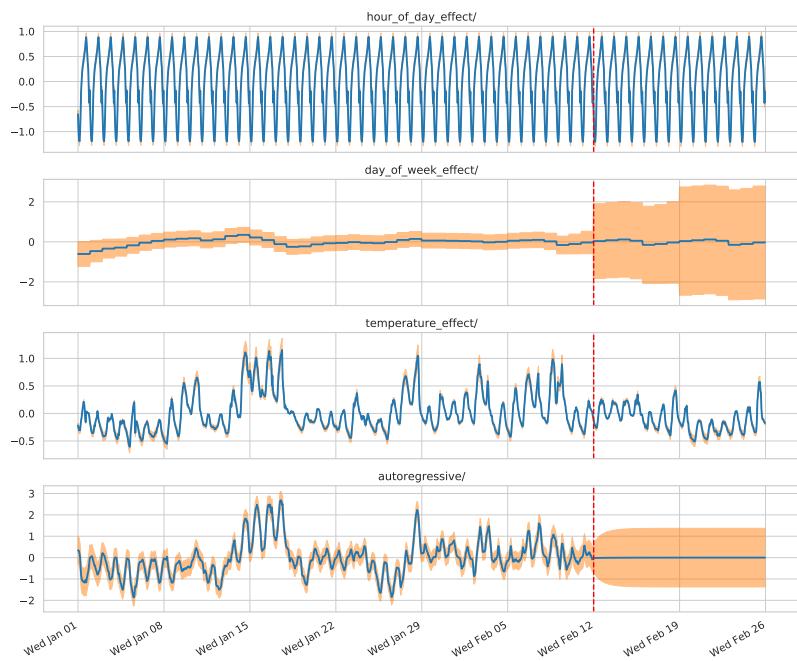


Figure 19.12: Components of the electricity forecasts. Generated by [STS TFP.ipynb](#).

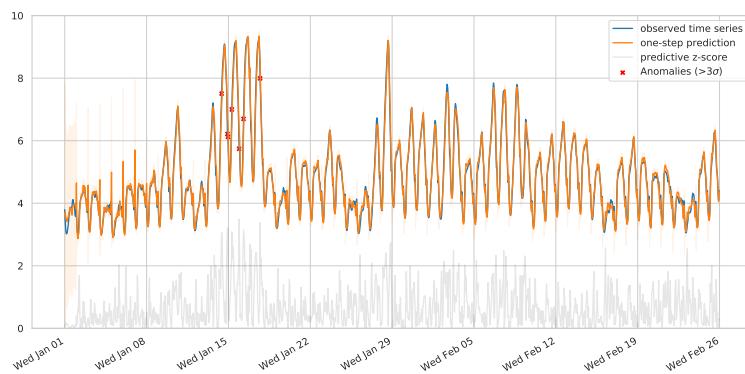


Figure 19.13: We plot the observed electricity data in blue and the predictions in orange. In gray, we plot the z-score at time t , given by $(y_t - \mu_t)/\sigma_t$, where $p(y_t|\mathbf{y}_{1:t-1}, \mathbf{x}_{1:t}) = \mathcal{N}(\mu_t, \sigma_t^2)$. Anomalous observations are defined as points where $z_t > 3$ and are marked with red crosses. Generated by [STS_TFP.ipynb](#).

most important effect. However, the hour of day effect is also quite significant, even after accounting for external temperature. The autoregressive effect is the most uncertain one, since it is responsible for modeling all of the residual variation in the data beyond what is accounted for by the observation noise.

We can also use the model for **anomaly detection**. To do this, we compute the one-step-ahead predictive distributions, $p(y_t|\mathbf{y}_{1:t-1}, \mathbf{x}_{1:t})$, for each timestep t , and then flag all timesteps where the observation is improbable. The results are shown in Figure 19.13.

19.3.2 Prophet

Prophet [TL18a] is a popular time series forecasting library from Facebook. It fits a generalized additive model of the form

$$y(t) = g(t) + s(t) + h(t) + \mathbf{w}^\top \mathbf{x}(t) + \epsilon_t \quad (19.26)$$

where $g(t)$ is a trend function, $s(t)$ is a seasonal fluctuation (modeled using linear regression applied to a sinusoidal basis set), $h(t)$ is an optional set of sparse “holiday effects”, $\mathbf{x}(t)$ are an optional set of (possibly lagged) covariates, \mathbf{w} are the regression coefficients, and $\epsilon(t)$ is the residual noise term, assumed to be iid Gaussian.

Prophet is a regression model, not an auto-regressive model, since it predicts the time series $\mathbf{y}_{1:T}$ given the time stamp t and the covariates $\mathbf{x}_{1:T}$, but without conditioning on past observations of y . To model the dependence on time, the trend function is assumed to be a piecewise linear trend with S changepoints, uniformly spaced in time. (See Section 30.5.2 for a discussion of changepoint detection.) That is, the model has the form

$$g(t) = (k + \mathbf{a}(t)^T \boldsymbol{\delta})t + (m + \mathbf{a}(t)^T \boldsymbol{\gamma}) \quad (19.27)$$

where k is the growth rate, m is the offset, $a_j(t) = \mathbb{I}(t \geq s_j)$, where s_j is the time of the j 'th changepoint, $\delta_t \sim \text{Laplace}(\tau)$ is the magnitude of the change, and $\gamma_j = -s_j \delta_j$ to make the function

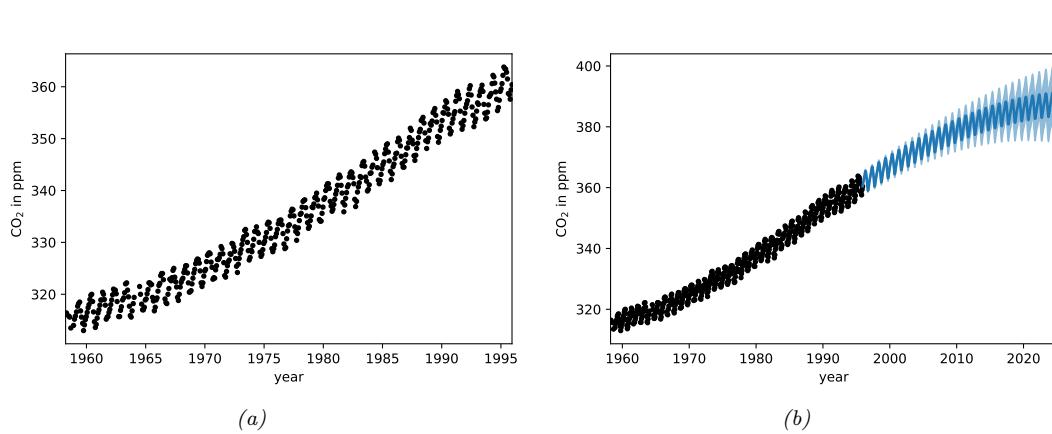


Figure 19.14: (a) The observed Mauna Loa CO₂ time series. (b) Forecasts from a GP. Generated by [gp_mauna_loa.ipynb](#).

continuous. The Laplace prior on δ ensures the MAP parameter estimate is sparse, so the difference across change point boundaries is usually 0.

For an interactive visualization of how prophet works, see <http://prophet.mbrouns.com/>.

19.3.3 Gaussian processes for timeseries forecasting

It is possible to use Gaussian processes (Chapter 18) to perform timeseries forecasting (see e.g., [Rob+13]). The basic idea is to model the unknown output as a function of time, $f(t)$, and to represent a prior about the form of f as a GP; we then update this prior given the observed evidence, and forecast into the future. Naively this would take $O(T^3)$ time. However, for certain stationary kernels, it is possible to reformulate the problem as a linear-Gaussian state space model, and then use the Kalman smoother to perform inference in $O(T)$ time, as explained in [SSH13; SS19; Ada+20]. This conversion can be done exactly for Matern kernels and approximately for Gaussian (RBF) kernels (see [SS19, Ch. 12]). In [SGF21], they describe how to reduce the linear dependence on T to $\log(T)$ time using a **parallel prefix scan** operator, that can be run efficiently on GPUs.

19.3.3.1 Example: Mauna Loa revisited

In this section, we revisit the Mauna Loa CO₂ dataset from Section 19.3.1.6. We show the raw data in Figure 19.14(a). We see that there is periodic (or quasi-periodic) signal with a year-long period superimposed on a long term trend. Following [RW06, Sec 5.4.3], we will model this with a composition of kernels:

$$k(r) = k_1(r) + k_2(r) + k_3(r) + k_4(r) \quad (19.28)$$

where $i k(t, t') = k_i(t - t')$ for the i 'th kernel.

To capture the long term smooth rising trend, we let k_1 be a squared exponential (SE) kernel,

where θ_0 is the amplitude and θ_1 is the length scale:

$$k_1(r) = \theta_0^2 \exp\left(-\frac{r^2}{2\theta_1^2}\right) \quad (19.29)$$

To model the periodicity, we can use periodic or exp-sine-squared kernel from Equation (18.17) with a period of 1 year. However, since it is not clear if the seasonal trend is exactly periodic, we multiply this periodic kernel with another SE kernel to allow for a decay away from periodicity; the result is k_2 , where θ_2 is the magnitude, θ_3 is the decay time for the periodic component, $\theta_4 = 1$ is the period, and θ_5 is the smoothness of the periodic component.

$$k_2(r) = \theta_2^2 \exp\left(-\frac{r^2}{2\theta_3^2} - \theta_5 \sin^2\left(\frac{\pi r}{\theta_4}\right)\right) \quad (19.30)$$

To model the (small) medium term irregularities, we use a rational quadratic kernel (Equation (18.19)):

$$k_3(r) = \theta_6^2 \left[1 + \frac{r^2}{2\theta_7^2\theta_8}\right]^{-\theta_8} \quad (19.31)$$

where θ_6 is the magnitude, θ_7 is the typical length scale, and θ_8 is the shape parameter.

The magnitude of the independent noise can be incorporated into the observation noise of the likelihood function. For the correlated noise, we use another SE kernel:

$$k_4(r) = \theta_9^2 \exp\left(-\frac{r^2}{2\theta_{10}^2}\right) \quad (19.32)$$

where θ_9 is the magnitude of the correlated noise, and θ_{10} is length scale. (Note that the combination of k_1 and k_4 is non-identifiable, but this does not affect predictions.)

We can fit this model by optimizing the marginal likelihood wrt $\boldsymbol{\theta}$ (see Section 18.6.1). The resulting forecast is shown in Figure 19.14(b).

19.3.4 Neural forecasting methods

Classical time series methods work well when there is little data (e.g., short sequences, or few covariates). However, in some cases, we have a lot of data. For example, we might have a single, but very long sequence, such as in anomaly detection from real-time sensors [Ahm+17]. Or we may have multiple, related sequences, such as sales of related products [Sal+19b]. In both cases, larger data means we can afford to fit more complex parametric models. Neural networks are a natural choice, because of their flexibility. Until recently, their performance in forecasting tasks was not competitive with classical methods, but this has recently started to change, as described in [Ben+20; LZ20].

A common benchmark in the univariate time series forecasting literature is the **M4 forecasting competition** [MSA18], which requires participants to make forecasts on many different kinds of (univariate) time series (without covariates). This was recently won by a neural method [Smy20]. More precisely, the winner of the 2019 M4 competition was a *hybrid* RNN-classical method called **ES-RNN** [Smy20]. The exponential smoothing (ES) part allows data-efficient adaptation to the observed past of the current time series; the recurrent neural network (RNN) part allows for learning

1 of nonlinear components from multiple related timeseries. (This is known as a **local+global** model,
2 since the ES part is “trained” just on the local timeseries, whereas the RNN is trained on a global
3 dataset of related time series.)
4

5 In [Ran+18] they adopt a different approach for combining RNNs and classical methods, called
6 **DeepSSM**. In particular, they train a single RNN to predict the parameters of a state-space model
7 (see Section 31.1). In more detail, let $\mathbf{x}_{1:T}^n$ represent the n 'th time series, and let $\boldsymbol{\theta}_t^n$ represent the
8 non-stationary parameters of a linear-trend SSM model (see Section 19.3.1). We train an RNN to
9 compute $\boldsymbol{\theta}_t^n = f(\mathbf{c}_{1:T}^n; \phi)$, where ϕ are the RNN parameters shared across all sequences. We can use
10 the predicted parameters to compute the log likelihood of the sequence, $L_n = \log p(\mathbf{x}_{1:T}^n | \mathbf{c}_{1:T}^n, \boldsymbol{\theta}_{1:T}^n)$,
11 using the Kalman filter. These two modules can be combined to allow for end-to-end training of ϕ
12 to maximize $\sum_{n=1}^N L_n$.

13 In [Wan+19c], they propose a different hybrid model known as **Deep Factors**. The idea is to
14 represent each time series (or its latent function, for non-Gaussian data) as a weighted sum of a
15 global time series, coming from a neural model, and a stochastic local model, such as an SSM or GP.
16 The **DeepGLO** (global-local) approach of [SYD19] proposes a related hybrid method, where the
17 global model uses matrix factorization to learn shared factors. This is then combined with temporal
18 convolutional networks.

19 It is also possible to train a purely neural model, without resorting to classical methods. For
20 example, the **N-BEATS** model of [Ore+20] trains a residual network to predict the weights of a set
21 of basis functions, corresponding to a polynomial trend and a periodic signal. The weights for the
22 basis functions are predicted for each window of input using the neural network. Another approach
23 is the **DeepAR** model of [Sal+19b], which fits a single RNN to a large number of time series. The
24 original paper used integer (count) time series, modeled with a negative binomial likelihood function.
25 This is a unimodal distribution, which may not be suitable for all tasks. More flexible forms, such as
26 mixtures of Gaussians, have also been proposed [Muk+18]. A popular alternative is to use **quantile**
27 **regression** [Koe05], in which the model is trained to predict quantiles of the distribution. For
28 example, [Gas+19] proposed **SQF-RNN**, which uses splines to represent the quantile function.
29 They used **CRPS** or **continuous-ranked probability score** as the loss function. This is a proper
30 scoring rule, but is less sensitive to outliers, and is more “distance aware”, than log loss.

31 The above methods all predict a single output (per time step). If there are multiple simultaneous
32 observations, it is best to try to model their interdependencies. In [Sal+19a], they use a (low-rank)
33 **Gaussian copula** for this, and in [Tou+19], they use a **nonparametric copula**.

34 In [Wen+17], they simultaneously predict quantiles for multiple steps ahead using dilated causal
35 convolution (or an RNN). They call their method **MQ-CNN**. In [WT19], they extend this to predict
36 the full quantile function, taking as input the desired quantile level α , rather than prespecifying a
37 fixed set of levels. They also use a copula to learn the dependencies between multiple univariate
38 marginals.

39

40 19.3.5 Causal impact of a time series intervention

41

42 In this section, we discuss how to perform **counterfactual reasoning** about the effect on an
43 intervention given some observational (non experimental) time series data. (We discuss counterfactuals
44 in more detail in Section 4.6.5.) For example, suppose y_t is the click through rate (CTR) of the web
45 page of some company at time t . The company launches an ad campaign at time n , and observes
46 outcomes $\mathbf{y}_{1:n}$ before the intervention and $\mathbf{y}_{n+1:m}$ after the intervention. A natural question to ask
47

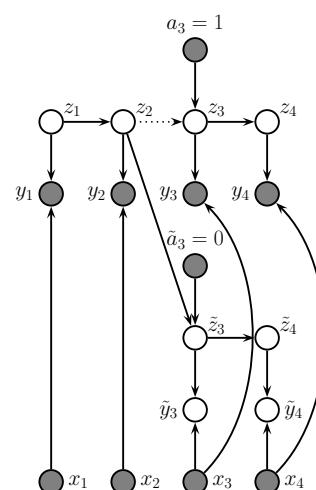


Figure 19.15: Twin network state space model for estimating causal impact of an intervention that occurs just after time step $n = 2$. We have $m = 4$ actual observations, denoted $\mathbf{y}_{1:4}$. We cut the incoming arcs to \mathbf{z}_3 since we assume $\mathbf{z}_{3:T}$ comes from a different distribution, namely the post-intervention distribution. However, in the counterfactual world, shown at the bottom of the figure (with tilde symbols), we assume the distributions are the same as in the past, so information flows along the chain uninterrupted.

is: what would the CTR have been had the company not run the ad campaign?

To answer this question, we will use a structural time series (STS) model (see Section 19.3.1 for details). An STS model is a linear-Gaussian state-space model, where arrows have a natural causal interpretation in terms of the **arrow of time**; thus a STS is a kind of structural equation model, and hence a structural causal model (see Section 4.6). The use of an SCM allows us to infer the latent state of the noise variables given the observed data; we can then “roll back time” to the point of intervention, where we explore an alternative “fork in the road” from the one we actually took by “rolling forward in time” in a new version of the model, using the twin network approach to counterfactual inference (see Section 4.6.5). This approach is known as “**causal impact**” [Bro+15], and was developed by econometricians at Google.

19.3.5.1 Basic idea

To explain the idea in more detail, consider the twin network in Figure 19.15. The intervention occurs after time $n = 2$, and there are $m = 4$ observations in total. We observe 2 data points before the intervention, $\mathbf{y}_{1:2}$, and 2 data points afterwards, $\mathbf{y}_{3:4}$. We assume observations are generated by latent states $\mathbf{z}_{1:4}$, which evolve over time. The states are subject to exogeneous noise terms, which can represent any set of unmodeled factors, such as the state of the economy. In addition, we have exogeneous covariates, $\mathbf{x}_{1:m}$.

To predict what would have happened if we had not performed the intervention, (an event denoted by $\tilde{a} = 0$), we replicate the part of the model that occurs after the intervention, and use it to make forecasts. The goal is to compute the counterfactual distribution, $p(\tilde{\mathbf{y}}_{n+1:m} | \mathbf{y}_{1:n}, \mathbf{x}_{1:m})$, where $\tilde{\mathbf{y}}_t$

1 represents counterfactual outcomes if the action had been $\tilde{a} = 0$. We can compute this counterfactual
2 distribution as follows:

3

$$p(\tilde{\mathbf{y}}_{n+1:m} | \mathbf{y}_{1:n}, \mathbf{x}_{1:m}) = \int p(\tilde{\mathbf{y}}_{n+1:m} | \tilde{\mathbf{z}}_{n+1:m}, \mathbf{x}_{n+1:m}, \boldsymbol{\theta}) p(\tilde{\mathbf{z}}_{n+1:m} | \mathbf{z}_n, \boldsymbol{\theta}) \times \quad (19.33)$$

4

$$p(\mathbf{z}_n, \boldsymbol{\theta} | \mathbf{x}_{1:n}, \mathbf{y}_{1:n}) d\boldsymbol{\theta} d\mathbf{z}_n d\tilde{\mathbf{z}}_{n+1:m} \quad (19.34)$$

5 where

6

$$p(\mathbf{z}_n, \boldsymbol{\theta} | \mathbf{x}_{1:n}, \mathbf{y}_{1:n}) = p(\mathbf{z}_n | \mathbf{x}_{1:n}, \mathbf{y}_{1:n}, \boldsymbol{\theta}) p(\boldsymbol{\theta} | \mathbf{x}_{1:n}, \mathbf{y}_{1:n}) \quad (19.35)$$

7 For linear Gaussian SSMs, the term $p(\mathbf{z}_n | \mathbf{x}_{1:n}, \mathbf{y}_{1:n}, \boldsymbol{\theta})$ can be computed using Kalman smoothing
8 (Section 8.4.1), and the term $p(\boldsymbol{\theta} | \mathbf{y}_{1:n}, \mathbf{x}_{1:n})$, can be computed using MCMC or variational inference.

9 We can use samples from the above posterior predictive distribution to compute a Monte Carlo
10 approximation to the distribution of the **treatment effect** per time step, $\tau_t^i = y_t - \tilde{y}_t^i$, where the
11 i index refers to posterior samples. We can also approximate the distribution of the cumulative
12 causal impact using $\sigma_t^i = \sum_{s=n+1}^t \tau_s^i$. (There will be uncertainty in these quantities arising both from
13 epistemic uncertainty, about the true parameters controlling the model, and aleatoric uncertainty,
14 due to system and observation noise.)

15 The validity of the method is based on 3 assumptions: (1) Predictability: we assume that the
16 outcome can be adequately predicted by our model given the data at hand. (We can check this by
17 using **backcasting**, in which we make predictions on part of the historical data.) (2) Unaffectedness:
18 we assume that the intervention does not change future covariates $\mathbf{x}_{n+1:m}$. (We can potentially check
19 this by running the method with each of the covariates as an outcome variable.) (3) Stability: we
20 assume that, had the intervention not taken place, the model for the outcome in the pre-treatment
21 period would have continued in the post-treatment period. (We can check this by seeing if we predict
22 an effect if the treatment is shifted earlier in time.)

23

24 19.3.5.2 Example: local level model

25 As a concrete example, let us assume we have a local level model and we use linear regression to
26 model the dependence on the covariates, as in Section 19.3.1.3. That is,

27

$$y_t = \mu_t + \boldsymbol{\beta}^\top \mathbf{x}_t + \mathcal{N}(0, \sigma_y^2) \quad (19.36)$$

28

$$\mu_t = \mu_{t-1} + \delta_{t-1} + \mathcal{N}(0, \sigma_\mu^2) \quad (19.37)$$

29

$$\delta_t = \delta_{t-1} + \mathcal{N}(0, \sigma_\delta^2) \quad (19.38)$$

30 See the graphical model in Figure 19.16. The static parameters of the model are $\boldsymbol{\theta} = (\boldsymbol{\beta}, \sigma_y^2, \sigma_\mu^2, \sigma_\delta^2)$,
31 the other terms are state or observation variables. (Note that we are free to use any kind of STS
32 model the local level model is just a simple default.)

33 The use of a linear combination of other “**donor**” time series is similar in spirit to the concept of a
34 “**synthetic control**” [Aba; Shi+21]. However we do not restrict ourselves to a convex combination of
35 donors. Furthermore, when we have many covariates, we can use a spike-and-slab prior (Section 15.2.4)
36 or horseshoe prior (Section 15.2.6) to select the relevant ones.

37 We now give a simple example using synthetic data. We create 3 random sequences of covariates,
38 $\mathbf{x}_t \in \mathbb{R}^3$, and define the observed output to be given by Equation (19.36), with regression weights
39

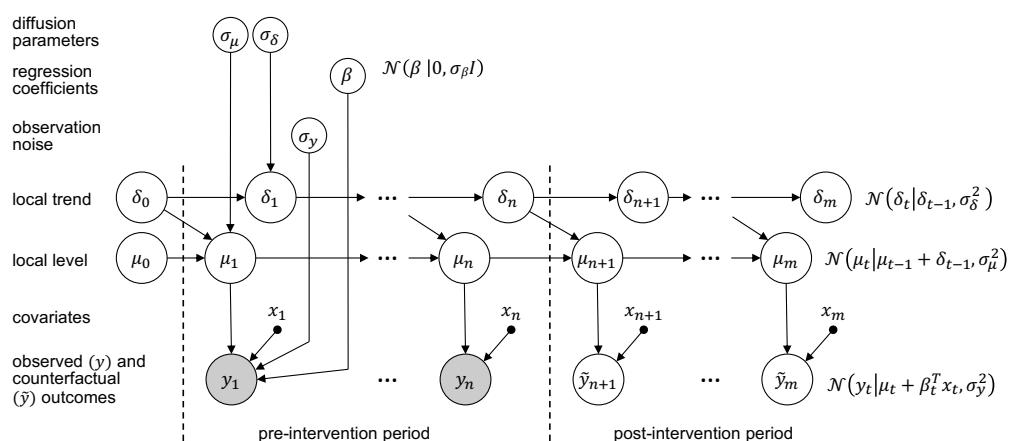
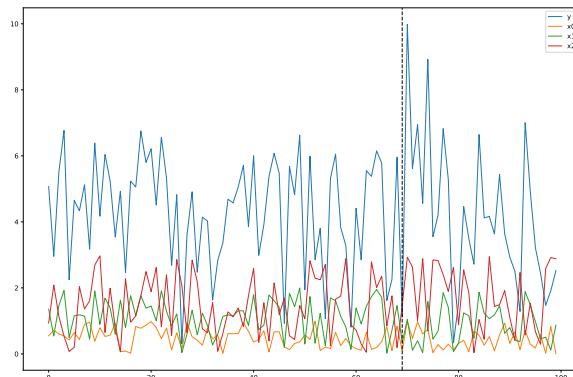


Figure 19.16: A graphical model representation of the local level causal impact model. The dotted line represents the time n at which an intervention occurs. Adapted from Figure 2 of [Bro+15]. Used with kind permission of Kay Brodersen.

$\beta = (2, 3, 0)$. At time step $n = 70$, we perform an artificial intervention by adding Δ_t to y_t , where Δ_t starts off at 5 and drops down to 0 over a period of 10 steps. This simulates the kind of transient lift one often sees when performing marketing campaigns. The data is shown in Figure 19.17(a). We see that there seems to be a small increase in the blue curve y at around time step 70, but it is hard to tell because of the noise.

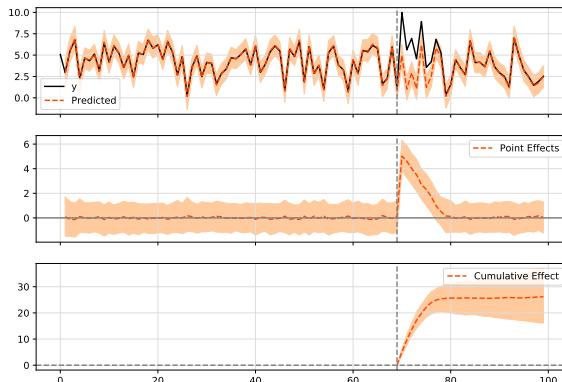
We fit a local level STS model using variational inference, and then make the counterfactual forecast shown in the top row of Figure 19.17(b). Now we see more clearly that, had the process continued without the intervention, the counterfactual outcome would have been smaller. We can therefore estimate the instantaneous and cumulative causal impact, shown in the middle and bottom rows of Figure 19.17(b). Furthermore, we find that the posterior mean estimate for the regression coefficient is $\bar{\beta} = (1.4927632, 1.945029, -0.10319362)$. We see that the third input variable is mostly ignored, as desired.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22



(a)

23
24
25
26
27
28
29
30
31
32
33
34
35
36



(b)

37
38
39
40
41
42
43
44
45
46
47

Figure 19.17: (a) Some simulated time series data which we use to estimate the causal impact of some intervention, which occurs at time $n = 70$, illustrated by the dotted line. The blue curve are the observed outcomes y , the other curves are covariates (inputs). (b) Output of causal inference. Top row: observed vs predicted outcomes. Middle row: Estimate of causal effect τ_t at each time step. Bottom row: Cumulative causal effect, σ_t , up to each time step. Generated by [causal_impact_tfp.ipynb](#).

20 Beyond the iid assumption

20.1 Introduction

The standard approach to supervised ML assumes the training and test sets both contain iid samples from the same distribution. However, there are many settings in which the test distribution may be different from the training distribution; this is known as **distribution shift**, as we discuss in Section 20.2. There are various techniques we can use to modify the training of the model to make it more robust to distribution shift, as we discuss in Section 20.3. Alternatively, we can wait until runtime, and hope that we can detect and/or adapt to the shifts as they occur, as we discuss in Section 20.4.

Another approach to handling distribution shift is to train using multiple related distributions; we discuss this in Section 20.5. We can also consider cases in which there is a single training distribution, but it changes over time; we discuss this in Section 20.7. Finally, in Section 20.8, we discuss settings in which the test distribution is chosen by an adversary to minimize performance of a pre-trained prediction system.

20.2 Distribution shift

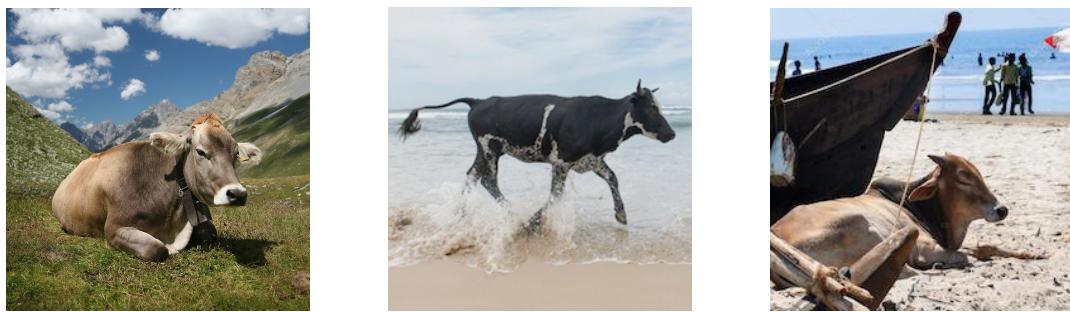
If the test distribution encountered “at run time” is different from the training distribution, we say that there has been a **distribution shift** or **dataset shift** [QC+08]. More formally, let $p_{\text{tr}}(\mathbf{x}, \mathbf{y})$ represent the training or source distribution, and $p_{\text{te}}(\mathbf{x}, \mathbf{y})$ represent the testing or target distribution. Distribution shift refers to the case where $p_{\text{te}} \neq p_{\text{tr}}$. Distributions can differ in many ways, and the nature and degree of shift will have profound impact on the performance of the model on the test distribution (i.e., its **out-of-distribution generalization**, or its **external validity**), as we discuss below.

20.2.1 Motivating examples

Figure 20.1 shows how adding a small amount of Gaussian noise can hurt performance of an otherwise high accuracy image classifier. Similar effects occur with other kinds of common corruption, such as image blurring [HD19]. Analogous problems can also occur in the text domain [Ryc+19]. These examples illustrate that models can be very sensitive to small changes in distribution.

Performance can also drop on “clean” images, but which exhibit “semantic shift”. Figure 20.2 gives an amusing example of this. In particular, it illustrates how the performance of a CNN image classifier can be very accurate on **in-domain** data, but can be very inaccurate on **out-of-domain**

10 *Figure 20.1: Effect of Gaussian noise of increasing magnitude on an image classifier. The model is a*
11 *ResNet-50 CNN trained on ImageNet with added Gaussian noise of standard deviation σ . From Figure 23 of*
12 *[For+19]. Used with kind permission of Justin Gilmer.*



(A) **Cow: 0.99**, Pasture: 0.99,
Grass: 0.99, No Person: 0.98,
Mammal: 0.98
(B) No Person: 0.99, Water: 0.98,
Beach: 0.97, Outdoors: 0.97,
Seashore: 0.97
(C) No Person: 0.97, **Mammal: 0.96**, Water: 0.94, Beach: 0.94, Two:

[c] Figure 20.2: Illustration of how image classifiers generalize poorly to new environments. (a) In the training data, most cows occur on grassy backgrounds. (b-c) In these test image, the cow occurs “out of context”, namely on a beach. In (b), the cow is not detected. In (c), it is classified with a generic “mammal” label. Top five labels and their confidences are produced by ClarifAI.com, which is a state of the art commercial vision system. From Figure 1 of [BVHP18]. Used with kind permission of Sara Beery.

³³ data. Although misclassifying cows may not seem like a big deal, the same problem can plague
³⁴ medical image classification systems (see e.g., [DJL21]).

Analogous problems arise with other kinds of ML models, as well as other data modalities, such as text (e.g., changing “he” to “she” can flip the output of a sentiment analysis system) audio (e.g., adding background noise can easily confuse speech recognition systems), and medical data [Ros22]. Furthermore, the changes to the input needed to change the output can often be imperceptible, as we discuss in the section on adversarial robustness (Section 20.8).

40 The root cause of many of these problems is the fact that discriminative models often leverage
41 features that are predictive of the output *in the training set*, but which are not reliable in general.
42 For example, in an image classification dataset, we may find that green grass in the background
43 is very predictive of the class label “cow”, but this is not a feature that is stable across different
44 distributions; this is called a **spurious correlation**. Unfortunately, such “**shortcut features**” are
45 often easier for models to learn, for reasons explained in [Gei+20a; Xia+21; Sha+20]. We discuss
46 some solutions to this problem later in this chapter.

Name	Source	Target	Causal
Covariate / domain shift	$p_{\text{tr}}(X)p_{\text{tr}}(Y X)$	$p_{\text{te}}(X)p_{\text{tr}}(Y X)$	Causal
Concept shift	$p_{\text{tr}}(X)p_{\text{tr}}(Y X)$	$p_{\text{tr}}(X)p_{\text{te}}(Y X)$	Causal
Label (prior) shift	$p_{\text{tr}}(Y)p_{\text{tr}}(X Y)$	$p_{\text{te}}(Y)p_{\text{tr}}(X Y)$	Anti-causal
Manifestation shift	$p_{\text{tr}}(Y)p_{\text{tr}}(X Y)$	$p_{\text{tr}}(Y)p_{\text{te}}(X Y)$	Anti-causal

Table 20.1: The 4 main types of distribution shift.

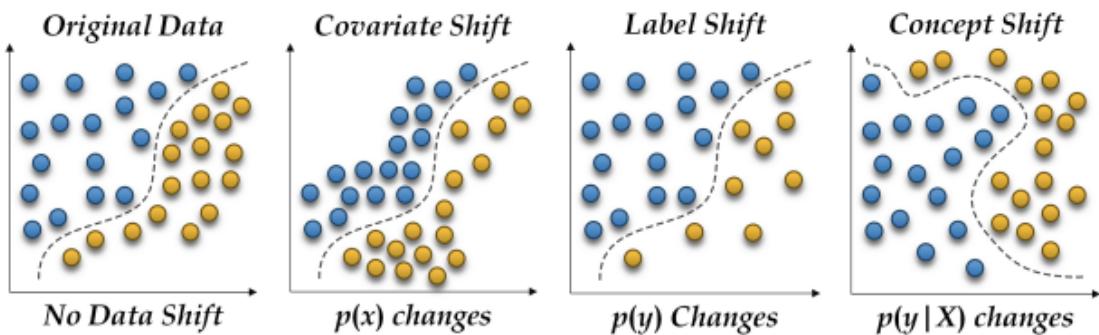


Figure 20.3: Illustration of distribution shift for a 2d binary classification problem. From Figure 1 of [PVP18]. Used with kind permission of Ali Pesaranghader.

20.2.2 A causal view of distribution shift

In the sections below, we briefly summarize some canonical kinds of distribution shift, and some strategies that can be adopted to ameliorate their impact. We adopt a causal view of the problem, as do many papers, e.g., [Sch+12a; Zha+13b; BP16; SS18b; Mei18a; CGW20]).¹ (See Section 4.6 for a brief discussion of causal DAGs, and Chapter 38 for details.)

We assume the inputs to the model (the covariates) are X and the targets to be predicted (the labels) are Y . If we believe that X causes Y , denoted $X \rightarrow Y$, we call it **causal prediction**. If we believe that Y causes X , denoted $Y \rightarrow X$, we call it **anti-causal prediction** [Sch+12a]. The decision about which kind of problem we are dealing with requires understanding the domain, and the nature of the **data generating process**. For example, suppose X is a medical image, and Y is an image segmentation created by a human expert or an algorithm. If we change the image, we will change the annotation, and hence $X \rightarrow Y$. Now suppose X is a medical image and Y is the ground truth disease state of the patient, as estimated by some other means (e.g., a lab test). In this case, we have $Y \rightarrow X$, since changing the disease state will change the appearance of the image. As another example, suppose X is a text review of a movie, and Y is a measure of how informative the review is. Clearly we have $X \rightarrow Y$. Now suppose Y is the star rating of the movie, representing the degree to which the user liked it; this will affect the words that they write, and hence $Y \rightarrow X$.

Based on the above discussion, we can identify 4 main types of distribution shift, as summarized

¹ In the causality literature, the question of whether a model can generalize to a new distribution is called the question of **external validity**. If a model is externally valid, we say that it is **transportable** from one distribution to another [BP16].

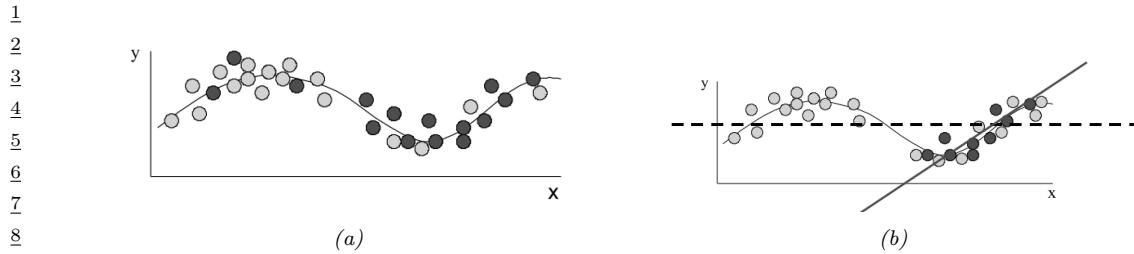


Figure 20.4: (a) Illustration of covariate shift. Light gray represents training distribution, dark gray represents test distribution. We see the test distribution has shifted to the right but the underlying input-output function is constant. (b) Dashed line: fitting a linear model across the full support of X . Solid black line: fitting the same model only on parts of input space that have high likelihood under the test distribution. From Figures 1–2 of [Sto09]. Used with kind permission of Amos Storkey.

in the different causal DAGs in Table 20.1. Note that “manifestation shift”, where $p_{\text{tr}}(Y)p_{\text{tr}}(X|Y)$ changes to $p_{\text{tr}}(Y)p_{\text{te}}(X|Y)$, is not widely considered. This leaves the 3 main types shown in Figure 20.3. See the sections below for more details.

20

20.2.3 Covariate shift

In this section, we consider **covariate shift**, which refers to settings in which we model the joint as $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$, and we assume $p(\mathbf{x})$ changes while $p(\mathbf{y}|\mathbf{x})$ is constant. For example, consider the 1d regression problem in Figure 20.4a: the light colored points represent the source distribution, and the dark colored points represent the target distribution. We see that the target distribution is shifted to the right of the source distribution. An example of covariate shift in the medical imaging context can occur if the model is trained on adults and tested on children, or any other kind of **population shift**.

For a discriminative model of the form $p(\mathbf{y}|\mathbf{x})$, it might seem that such a change in $p(\mathbf{x})$ will not affect the predictions. If the predictor $p(\mathbf{y}|\mathbf{x})$ is the correct model for all parts of the input space \mathbf{x} , then this conclusion is warranted. However, most models will only be accurate in certain parts of the input space, and may “waste” capacity modeling the training distribution, instead of focusing on the test distribution. In such cases, we might want to bias the model so that it is more accurate on the test distribution, as illustrated in Figure 20.4b. We discuss a way to do this in Section 20.3.1.

36

20.2.4 Domain shift

We now discuss a special kind of covariate shift known as **domain shift**, also called **acquisition shift**. Instead of using the model $X \rightarrow Y$, we introduce a latent variable Z which specifies the underlying state of the world (e.g., the shape of an object, or the thought in someone’s head), and we treat X as a noisy measurement of Z . Domain shift refers to the setting in which the sensor model $p(X|Z)$ can change from source to target, even though the distribution over the world states, $p(Z)$, remains the same. The change in $p(Z)$ induces a change in $p(X)$, but we treat this as distinct from covariate shift because the solution techniques can be different (see Section 20.3.2).

Here are some examples of domain shift: the training distribution may be clean images of coffee

pots, and the test distribution may be images of coffee pots with Gaussian noise, as shown in Figure 20.1; or the training distribution may be photos of objects in a catalog, with uncluttered white backgrounds, and the test distribution may be photos of the same kinds of objects collected “in the wild”; or the training data may be synthetically generated images, and the test distribution may be real images. In all of these cases, the distribution over scenes $p(Z)$ is the same, but the appearance of the scenes differs. Similar shifts can occur in the text domain; for example, the training distribution may be movie reviews written in English, and the test distribution may be translations of these reviews into Spanish.

20.2.5 Label / prior shift

In this section, we consider **label shift**, also called **prior shift** or **prevalence shift**, which refers to generative classifiers of the form $p(Y)p(X|Y)$ where the distribution over labels $p(Y)$ changes. For example, consider the medical imaging context, where $Y = 1$ if the patient has some disease and $Y = 0$ otherwise. If the training distribution is an urban hospital and the test distribution is a rural hospital, then the prevalence of the disease, represented by $p(Y = 1)$, might very well be different.

To tackle this, let us assume we fit the generative classifier $p_{\text{tr}}(\mathbf{y})p_{\text{tr}}(\mathbf{x}|\mathbf{y})$ to labeled data from the source, $\mathcal{D}_1^{XY} \sim p_{\text{tr}}^{XY}$. If we have labeled examples from the target distribution, $\mathcal{D}_2^{XY} \sim p_{\text{te}}^{XY}$, we can estimate $p_{\text{te}}(\mathbf{y})$ and then modify our predictor to be

$$p_{\text{te}}(\mathbf{y}|\mathbf{x}) \propto p_{\text{te}}(\mathbf{y})p_{\text{te}}(\mathbf{x}|\mathbf{y}) = p_{\text{te}}(\mathbf{y})p_{\text{tr}}(\mathbf{x}|\mathbf{y}) \quad (20.1)$$

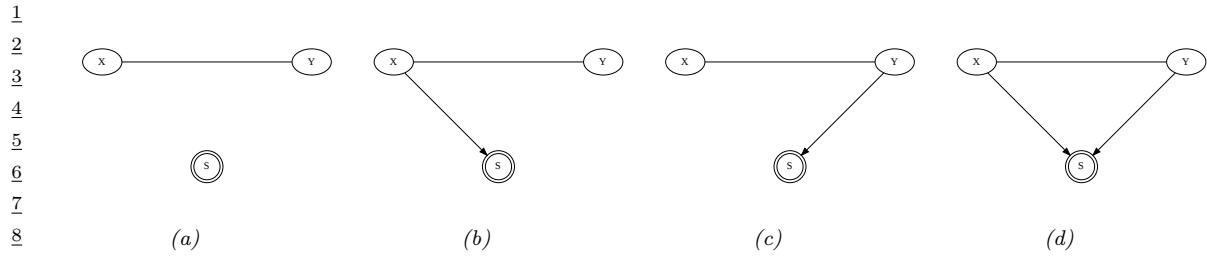
However, if we don’t have labeled target data, we may still be able to estimate $p_{\text{te}}(\mathbf{y})$, as we discuss in Section 20.3.5.

20.2.6 Concept shift

In this section, we consider **concept shift**, also called **annotation shift**, which refers to discriminative classifiers of the form $p(X)p(Y|X)$ where distribution over labels $p(Y|X)$ changes. For example, consider the medical imaging context: the conventions for annotating images might be different between the training distribution and test distribution. Since this is a difference in what we “mean” by a label, it is hard to fix this problem without coming to some shared agreement. Another example of concept shift occurs when a new label can occur in the target distribution that was not part of the source distribution. This is related to open world recognition, discussed in Section 20.4.4.

20.2.7 Manifestation shift

Conditional shift [Zha+13b], also called **manifestation shift** [CWG20], refers to generative models of the form $p(Y)p(X|Y)$ where the distribution over $p(X|Y)$ changes. This is, in some sense, the inverse of concept shift, and is related to domain shift. For example, consider the medical imaging context: the way that the same disease Y manifests itself in the shape of a tumor X might be different between the training distribution and test distribution for various reasons (e.g., different age of the patients).



10 *Figure 20.5: Causal diagrams for different kinds of dataset selection. (The double ringed S node is an*
 11 *indicator variable that is set to S = 1 iff its parents meet some criterion, and only samples where S = 1 are*
 12 *included.) (a) No selection. (b) Selection on X. (c) Selection on Y. (d) Selection on X and Y, which can*
 13 *cause selection bias.*

20.2.8 Selection bias

19 In some cases, we may induce a shift in the distribution just due to the way the data is collected.
 20

21 In particular, let $S = 1$ if a sample from the population is included in the training set, and $S = 0$
 22 otherwise. We may have $p_{\text{tr}}(X, Y) = p(X, Y|S = 1)$ but $p_{\text{te}}(X, Y) = p(X, Y)$.

23 In Figure 20.5 we visualize the four kinds of selection. For example, suppose we select based on X
 24 meeting certain criteria, e.g., images of a certain quality, or exhibiting a certain pattern; this can
 25 induce domain shift or covariate shift. Now suppose we select based on Y meeting certain criteria,
 26 e.g., we are more likely to select rare examples where $Y = 1$, in order to **balance the dataset** (for
 27 reasons of computational efficiency); this can induce label shift. Finally, suppose we select based on
 28 both X and Y ; this can induce non-causal dependencies between X and Y , a phenomenon known as
 29 **selection bias** (see Section 4.2.3.2 for details).

20.3 Training-time techniques for distribution shift

33 Our goal is to train a predictor $\hat{y} = f(\mathbf{x})$ that minimizes the expected loss or risk on the target
 34 distribution:

$$35 \quad \mathcal{L} = \int \int \ell(f(\mathbf{x}), \mathbf{y}) p_{\text{te}}(\mathbf{y}|\mathbf{x}) p_{\text{te}}(\mathbf{x}) d\mathbf{x} d\mathbf{y} \quad (20.2)$$

39 If we have lots of labeled samples from the target distribution, we can use standard empirical risk
 40 minimization. If we have a few labeled samples from the target distribution, we can use a technique
 41 called transfer learning, which we discuss in Section 20.5.1. In this section, we assume we only have
 42 access to *unlabeled* samples from the target distribution at training time. This can enable us to tackle
 43 covariate and domain shift. We can also (with some assumptions) tackle label shift. However, we
 44 cannot tackle concept shift or manifestation shift, since this involves a change in the definition of
 45 what we mean by each label, and this cannot be learned without labeled examples.

46

1

20.3.1 Importance weighting for covariate shift

2
3 In this section, we discuss a common technique for minimizing the impact of covariate shift. Suppose
4 for the moment that we know the target distribution, so we can compute the importance weights
5

6

$$w_n = \frac{p_{\text{te}}(\mathbf{x}_n)}{p_{\text{tr}}(\mathbf{x}_n)} \quad (20.3)$$

7

8
9 Then we can use weighted empirical risk minimization on $\mathcal{D}_{XY}^{\text{train}}$ to approximate the above risk, as
10 proposed by [Shi00a; SKM07]. Thus our goal becomes
11

12

$$\min_f \frac{1}{N} \sum_{n=1}^N w_n \ell(f(\mathbf{x}_n), \mathbf{y}_n) \quad (20.4)$$

13

14
15 where the samples are from the source distribution, $(\mathbf{x}_n, \mathbf{y}_n) \sim p_{\text{tr}}$.

16
17 In most cases we do not know the target distribution. However, assume we have access to an
18 unlabeled dataset from the target distribution, $\mathcal{D}_2^X \sim p_{\text{tr}}^X$. Rather than trying to estimate the density
19 $p_{\text{te}}(\mathbf{x})$, it suffices instead to estimate the density ratio $p_{\text{te}}(\mathbf{x})/p_{\text{tr}}(\mathbf{x})$. We can do this by fitting a
20 binary classifier, as discussed in Section 2.9.5. In particular, suppose we have an equal number of
21 samples from $p_{\text{tr}}(\mathbf{x})$ and $p_{\text{te}}(\mathbf{x})$. Let us label the first set with $c = -1$ and the second set with $c = 1$.
22 Then the probability of this source / target label for a given sample is

23

$$p(c = 1|\mathbf{x}) = \frac{p_{\text{te}}(\mathbf{x})}{p_{\text{te}}(\mathbf{x}) + p_{\text{tr}}(\mathbf{x})} \quad (20.5)$$

24

25
26 and hence $\frac{p(c=1|\mathbf{x})}{p(c=-1|\mathbf{x})} = \frac{p_{\text{te}}(\mathbf{x})}{p_{\text{tr}}(\mathbf{x})}$. If the classifier has the form $f(\mathbf{x}) = p(c = 1|\mathbf{x}) = \sigma(h(\mathbf{x})) =$
27 $\frac{1}{1 + \exp(-h(\mathbf{x}))}$, where $h(\mathbf{x})$ is the prediction function that returns the logits, then the importance
28 weights are given by
29

30

$$w_n = \frac{1/(1 + \exp(-h(\mathbf{x}_n)))}{\exp(-h(\mathbf{x}_n))/(1 + \exp(-h(\mathbf{x}_n)))} = \exp(h(\mathbf{x}_n)) \quad (20.6)$$

31

32
33 Of course this method requires that \mathbf{x} values that may occur in the test distribution should also be
34 possible in the training distribution, i.e. $p_{\text{te}}(\mathbf{x}) > 0 \implies p_{\text{tr}}(\mathbf{x}) > 0$. Hence there are no guarantees
35 about this method being able to interpolate beyond the training distribution.
36

37

20.3.1.1 Conformal prediction with covariate shift

38
39 It is possible to use this importance weighting method for conformal prediction (Section 14.3) in the
40 presence of covariate shift; this is known as **weighted conformal prediction** [Tib+19]. Given a
41 calibration set of N examples, we precompute the importance weights $w_i = \frac{p_{\text{te}}(\mathbf{x}_i)}{p_{\text{tr}}(\mathbf{x}_i)}$, as above. At test
42 time, given new input \mathbf{x}_{N+1} , we compute the weights for $i = 1 : N + 1$:

43

$$p_i^w(\mathbf{x}) = \frac{w(\mathbf{x}_i)}{\sum_{j=1}^N w(\mathbf{x}_j) + w(\mathbf{x})}, \quad (20.7)$$

44

1 We then compute the $1 - \alpha$ quantile of the reweighted distribution of the conformity scores $s_i =$
2 $s(\mathbf{x}_i, y_i)$:
3

4

$$\hat{q}(\mathbf{x}) = \min\{s_j : \sum_{i=1}^j p_i^w(\mathbf{x}) \mathbb{I}(s_i \leq s_j) \geq 1 - \alpha\} \quad (20.8)$$

5

6 Finally, we define the predictive set to be $\mathcal{T}(\mathbf{x}) = \{y : s(\mathbf{x}, y) \leq \hat{q}(\mathbf{x})\}$, as usual.
7

8 If we set $p_i^w(\mathbf{x}) = \frac{1}{N+1}$, we recover the standard method, which selects the largest $\lceil (N+1)(1-\alpha) \rceil$ 'th
9 largest score for \hat{q} . However, by using weighting, we can adapt the width of the prediction interval
10 based on the input \mathbf{x} : If the covariate shift makes easier values of \mathbf{x} more likely, we make the quantile
11 smaller; if the shift makes harder examples more likely, we make the quantile larger.
12

13 Conformal prediction can also be extended to more general kinds of distribution shift. In [Cau+20],
14 they propose a method to adapt the quantile threshold \hat{q} based on the estimated variability of the
15 conformity scores on the calibration set, on the assumption that the test distribution's scores will be
16 reasonably close in distribution (as measured by f -divergence). If the assumptions are met, then the
17 prediction set derived from this “robustified” threshold is guaranteed to have the desired coverage
18 even if the test distribution is different than the calibration distribution.
19

20 20.3.2 Domain adaptation

21 A common approach to minimizing the impact of domain shift is to use a technique called (unsupervised)
22 **domain adaptation** (see e.g., [KL21a] for a review). In this setup, we have a labeled dataset
23 from the source distribution, $\mathcal{D}_1^{XY} \sim p_{tr}^{XY}$, and an unlabeled dataset from the target distribution,
24 $\mathcal{D}_2^X \sim p_{te}^X$. We then fit a model on $\mathcal{D}_1^{XY} \cup \mathcal{D}_2^X$, and then evaluate it on a labeled dataset from the
25 target distribution, $\mathcal{D}_2^{XY} \sim p_2^{XY}$.
26

27 One way to fit a model on the combined data is to train the source classifier in such a way that
28 it cannot distinguish whether the input is coming from the source or target distribution; in this
29 case, it will only be able to use features that are common to both domains. This is called **domain**
30 **adversarial learning** [Gan+16a]. More formally, let $d_n \in \{1, 2\}$ be a label that specifies if the data
31 example n comes from domain 1 or 2. We want to optimize

32

$$\min_{\phi} \max_{\theta} \frac{1}{N_1 + N_2} \sum_{n \in \mathcal{D}_1, \mathcal{D}_2} \ell(d_n, f_{\theta}(\mathbf{x}_n)) + \frac{1}{N_1} \sum_{n \in \mathcal{D}_1} \ell(y_n, g_{\phi}(f_{\theta}(\mathbf{x}_n))) \quad (20.9)$$

33

34 where $N_1 = |\mathcal{D}_1|$, $N_2 = |\mathcal{D}_2|$, $f: \mathcal{X}_1 \cup \mathcal{X}_2 \rightarrow \mathcal{H}$ is a feature extractor defined on the two input domains,
35 and $g: \mathcal{H} \rightarrow \mathcal{Y}$ is a classifier that maps from the feature space \mathcal{H} to the label space. The objective in
36 Equation (20.9) minimizes the loss on the desired task of classifying y , but *maximizes* the loss on
37 the auxiliary task of classifying the domain label d . This can be implemented by the **gradient sign**
38 **reversal** trick, and is related to GANs (Section 27.7.6).
39

40 See [KL21b] for a recent review of other approaches to domain adaptation, and [CB20] for a causal
41 perspective.
42

43 20.3.3 Domain randomization

44 A special case of domain shift occurs when the training set is synthetically generated (e.g., from a
45 computer graphics engine), and the test set is the real world. Such a setting is quite common in
46
47

1 robotics. However, a model which is just trained on synthetic data may work poorly when deployed
 2 in the field due to the **sim2real gap**. [Tob+17] proposed a simple but effective solution to this
 3 known as **domain randomization**. The basic idea is to make the training data be as diverse as
 4 possible, by including random backgrounds, random lighting, etc. This will force the learned model
 5 to be robust to “semantically irrelevant” changes. Then it can treat the idiosyncrasies of the real
 6 world as just another source of noise.
 7

9 20.3.4 Data augmentation

10 In settings in which we don’t have access to samples from the target distribution, we may be able
 11 to simulate such samples by modifying the source data. This is called **data augmentation**, and is
 12 widely used in the deep learning community. For example, it is standard to apply small perturbations
 13 to images (e.g., shifting them or rotating them), while keeping the label the same (assuming that
 14 the label should be invariant to such changes); see e.g., [SK19; Hen+20] for details. Similarly, in
 15 NLP (natural language processing), it is standard to change words that should not affect the label
 16 (e.g., replacing “he” with “she” in a sentiment analysis system), or to use **back translation** (from
 17 a source language to a target language and back) to generate paraphrases; see e.g., [Fen+21] for a
 18 review of such techniques. For a causal perspective on data augmentation, see e.g., [Kau+21].
 19

20 20.3.5 Unsupervised label shift estimation

22 In this section, we describe an approach known as **black box shift estimation**, due to [LWS18],
 23 which can be used to tackle the label shift problem in an unsupervised way. We assume we are using
 24 a generative classifier of the form $p(\mathbf{y}|\mathbf{x}) \propto p(\mathbf{y})p(\mathbf{x}|\mathbf{y})$. We also assume the label shift assumption
 25 holds, so the only thing that changes in the target distribution is the label prior. In other words, if
 26 the source distribution is denoted by $p(\mathbf{x}, \mathbf{y})$ and target distribution is denoted by $q(\mathbf{x}, \mathbf{y})$, we assume
 27 $q(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})q(\mathbf{y})$.

28 First note that, for any deterministic function $f : \mathcal{X} \rightarrow \mathcal{Y}$, we have

$$30 \quad p(\mathbf{x}|y) = q(\mathbf{x}|y) \implies p(f(\mathbf{x})|y) = q(f(\mathbf{x})|y) \implies p(\hat{y}|y) = q(\hat{y}|y) \quad (20.10)$$

31 where $\hat{y} = f(\mathbf{x})$ is the predicted label. Let $\mu_i = q(\hat{y} = i)$ be the empirical fraction of times the model
 32 predicts class i on the test set, and let $q(y = i)$ be the true but unknown label distribution on the test
 33 set. Then we have

$$35 \quad \mu(\hat{y}) = \sum_y q(\hat{y}|y)q(y) = \sum_y p(\hat{y}|y)q(y) = \sum_y p(\hat{y}, y) \frac{q(y)}{p(y)} \quad (20.11)$$

38 We can estimate μ_i from unlabeled test data. We can also estimate the class confusion matrix
 39 $C_{ij} = p(\hat{y} = i|y = j)$ on the labeled training (or validation) set. Hence we can solve $\mathbf{q} = \mathbf{C}^{-1}\boldsymbol{\mu}$,
 40 providing that \mathbf{C} is not singular. Once we know the new label distribution, $q(\mathbf{y})$, we can adjust our
 41 classifier to use $q(\mathbf{y}|\mathbf{x}) \propto q(\mathbf{y})p(\mathbf{x}|\mathbf{y})$.

42 The confusion matrix will be invertible if \mathbf{C} is strongly diagonal, i.e., the model predicts class
 43 y_i correctly more often than any other class y_j . We also require that for every $q(y) > 0$ we have
 44 $p(y) > 0$, which means we see every label at training time. Finally the label shift assumption most
 45 hold. If these three conditions hold, the above approach is a valid estimator. See [LWS18] for the
 46 details.

47

1 **20.3.6 Distributionally robust optimization**

3 We can make a discriminative model that is robust to (some forms of) covariate shift by modifying
4 the importance weighted optimization problem in Equation (20.4) as follows:

$$\min_{f \in \mathcal{F}} \max_{\mathbf{w} \in \mathcal{W}} \frac{1}{N} \sum_{n=1}^N w_n \ell(f(\mathbf{x}_n), \mathbf{y}_n) \quad (20.12)$$

9 where the samples are from the source distribution, $(\mathbf{x}_n, \mathbf{y}_n) \sim p_{\text{tr}}$. This is an example of a **min-max**
10 **optimization problem**, in which we want to minimize the worst case risk. The specification of the
11 robustness set, \mathcal{W} , is a key factor that determines how well the method works, and how difficult the
12 optimization problem is. For details, see e.g., [WYG14; Hu+18]. More generally, we can use methods
13 from **distributionally robust optimization** (see e.g., [CP20a; LFG21]).

14

15 **20.4 Test-time techniques for distribution shift**

16 In general it will not be possible to make a model robust to all of the ways a distribution can shift
17 at test time, nor will we always have access to test samples at training time. As an alternative, it
18 may be sufficient for the model to *detect* that a shift has happened, and then to respond in the
19 appropriate way. There are several ways of detecting distribution shift, some of which we summarize
20 below. (See also Section 30.5.2, where we discuss changepoint detection in time series data.) The
21 main distinction between methods is based on whether we have a set of samples from the target
22 distribution, or just a single sample, and whether the test samples are labeled or unlabeled. We
23 discuss these different scenarios below.

24

25 **20.4.1 Detecting shifts using two-sample testing**

26 Suppose we collect a set of samples from the source and target distribution. We can then use standard
27 techniques for **two-sample testing** to estimate if the null hypothesis, $p_{\text{tr}}(\mathbf{x}, y) = p_{\text{te}}(\mathbf{x}, y)$, is true
28 or not. (If we have unlabeled samples, we just test if $p_{\text{tr}}(\mathbf{x}) = p_{\text{te}}(\mathbf{x})$.) For example, we can use
29 MMD (Section 2.9.3) to measure the distance between the set of samples (see e.g., [Liu+20a]).

30 In some cases it may be possible to just test if the distribution of the labels $p(y)$ has changed, which
31 is an easier problem than testing for changes in the distribution of inputs $p(\mathbf{x})$. In particular, if the
32 label shift assumption (Section 20.2.5) holds (i.e., $p_{\text{te}}(\mathbf{x}|y) = p_{\text{tr}}(\mathbf{x}|y)$), plus some other assumptions,
33 then we can use the blackbox shift estimation technique from Section 20.3.5 to estimate $p_{\text{te}}(y)$. If we
34 find that $p_{\text{te}}(y) = p_{\text{tr}}(y)$, then we can conclude that $p_{\text{te}}(\mathbf{x}, y) = p_{\text{tr}}(\mathbf{x}, y)$. In [RGL19], they showed
35 experimentally that this method worked well for detecting distribution shifts even when the label
36 shift assumption does not hold.

37 It is also possible to use conformal prediction (Section 14.3) to develop “distribution free” methods
38 for detecting covariate shift, given only access to a calibration set and some conformity scoring
39 function [HL20].

40

41 **20.4.2 Detecting single out-of-distribution (OOD) inputs**

42 Now suppose we just have *one* unlabeled sample from the target distribution, $\mathbf{x} \sim p_{\text{te}}$, and we want
43 to know if \mathbf{x} is in-distribution (**ID**) or out-of-distribution (**OOD**). We will call this problem **out-of-**
44

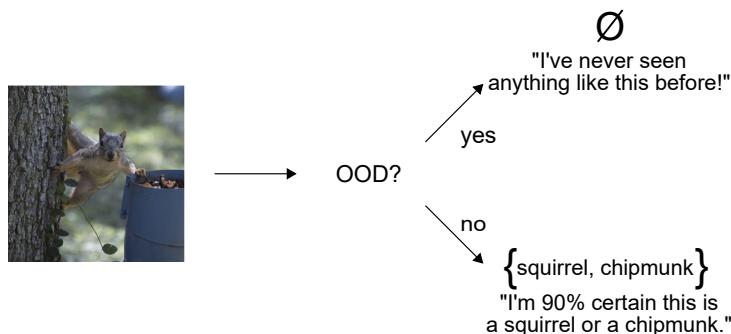


Figure 20.6: Illustration of a two-stage decision problem. First we must decide if the input image is out-of-distribution (OOD) or not. If it is not, we must return the set of class labels that have high probability. From Figure from [AB21]. Used with kind permission of Anastasios Angelopoulos.

distribution detection, although it is also called **anomaly detection**, and **novelty detection**.²

The OOD detection problem requires making a binary decision about whether the test sample is ID or OOD. If it is ID, we may optionally require that we return its class label, as shown in Figure 20.6. In the sections below, we give a brief overview of techniques that have been proposed for tackling this problem, but for more details, see e.g., [Pan+21; Ruf+21; Bul+20; Yan+21; Sal+21; Hen+19b].

20.4.2.1 Supervised ID/OOD methods (outlier exposure)

The simplest method for OOD detection assumes we have access to labeled ID and OOD samples at training time. Then we just fit a binary classifier to distinguish the OOD or background class (called “**known unknowns**”) from the ID class (called “**known knowns**”). This technique is called **outlier exposure** (see e.g., [HMD19; Thu+21; Bit+21]) and can work well. However, in most cases we will not have enough examples of the OOD “class”, since the OOD set is basically the set of all possible inputs except for the ones of interest.

20.4.2.2 Classification confidence methods

Instead of trying to solve the binary ID/OOD classification problem, we can directly try to predict the class of the input. Let the probabilities over the C labels be $p_c = p(y = c | \mathbf{x})$, and let the logits be $\ell_c = \log p_c$. We can derive a **confidence score** or **uncertainty metric** in a variety of ways from these quantities, e.g., the max probability $s = \max_c p_c$, the margin $s = \max_c \ell_c - \max_c^2 \ell_c$ (where \max^2 means the second largest element), the entropy $s = \mathbb{H}(\mathbf{p}_{1:C})$ ³, the “**energy score**” $\sum_c \ell_c$

2. The task of **outlier detection** is somewhat different from anomaly or OOD detection, despite the similar name. In the outlier detection literature, the assumption is that there is a single unlabeled dataset, and the goal is to identify samples which are “untypical” compared to the majority. This is often used for **data cleaning**. (Note that this is a **transductive learning** task, where the model is trained and evaluated on the same data. We focus on inductive tasks, where we train a model on one dataset, and then test it on another.)
3. [Kir+21] argues against using entropy, since it confuses uncertainty about which of the C labels to use with uncertainty about whether any of the labels is suitable, compared to a “none-of-the-above” option.

[Liu+21b], etc. Several sophisticated methods have been proposed for uncertainty quantification, but [Mil+21b; Vaz+22] show that the simple max probability baseline performs very well in practice.

20.4.2.3 Conformal prediction

It is possible to create a method for OOD detection and ID classification that has provably bounded risk using conformal prediction (Section 14.3). The details are in [Ang+21], but we sketch the basic idea here.

We want to solve the two-stage decision problems illustrated in Figure 20.6. We define the prediction set as follows:

$$\mathcal{T}_\lambda(\mathbf{x}) = \begin{cases} \emptyset & \text{if } \text{OOD}(\mathbf{x}) > \lambda_1 \\ \text{APS}(\mathbf{x}) & \text{otherwise} \end{cases} \quad (20.13)$$

where $\text{OOD}(\mathbf{x})$ is some heuristic OOD score, and $\text{APS}(\mathbf{x})$ is the adaptive prediction set method of Section 14.3.1, which returns the set of the top K class labels, such that the sum of their probabilities exceeds threshold λ_2 . (Formally, $\text{APS}(\mathbf{x}) = \{\pi_1, \dots, \pi_K\}$ where π sorts $f(\mathbf{x})_{1:C}$ in descending order, and $K = \min\{K' : \sum_{c=1}^{K'} f(\mathbf{x})_c > \lambda_2\}$.)

We choose the thresholds λ_1 and λ_2 using a calibration set and a frequentist hypothesis testing method (see [Ang+21]). The resulting thresholds will jointly minimize the following risks:

$$R_1(\boldsymbol{\lambda}) = P(\mathcal{T}_\lambda(\mathbf{x}) = \emptyset) \quad (20.14)$$

$$R_2(\boldsymbol{\lambda}) = P(\mathbf{y} \notin \mathcal{T}_\lambda(\mathbf{x}) | \mathcal{T}_\lambda(\mathbf{x}) \neq \emptyset) \quad (20.15)$$

where $P(\mathbf{x}, \mathbf{y})$ is the true but unknown distribution (of ID samples, no OOD samples required), R_1 is the chance that an ID sample will be incorrectly rejected as OOD (type-I error), and R_2 is the chance (conditional on the decision to classify) that the true label is not in the predicted set. The goal is to set λ_1 as large as possible (so we can detect OOD examples when they arise) while controlling the type-I error (e.g., we may want to ensure that we falsely flag (as OOD) no more than 10% of in-distribution samples). We then set λ_2 in the usual way for the APS method in Section 14.3.1.

20.4.2.4 Unsupervised methods

If we don't have labeled examples, a natural approach to OOD detection is to fit an unconditional density model (such as a VAE or AR model) to the ID samples, and then to evaluate the likelihood $p_{\text{tr}}(\mathbf{x})$ and compare this to some threshold value. Unfortunately for many kinds of deep model and datasets, we find that $p_{\text{tr}}(\mathbf{x})$ is lower for samples that are from the training distribution than from a novel test distribution. For example, if we train a pixel-CNN model (Section 23.3.2) or a normalizing-flow model (Chapter 24) on Fashion-MNIST and evaluate it on MNIST, we find it gives higher likelihood to the MNIST samples [Nal+19a; Ren+19; KIW20; ZGR21]. This phenomenon occurs for several other models and datasets (see Figure 20.7).

One solution to this is to use log a **likelihood ratio** relative to a baseline density model, $R(\mathbf{x}) = \log p(\mathbf{x})/q(\mathbf{x})$, as opposed to the raw log likelihood, $L(\mathbf{x}) = \log p(\mathbf{x})$. (This technique was explored in [Ren+19], amongst other papers.) An important advantage of this is that the ratio is invariant to transformations of the data. To see this, let $\mathbf{x}' = \phi(\mathbf{x})$ be some invertible, but possibly nonlinear, transformation. By the change of variables, we have $p(\mathbf{x}') = p(\mathbf{x}) |\det \mathbf{J}_{\mathbf{x}} \phi^{-1}(\mathbf{x})|$. Thus $L(\mathbf{x}')$ will

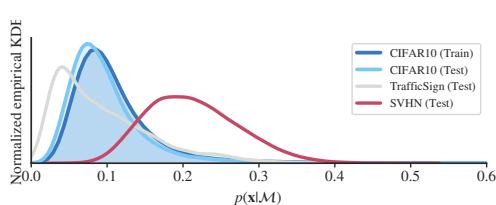


Figure 20.7: Likelihoods from a Glow normalizing flow model (Section 24.2.1) trained on CIFAR10 and evaluated on different test sets. The SVHN street sign dataset has lower visual complexity, and hence higher likelihood. Qualitatively similar results are obtained for other generative models and data set. From Figure 1 of [Ser+20]. Used with kind permission of Joan Serra.

differ from $L(\mathbf{x})$ in a way that depends on the transformation. By contrast, we have $R(\mathbf{x}) = R(\mathbf{x}')$, regardless of ϕ , since

$$R(\mathbf{x}') = \log p(\mathbf{x}') - \log q(\mathbf{x}') = \log p(\mathbf{x}) + \log |\det \mathbf{J}_{\mathbf{x}}\phi^{-1}(\mathbf{x})| - \log q(\mathbf{x}) - \log |\det \mathbf{J}_{\mathbf{x}}\phi^{-1}(\mathbf{x})| \quad (20.16)$$

Various other strategies have been proposed, such as computing the log-likelihood adjusted by a measure of the complexity (coding length computed by a lossless compression algorithm) of the input [Ser+20], computing the likelihood of model features (such as the output probability itself) [Mor+21], etc.

A closely related technique relies on **reconstruction error**. The idea is to fit an autoencoder or VAE (Section 22.2) to the ID samples, and then measure the reconstruction error of the input: a sample that is OOD is likely to incur larger error (see e.g. [Pol+19]). However, this suffers from the same problems as density estimation methods.

An alternative to trying to estimate the likelihood, or reconstruct the output, is to use a GAN (Chapter 27) that is trained to discriminate “real” from “fake” data. This has been extended to the open set recognition setting in the **OpenGAN** method of [KR21b].

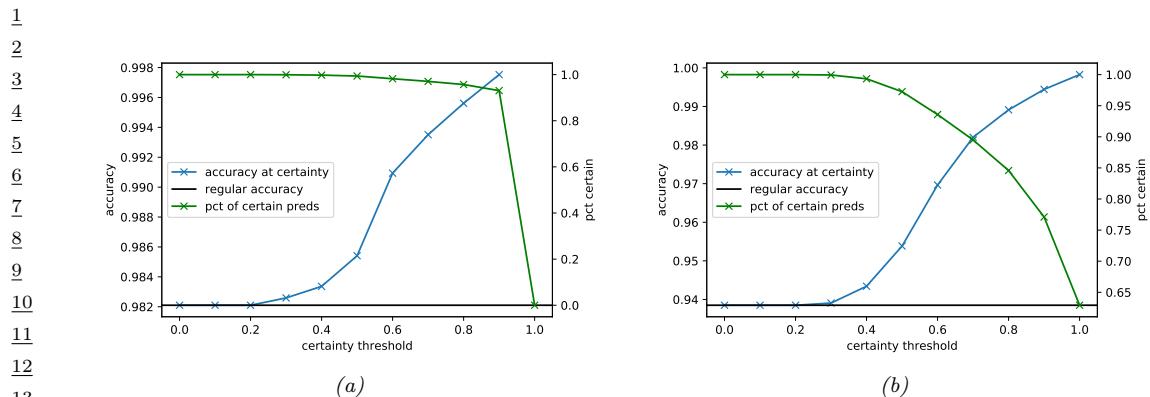
20.4.3 Selective prediction

In this section, we discuss some ways that a classifier can respond at run time if the input distribution has shifted.

Suppose the system has a confidence level of p that an input is OOD (see Section 20.4.4 for a discussion of some ways to compute such confidence scores). If p is below some threshold, the system may choose to **abstain** from classifying it with a specific label. By varying the threshold, we can control the tradeoff between accuracy and abstention rate. This is called **selective prediction**, and is useful for applications where an error can be more costly than asking a human expert for help (e.g., medical image classification).

20.4.3.1 Example: SGLD vs SGD for MLPs

One way to improve performance of OOD detection is to “be Bayesian” about the parameters of the model, so that the uncertainty in their values is reflected in the posterior predictive distribution. This can result in better performance in selective prediction tasks.



¹⁴ Figure 20.8: Accuracy vs confidence plots for an MLP fit to the MNIST training set, and then evaluated
¹⁵ on one batch from the MNIST test set. (a) Plugin approach, computed using SGD. (b) Bayesian approach,
¹⁶ computed using 10 samples from SGLD. Generated by `bnn mnist sgld whitejax.ipynb`.

In this section, we give a simple example of this, where we fit a shallow MLP to the MNIST dataset using either standard SGD (specifically RMSprop) or preconditioned Stochastic Gradient Langevin Dynamics (see Section 12.7.1), which is a form of MCMC inference. We use 5,000 training steps, where each step uses a minibatch of size 1,000. After fitting the model to the training set, we evaluate its predictions on the test set. To assess how well calibrated the model is, we select a subset of predictions whose confidence is above a threshold t . (The confidence value is just the probability assigned to the MAP class.) As we increase the threshold t from 0 to 1, we make predictions on fewer examples, but the accuracy should increase. This is shown in Figure 20.8 (green curve is the fraction of the test set for which we make a prediction, and blue curve is the accuracy). On the left we show SGD, and on the right we show SGLD. We see that SGD assigns nearly all of the predictions a very high confidence, whereas SGLD is more conservative. The prediction accuracy in both models is very high.⁴

In Figure 20.9 we show what happens when we apply these models to OOD data, where the inputs are drawn from the FashionMNIST dataset. We see that SGD remains quite confident in many of its predictions: If we consider a confidence threshold of 0.6, the SGD approach predicts on about 80% of the examples, even though the accuracy is only about 6% on these (and this accuracy is at chance level, due to the completely different set of output labels). The SGLD method is more conservative, and only predicts on about 20% of the examples at this confidence threshold.

³⁷ More details on the behavior of Bayesian neural networks under distribution shift can be found in
³⁸ Section 17.5.7.

41 20.4.4 Open world recognition

⁴² In Section 20.4.3, we discussed methods that “refuse to classify” if the input is suspected to be OOD,
⁴³ i.e., is not one of the existing classes. An alternative approach is to treat the input as an example
⁴⁴

⁴⁵ 4. In this example, SLGD accuracy is slightly worse than SGD, but this can be fixed by suitable tweaking of the
⁴⁶ hyperparameters.

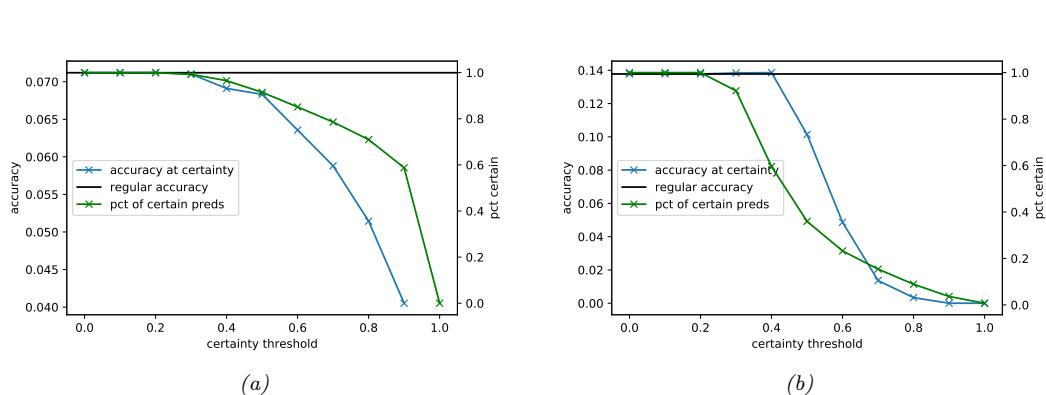


Figure 20.9: Similar to Figure 20.8, except that performance is evaluated on the Fashion MNIST dataset. Generated by `bnn mnist sqld.ipynb`.

from a new class. If the set of classes is allowed to grow over time in this way, the problem is called **open world classification** [BB15a].⁵ Note that open world classification is most naturally tackled in the context of a continual learning system, which we discuss in Section 20.7.3.

20.4.5 Online adaptation

In some settings, it is possible to continuously update the model parameters. This allows the model to adapt to changes in the input distribution. If the input stream is labeled, we can use continual learning methods, which we discuss in Section 20.7.

If the input stream is unlabeled, we can use any of the unsupervised adaptation techniques from Section 20.3. In [Sun+20] they proposed an approach called “**test-time training**”, in which a self-supervised proxy task is used to create pseudo-labels, which can then be used to adapt the model at run time. In more detail, suppose we create a Y-structured network, where we first perform feature extraction, $\mathbf{x} \rightarrow \mathbf{h}$, and then use \mathbf{h} to predict the output \mathbf{y} and some proxy output \mathbf{r} , such as the angle of rotation of the input image. The rotation angle is known if we use data augmentation. Hence we can apply this technique at test time, even if \mathbf{y} is unknown, and update the $\mathbf{x} \rightarrow \mathbf{h} \rightarrow \mathbf{r}$ part of the network, which influences the prediction for \mathbf{y} via the shared bottleneck (feature layer) \mathbf{h} .

In [ZLF21], they propose a method called “**MEMO**” (marginal entropy minimization with one test point) that can be used for any architecture. The idea is, once again, to apply data augmentation at test time to the input x , to create a set of inputs, $\tilde{x}_1, \dots, \tilde{x}_B$. Now we update the parameters so as to minimize the predictive entropy produced by the averaged distribution

$$\bar{p}(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{B} \sum_{b=1}^B p(\mathbf{y}|\tilde{\mathbf{x}}_b, \boldsymbol{\theta}) \quad (20.17)$$

⁵. This is not to be confused with **open set recognition**, which refers to classification problems in which the system should label samples from unknown classes as OOD, rather than trying to incrementally learn about new classes. See e.g., [GHC20] for a review of open set recognition methods.

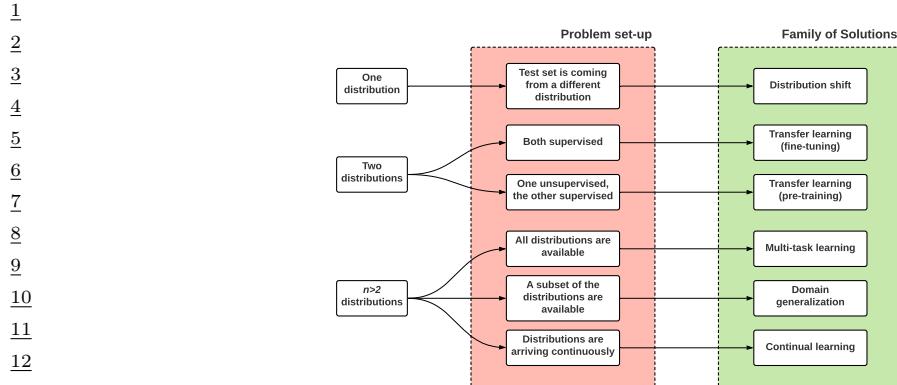


Figure 20.10: Schematic overview of techniques for learning from 1 or more different distributions. Adapted from slide 3 of [Sca21].

This ensures that the model gives the same predictions for each perturbation of the input, and that the predictions are confident (low entropy).

20.5 Learning from multiple distributions

In Section 20.2, we discussed distribution shift, in which a model is trained on a source distribution, and then evaluated on a distinct target distribution. In this section, we generalize this to a setting in which the model is trained on data from two or more training **source distributions**, before being tested on data from a **target distribution**. Our goal is to minimize the population risk on the test set

$$R(f, p_{\text{te}}) = \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p_{\text{te}}} [\ell(\mathbf{y}, f(\mathbf{x}))] \quad (20.18)$$

which we will do by minimizing some variant of the empirical risk on data \mathcal{D} drawn from the source distributions p_{tr} :

$$R(f, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{D}} \ell(\mathbf{y}_n, f(\mathbf{x}_n)) \quad (20.19)$$

We summarize the different problem set-ups that we consider in Figure 20.10.

20.5.1 Transfer learning

Suppose we have labeled training data from a source distribution, $\mathcal{D}_1 \sim p_1$, and also some labeled data from the target distribution, $\mathcal{D}_2 \sim p_2$, where $p_{\text{te}} = p_2$. Our goal is to minimize the risk on p_{te} , which we can approximate empirically using

$$f_2^* = \operatorname{argmin}_f R(f, \mathcal{D}_2) \quad (20.20)$$

If \mathcal{D}_2 is large enough, we can just ignore \mathcal{D}_1 , and then we have a standard supervised learning problem with a single distribution. However, if \mathcal{D}_2 is small, we might want to somehow use \mathcal{D}_1 to learn a model that works well on p_2 . This is called **transfer learning**, since we hope to “transfer knowledge” from p_1 to p_2 . There are many approaches to transfer learning (see e.g., [Zhu+21] for a review). We briefly mention a few below.

20.5.1.1 Pre-train and fine-tune

The simplest and most widely used approach to transfer learning is the **pre-train and fine-tune** approach. We first fit a model to the source distribution by computing $f_1^* = \operatorname{argmin}_f R(f, \mathcal{D}_1)$, and then we adapt it to work on the target distribution:

$$f_2^* = \operatorname{argmin}_f R(f, \mathcal{D}_2) + \lambda \|f - f_1^*\| \quad (20.21)$$

where $\|f - f_1^*\|$ is some distance between the functions, and $\lambda \geq 0$ controls the degree of regularization.

For example, suppose f_1 is a DNN classifier, and the target distribution has a different label distribution. We define the distance between functions in terms of the Euclidean distance in their parameter vectors. Then we can solve Equation (20.21) by “chopping off the head” from f_1 and replacing it with a new linear layer, to map to the new set of labels, and then just training this final layer. When tackling image classification problems, f_1 is often a large CNN which is pre-trained on ImageNet (with 1000 class labels) or some other large dataset. We can then fine-tune it on a smaller, specialized dataset, such as dogs vs cats or medical images, to create f_2 . Alternatively, f_1 might be a large unsupervised language model (e.g., GPT, Section 23.4.1) to which we add a linear (or nonlinear) classification layer, which we train on the target distribution. Since we assume that we have very few samples from the target distribution, we typically “freeze” most of the parameters of the source model. (This makes an implicit assumption that the features that are useful for the source distribution also work well for the target.)

20.5.2 Few-shot learning

People can learn to predict from very few labeled examples. This is called **few-shot learning** (see e.g., [Lu+20]). If we only have a single example of each class, this is called **one-shot learning**. We can think of the new classes as coming from a new distribution, p_2 , while the existing classes are from p_1 . We usually assume the labeled training set from p_2 is small. The most common ways to tackle FSL are to use transfer learning (Section 20.5.1) and meta learning (Section 20.6). See e.g., [Dum+21] for more details.

20.5.3 Prompt tuning

Recently another approach to FSL has been developed, that leverages large models, such as transformers (Section 23.4), which are trained on massive web datasets, usually in an unsupervised way, and then adapted to a small, task-specific target distribution. The interesting thing about this approach is the parameters of the original model, f_1^* , are not changed; instead, the model is simply “conditioned” on new training data, usually in the form of a text **prompt z** . That is, we compute

$$f_2(\mathbf{x}) = f_1^*(\mathbf{x} \cup \mathbf{z}) \quad (20.22)$$

1 where we optimize \mathbf{z} using $(\mathbf{x}, \mathbf{y}) \sim p_2$ pairs from the new distribution, while keeping f_1^* (which was
 2 trained on p_1) frozen. This approach works because f_1 uses attention (Section 16.2.7) to “look at” all
 3 its inputs, and modifies its output in response so \mathbf{z} can modulate the input-output mapping. See
 4 e.g., [Liu+21a] for a review of such methods.
 5

6

7

8 20.5.4 Zero-shot learning

9
 10 An extreme case of FSL occurs when no labeled examples of the new class are given; this is called
 11 **zero-shot learning**. We cannot solve such problems without additional information, since it is
 12 obviously impossible to predict a new label if the label has not been defined, either by examples or
 13 by some other means.

14 There are several approaches to this problem (see e.g., [Pou+20] for a recent review). One
 15 method assumes that the class label y can be generated by a known mapping m from a set of
 16 attributes, $\mathbf{a} = f(\mathbf{x})$, so $\hat{y}(\mathbf{x}) = m(f(\mathbf{x}))$, where m is fixed. For example, in the context of
 17 animal classification, the attributes might be a list of binary features such as: $a_1=\text{hairy}$, $a_2=\text{four-}$
 18 legged , $a_3=\text{small}$, $a_4=\text{vegetarian}$, $a_5=\text{swims}$, $a_6=\text{has-tail}$, etc. We then define the label “dog” to be
 19 $(a_1 = 1, a_2 = 1, a_3 = ?, a_4 = 0, a_5 = 0, a_6 = 1)$, and the label “cow” to be $(a_1 = 0, a_2 = 1, a_3 = 0, a_4 =$
 20 $1, a_5 = 0, a_6 = 1)$, etc. As long as the attribute predictor f is transportable from p_1 to p_2 , then we
 21 can predict new labels (e.g., “cow”) even if we have never seen them before.

22 An alternative solution is to train an embedder to map the input and label to a shared low-
 23 dimensional embedding space, $\mathbf{e}_x = f_x(\mathbf{x})$, $\mathbf{e}_y = f_y(y)$, and then predict the label using $\hat{y}(\mathbf{x}) =$
 24 $\text{argmax } S(\mathbf{e}_x, \mathbf{e}_y)$ using some similarity metric (e.g., cosine distance). We train the embedding
 25 functions f_x, f_y on data from p_1 . If this training distribution is diverse enough, then novel labels
 26 from p_2 can still be embedded reliably, and thus the similarity computation is transportable. This is
 27 the approach to ZSL used by CLIP (Section 34.1).

28 In the NLP literature, a common approach to zero-shot learning is to just define a task as a prompt
 29 (see Section 20.5.3) and then rely on the fact that empirically, a large pre-trained language model
 30 will often do the right thing (see e.g., [Wei+22]).

31

32

33 20.5.5 Multi-task learning

34

35 In **multi-task learning** [Car97], we have labeled data from T different distributions, $\mathcal{D}^t = \{(\mathbf{x}_n^t, \mathbf{y}_n^t) : n = 1 : N_t\}$, and the goal is to learn a model that predicts well on all T of them simultaneously:

36

$$37 \\ 38 \quad f^* = \underset{f}{\operatorname{argmin}} \mathbb{E}_{(\mathbf{x}, \mathbf{y}_{1:t}) \sim p_{\text{te}}(\mathbf{x}, \mathbf{y})} \left[\sum_{t=1}^T \ell_t(\mathbf{y}_t, f_t(\mathbf{x})) \right] \quad (20.23) \\ 39 \\ 40$$

41

42 There are many approaches to solving MTL. The simplest is to fit a single model with multiple
 43 “output heads”, as illustrated in Figure 20.11. The main challenge is defining a suitable architecture
 44 (i.e., deciding which parts of the feature extractor to share across tasks), and how to balance the
 45 different losses from each task. See [BLS11] for a theoretical analysis of this problem, and [ZY21] for
 46 a more detailed review of neural approaches.

47

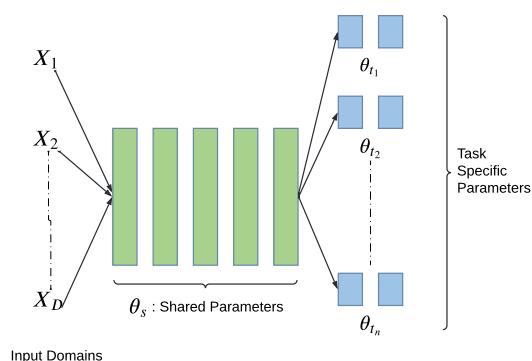


Figure 20.11: Illustration of multi-headed network for multi-task learning.

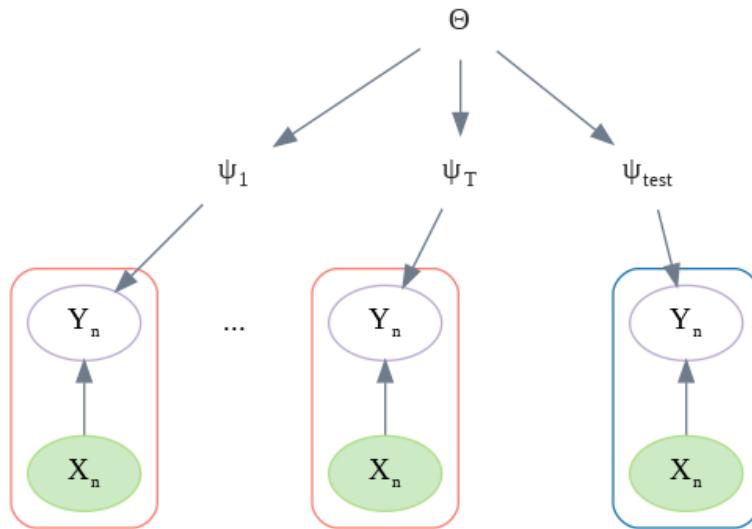


Figure 20.12: Hierarchical Bayesian model for learning from T domains, and then testing on a new distribution. We assume all the parameters of each distribution ϕ_t come from a common prior $p(\psi_t|\Theta)$. which lets us share information between them.

1 **20.5.6 Domain generalization**

3 The problem of **domain generalization** assumes we train on T different labeled source distributions
4 and then test on some unseen labeled target distribution. The source distributions are often
5 called **environments** or **domains**, and are assumed to be related in some way. In some cases the
6 relationship is not specified, so each environment is just identified with a meaningless integer id;
7 in this case, we can think of the set of T distributions as a mixture distribution, and we receive
8 samples from each component, one at a time. In more realistic settings, each different distribution
9 has associated **meta-data** or **context variables** that characterizes the environment in which the
10 data was collected, such as the time, location, imaging device, etc. In both cases, the input to the
11 training algorithm is a set of datasets, $\mathcal{D}_{\text{train}} = \{\mathcal{D}_1, \dots, \mathcal{D}_T\}$, where $\mathcal{D}_t = \{(\mathbf{x}_n^t, \mathbf{y}_n^t) : n = 1 : N_t\}$,
12 where $(\mathbf{x}_n^t, \mathbf{y}_n^t) \sim p_t$ for domain t . The goal is to learn a prediction function that will have low
13 expected loss on some new distribution:
14

15
$$f^* = \underset{f}{\operatorname{argmin}} \mathbb{E}_{\mathcal{D}_{\text{test}} \sim p(\mathcal{D})} [R(f, \mathcal{D}_{\text{test}})] \quad (20.24)$$

16

17

18 where $p(\mathcal{D})$ is a distribution over datasets or environments.

19 Since we don't have access to future test distributions, we need to make some assumptions about
20 how the distributions are related, i.e., we need to learn or model $p(\mathcal{D})$. One approach is to adopt a
21 hierarchical Bayesian approach, in which we assume each p_t has its own local parameters, ψ_t , which
22 are all generated from a common prior with hyper-parameters θ . See Figure 20.12 for an illustration
23 for the overall setup.⁶ Alternatively, we can use invariant risk minimization (Section 20.5.7). Many
24 other techniques have been proposed. Note, however, that [GLP21] found that none of these methods
25 (including IRM) worked consistently better than the baseline approach of performing empirical risk
26 minimization across all the provided datasets. For more information, see e.g., [GLP21; She+21;
27 Wan+21] for reviews of this topic, and [Chr+21] for a causal perspective.
28

29 **20.5.7 Invariant risk minimization**

30 As we discussed in Section 20.2.1, discriminative models are often prone to exploiting "spurious
31 correlations" between the input features and the output label, because these seem to be easier
32 (for current methods) to learn. However, such spurious features are not stable to changes in the
33 distribution. We would like to encourage the model to use stable or causal features, that are invariant
34 across changes in the distribution.
35

36 One approach to this problem is to assume that we have samples from multiple training **envi-**
37 **ronments**, $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_T\}$, where $\mathcal{D}_t = \{(\mathbf{x}_n^t, \mathbf{y}_n^t) \sim p_t : n = 1 : N_t\}$ are samples from the t 'the
38 environment. We want the learned predictor to work well in a new environment. (This is an example
39 of domain generalization, which we discuss in Section 20.5.6.)

40 In [PBM16b], they propose a method called **invariant causal prediction**, that uses hypothesis
41 testing methods to find the set of predictors (features) that directly cause the outcome in each
42 environment, rather than features that are indirect causes, or are just correlated with the outcome.
43 See Figure 20.13 for an illustration.
44

45 6. The difference from the hierarchical model in Figure 17.20a is that here we only care about performance on the new
46 distribution, p_{T+1} , whereas in standard hierarchical modeling, we care about performance on all distributions, $p_{1:T}$.

47

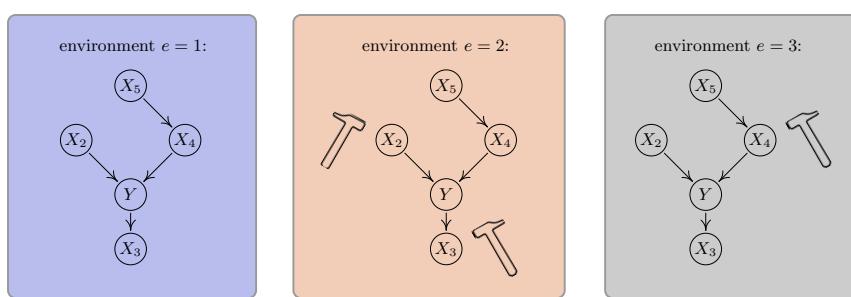


Figure 20.13: Illustration of invariant causal prediction. The hammer symbol represents variables whose distribution is perturbed in the given environment. An invariant predictor must use features $\{X_2, X_4\}$. Considering indirect causes instead of direct ones (e.g. $\{X_2, X_5\}$) or an incomplete set of direct causes (e.g. $\{X_4\}$) may not be sufficient to guarantee invariant prediction. From Figure 1 of [PBM16b]. Used with kind permission of Jonas Peters.

In [Arj+19], they proposed an extension of ICP to handle the case of high dimensional inputs, where the individual variables do not have any causal meaning (e.g., they correspond to pixels). Their approach is called **invariant risk minimization** or **IRM**, and corresponds to finding a predictor that works well on average, across all environments, while also being optimal for each individual environment. That is, we want to find

$$f^* = \operatorname{argmin}_{f \in \mathcal{F}} \sum_{t \in \mathcal{E}} \frac{1}{N_t} \sum_{n=1}^{N_t} \ell(\mathbf{y}_n^t, f(\mathbf{x}_n^t)) \quad (20.25)$$

$$\text{such that } f \in \arg \min_{g \in \mathcal{F}} \frac{1}{N_t} \sum_{n=1}^{N_t} \ell(\mathbf{y}_n^t, g(\mathbf{x}_n^t)) \text{ for all } t \in \mathcal{E} \quad (20.26)$$

The intuition behind this is as follows: there may be many functions that achieve low empirical loss on any given environment, since the problem may be underspecified, but if we pick the one that also works well on all environments, it is more likely to rely on causal features rather than spurious features.

Unfortunately, more recent work has shown that the IRM principle often does not work well for covariate shift, both in theory [RRR21] and practice [GLP21], although it can work well in some anti-causal problems [Ahu+21].

20.6 Meta-learning

The goal of **meta-learning** is to “learn the learning algorithm” [TP97]. A common way to do this is to provide the meta-learner with a set of datasets from different distributions, similar to domain generalization (Section 20.5.6). A general review can be found in [Hos+20]. Our presentation follows the unifying “meta-learning as probabilistic inference for prediction” or **ML-PIP** framework of [Gor+19].

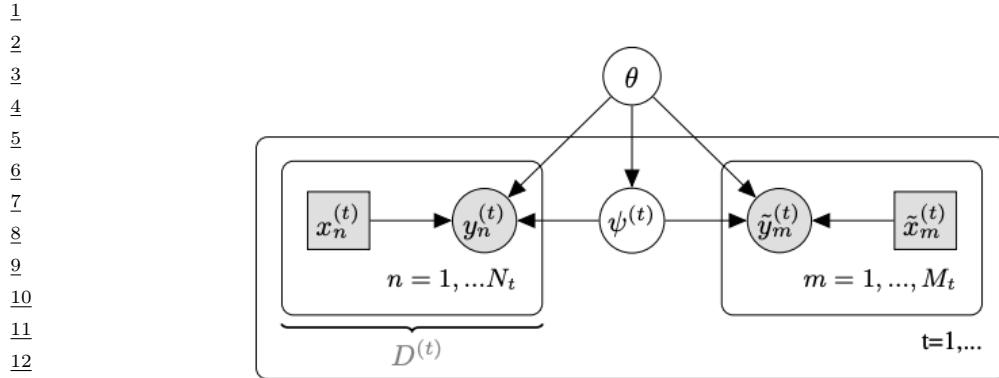


Figure 20.14: Hierarchical Bayesian model for meta-learning. There are T tasks, each of which has a training set $\mathcal{D}^t = \{(\mathbf{x}_n^t, \mathbf{y}_n^t) : n = 1 : N_t\}$ and a test set $\mathcal{D}_{\text{test}}^t = \{(\tilde{\mathbf{x}}_m^t, \tilde{\mathbf{y}}_m^t) : m = 1 : M_t\}$. ψ^t are the task specific parameters, and θ are the shared parameters. From Figure 1 of [Gor+19]. Used with kind permission of Jonathan Gordon.

20.6.1 Meta-learning as probabilistic inference for prediction

We assume there are T tasks (distributions), each of which has a training set $\mathcal{D}^t = \{(\mathbf{x}_n^t, \mathbf{y}_n^t) : n = 1 : N_t\}$ and a test set $\mathcal{D}_{\text{test}}^t = \{(\tilde{\mathbf{x}}_m^t, \tilde{\mathbf{y}}_m^t) : m = 1 : M_t\}$. In addition, ψ^t are the task specific parameters, and θ are the shared parameters. See Figure 20.14. We will learn a point estimate for θ (since it is shared across all datasets, and thus has little uncertainty), but compute an approximate posterior for ψ^t , since each task often has little data. We denote this posterior by $p(\psi^t | \tilde{\mathbf{x}}^t, \mathcal{D}^t, \theta)$. From this, we can compute the posterior predictive distribution for each task:

$$p(\tilde{\mathbf{y}}^t | \tilde{\mathbf{x}}^t, \mathcal{D}^t, \theta) = \int p(\tilde{\mathbf{y}}^t | \tilde{\mathbf{x}}^t, \psi^t) p(\psi^t | \mathcal{D}^t, \theta) d\psi^t \quad (20.27)$$

where θ is estimated based on $\mathcal{D}^{1:T}$.

Since computing the posterior is in general intractable, we will learn an amortized approximation (see Section 10.3.7) denoted $q_\phi(\psi^t | \tilde{\mathbf{x}}^t, \mathcal{D}^t, \theta)$. We choose the parameters of the prior θ and the inference network ϕ so as to maximize the expected accuracy of the *posterior predictive distribution* for any given dataset:

$$\phi^* = \underset{\phi}{\operatorname{argmin}} \mathbb{E}_{p(\mathcal{D}, \tilde{\mathbf{x}})} [D_{\text{KL}}(p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}, \theta) \| q_\phi(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}, \theta))] \quad (20.28)$$

$$= \underset{\phi}{\operatorname{argmin}} \mathbb{E}_{p(\mathcal{D}, \tilde{\mathbf{x}}, \tilde{\mathbf{y}})} \left[\log \int p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \psi) q_\phi(\psi | \mathcal{D}, \theta) \right] \quad (20.29)$$

We can make a Monte Carlo approximation to the outer expectation by sampling T tasks (distributions) from $p(\mathcal{D})$, each of which gets partitioned into a train and test set, $\{(\mathcal{D}^t, \mathcal{D}_{\text{test}}^t) \sim p(\mathcal{D}) : t = 1 : T\}$. We can make an MC approximation to the inner expectation (the integral) by drawing S samples from the task-specific parameter posterior $\psi_s^t \sim q_\phi(\psi^t | \mathcal{D}^t, \theta)$. The resulting objective has the form

(where we assume each test set has M samples for notational simplicity):

$$\mathcal{L}_{\text{ML-PIP}}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \frac{1}{MT} \sum_{m=1}^M \sum_{t=1}^T \log \left(\frac{1}{S} \sum_{s=1}^S p(\tilde{\mathbf{y}}_m^t | \tilde{\mathbf{x}}_m^t, \boldsymbol{\psi}_s^t) \right) \quad (20.30)$$

Note that this is different from standard (amortized) variational inference, that focuses on approximating the expected posterior accuracy *of the parameters* given all of the data for a task, $\mathcal{D}_{\text{all}}^t = \mathcal{D}^t \cup \mathcal{D}_{\text{test}}^t$, rather than focusing on predictive accuracy of a test set given a training set. Indeed, the standard objective has the form

$$\mathcal{L}_{\text{VI}}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \frac{1}{T} \sum_{t=1}^T \left(\sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}_{\text{all}}^t} \left[\frac{1}{S} \sum_{s=1}^S \log p(\tilde{\mathbf{y}}_m^t | \tilde{\mathbf{x}}_m^t, \boldsymbol{\psi}_s^t) \right] - D_{\text{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{\psi}^t | \mathcal{D}_{\text{all}}^t, \boldsymbol{\theta}) \| p(\boldsymbol{\psi}^t | \boldsymbol{\theta})) \right) \quad (20.31)$$

where $\boldsymbol{\psi}_s^t \sim q_{\boldsymbol{\phi}}(\boldsymbol{\psi}^t | \mathcal{D}_{\text{all}}^t)$. We see that the standard formulation takes the average of a log, but the meta-learning formulation takes the log of an average. The latter can give provably better predictive accuracy, as pointed out in [MAD20]. Another difference is that the meta-learning formulation optimizes the forward KL, not reverse KL. Finally, in the meta-learning formulation, we do not need to specify a prior $p(\boldsymbol{\psi}^t | \boldsymbol{\theta})$, instead we just need to specify the form of the posterior $q_{\boldsymbol{\phi}}(\boldsymbol{\psi}^t | \mathcal{D}^t, \boldsymbol{\theta})$.

We now show how this ML-PIP framework includes several common approaches to meta-learning. Most approaches compute a point estimate for the task-specific parameters $q(\boldsymbol{\psi}^t | \mathcal{D}^t, \boldsymbol{\theta}) = \delta(\boldsymbol{\psi}^t - \boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta}))$, where $\boldsymbol{\psi}^*$ is some function that depends on the method.

20.6.2 Gradient-based meta-learning

In **gradient-based meta-learning**, we define the task specific inference procedure as follows:

$$\boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta}) = \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\psi}} \log \sum_{n=1}^{N_t} p(\mathbf{y}_n^t | \mathbf{x}_n^t, \boldsymbol{\psi}) |_{\boldsymbol{\theta}} \quad (20.32)$$

That is, we compute the gradient starting at the prior value of $\boldsymbol{\theta}$, and then take one step in that direction, with step size η . This approach is called **model-agnostic meta-learning** or **MAML** [FAL17]. It is also possible to take multiple gradient steps, by feeding the gradient into an RNN [RL17].

20.6.3 Metric-based few-shot learning

Now suppose $\boldsymbol{\theta}$ correspond to the parameters of a shared neural feature extractor, $h_{\boldsymbol{\theta}}(\mathbf{x})$, and the task specific parameters are the weights and biases of the last linear layer of a classifier, $\boldsymbol{\psi}^t = \{\mathbf{w}_c^t, b_c^t\}_{c=1}^C$. Let us compute the average of the feature vectors for each class in each task:

$$\boldsymbol{\mu}_c^t = \frac{1}{|\mathcal{D}_c^t|} \sum_{\mathbf{x}_n^c \in \mathcal{D}_c^t} h_{\boldsymbol{\theta}}(\mathbf{x}_n^c) \quad (20.33)$$

Now define the task specific inference procedure as follows:

$$\boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta}) = \{\boldsymbol{\mu}_c^t, -\|\boldsymbol{\mu}_c^t\|^2/2\}_{c=1}^C \quad (20.34)$$

1 The predictive distribution becomes
2

$$\frac{3}{4} p(\tilde{y}^t = c | \tilde{\mathbf{x}}^t, \mathcal{D}^t, \boldsymbol{\theta}) \propto \exp(-d(h_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}), \boldsymbol{\mu}_c^t)) = \exp\left(h_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})^T \boldsymbol{\mu}_c^t - \frac{1}{2} \|\boldsymbol{\mu}_c^t\|^2\right) \quad (20.35)$$

5 where $d(\mathbf{u}, \mathbf{v})$ is the Euclidean distance. This is equivalent to the technique known as **prototypical**
6 **networks** [SSZ17].
7

8 20.6.4 VERSA

10 We can also compute a posterior over the parameters of the last layer weights, rather than a point
11 estimate, by using
12

$$\frac{13}{14} q_{\boldsymbol{\phi}}(\boldsymbol{\psi} | \mathcal{D}, \boldsymbol{\theta}) = \prod_{c=1}^C q_{\boldsymbol{\phi}}(\boldsymbol{\psi}_c | \mathcal{D}_c^t, \boldsymbol{\theta}) \quad (20.36)$$

16 where $q_{\boldsymbol{\phi}}(\boldsymbol{\psi}_c | \mathcal{D}_c^t, \boldsymbol{\theta})$ is a network that takes the set of examples of class c in \mathcal{D}^t and returns a
17 distribution over the parameters for class c . This is equivalent to the technique known as **VERSA**
18 [Gor+19].
19

20 20.6.5 Neural processes

21 In the special case that the task-specific inference network computes a point estimate, $q(\boldsymbol{\psi}^t | \mathcal{D}^t, \boldsymbol{\theta}) =$
22 $\delta(\boldsymbol{\psi}^t - \boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta}))$, the posterior predictive distribution becomes
23

$$\frac{24}{25} q(\tilde{y}^t | \tilde{\mathbf{x}}^t, \mathcal{D}^t, \boldsymbol{\theta}) = \int p(\tilde{y}^t | \tilde{\mathbf{x}}^t, \boldsymbol{\psi}^t) q(\boldsymbol{\psi}^t | \mathcal{D}^t, \boldsymbol{\theta}) d\boldsymbol{\psi}^t = p(\tilde{y}^t | \tilde{\mathbf{x}}^t, \boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta}), \boldsymbol{\theta}) \quad (20.37)$$

26 where $\boldsymbol{\psi}^*(\mathcal{D}^t, \boldsymbol{\theta})$ is a function that takes in a set, and returns some parameters. We can directly
27 optimize this by maximum likelihood. This gives rise to a class of methods called **neural processes**
28 [Gar+18e; Gar+18d]. (See [DGF20] for a good tutorial.)
29

30 20.7 Continual learning

31 In this section, we discuss **continual learning** (see e.g., [Had+20; Del+21; Qu+21; LCR21; Mai+22]),
32 also called **life-long learning** (see e.g., [Thr98; CL18]), in which the system learns from a sequence
33 of different distributions, p_1, p_2, \dots . In particular, at each time step t , the model receives a batch of
34 labeled data,
35

$$\frac{37}{38} \mathcal{D}_t = \{(\mathbf{x}_n, \mathbf{y}_n) : n = 1 : N_t, \mathbf{x}_n \sim p_t(\mathbf{x}), \mathbf{y}_n \sim p(\mathbf{y} | f_t(\mathbf{x}_n))\} \quad (20.38)$$

39 where $p_t(\mathbf{x})$ is the unknown input distribution, and $f_t : \mathcal{X}_t \rightarrow \mathcal{Y}_t$ is the unknown prediction function.
40 (We typically assume the input space \mathcal{X}_t is the same at each time step (e.g., $\mathcal{X} = \mathbb{R}^D$), although the
41 support p_t over \mathcal{X} can change.) The learner is then expected to update its belief state about the
42 true function f , and to use it to make predictions on a test set,
43

$$\frac{43}{44} \mathcal{D}_t^{\text{test}} = \{(\mathbf{x}_n, \mathbf{y}_n) : n = 1 : N_t^{\text{test}}, \mathbf{x}_n \sim p_t^{\text{test}}(\mathbf{x}), \mathbf{y}_n \sim p(\mathbf{y} | f_t^{\text{test}}(\mathbf{x}_n))\} \quad (20.39)$$

45 Depending on how we assume $p_t(\mathbf{x})$ and f_t evolve over time, and how the test set is defined, we can
46 create a variety of different CL scenarios, as we discuss below.
47

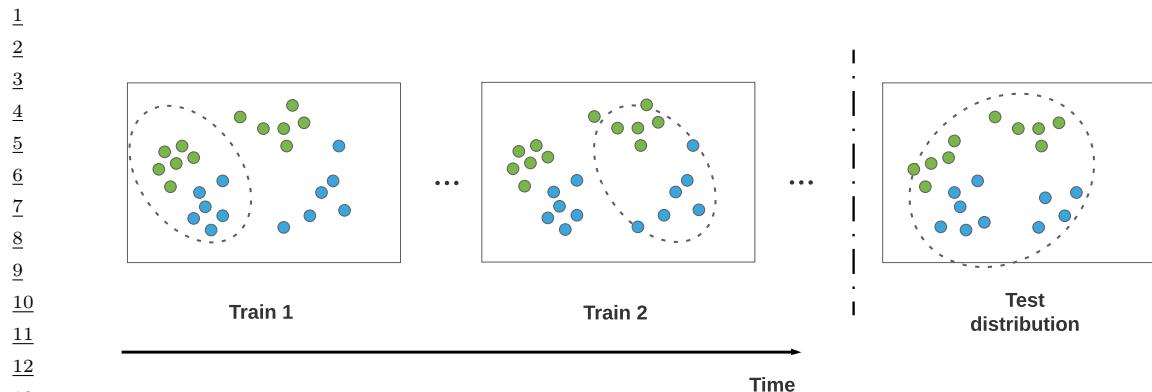


Figure 20.15: An illustration of domain drift.

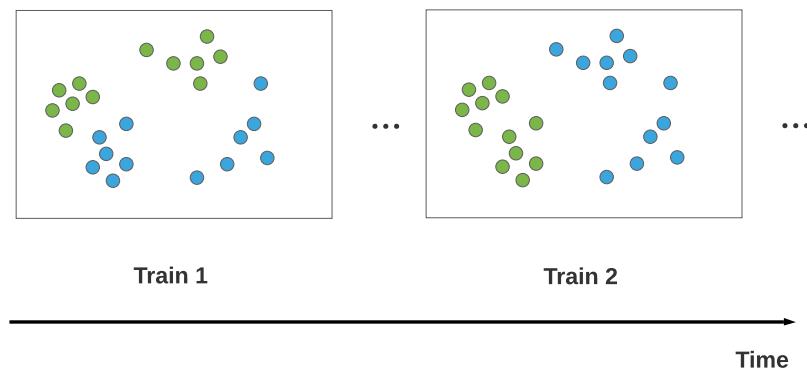


Figure 20.16: An illustration of concept drift.

20.7.1 Domain drift

The problem of **domain drift** refers to the setting in which $p_t(\mathbf{x})$ changes over time (i.e., covariate shift), but the functional mapping $f_t : \mathcal{X} \rightarrow \mathcal{Y}$ is constant. For example, the vision system of a self driving car may have to classify cars vs pedestrians under shifting lighting conditions, so we want our model to perform **online adaptation** of its parameters.

To evaluate such a model, we can define $p_t^{\text{test}}(\mathbf{x})$ to be the current input distribution p_t (e.g., if it is currently night time, we want the detector to work well on dark images), or we can define it to be the union of all the input distributions, $p_t^{\text{test}} = \cup_{t=1}^T p_t$ (e.g., we want the detector to work well on dark and light images). This latter assumption is illustrated in Figure 20.15.

20.7.2 Concept drift

The problem of **concept drift** refers to the setting where the functional mapping $f_t : \mathcal{X} \rightarrow \mathcal{Y}$ changes over time, but the input distribution $p_t(\mathbf{x})$ is constant [WK96]. For example, we can imagine a

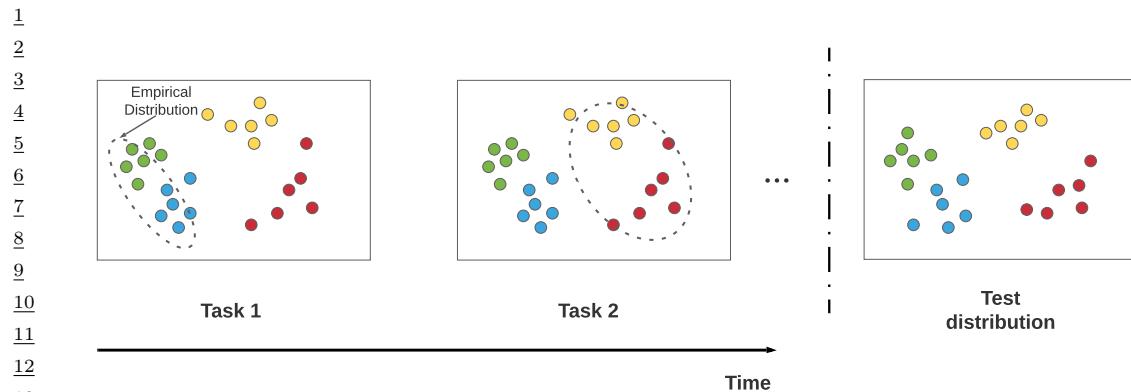


Figure 20.17: An illustration of class incremental learning. Adapted from Figure 1 of [LCR21].

setting in which people engage in certain behaviors, and at step t some of these are classified as illegal, and at step $t' > t$, the definition of what is legal changes, and hence the decision boundary changes. This is illustrated in Figure 20.16.

As another example, we might initially be faced with a sort-by-color task, where red objects go on the left and blue objects on the right, and then a sort-by-shape task, where square objects go on the left and circular objects go on the right.⁷ We can think of this as a problem where $p(y|\mathbf{x}, \text{task})$ is stationary, but the task is unobserved, so $p(y|\mathbf{x})$ changes.

In the concept drift scenario, we see that the prediction for the same underlying input point $\mathbf{x} \in \mathcal{X}$ will change depending on when the prediction is performed. This means that the test distribution also needs to change over time for meaningful identification. Alternatively, we can “tag” each input with the corresponding time stamp or task id.

20.7.3 Task incremental learning

A very widely studied form of continual learning focuses on the setting in which new class labels are “revealed” over time. That is, there is assumed to be a true static prediction function $f : \mathcal{X} \rightarrow \mathcal{Y}$, but at step t , the learner only sees samples from $(\mathcal{X}, \mathcal{Y}_t)$, where $\mathcal{Y}_t \subset \mathcal{Y}$. For example, \mathcal{X} may be the space of images, and \mathcal{Y}_1 might be {cats, dogs}, and \mathcal{Y}_2 might be {cars, bikes, trucks}. Learning to classify with an increasing number of categories is called **class incremental learning** (see e.g., [Mas+20]). This is also called **task incremental learning**, since each distribution is considered as a different **task**, also **data stream classification** (see e.g., [Din+21]). See Figure 20.17 for an illustration.

The problem of class incremental learning has been studied under a variety of different assumptions, as discussed in [Hsu+18; VT18; FG18; Del+21]. The most common scenarios are shown in Figure 20.18. If we assume there are no well defined boundaries between tasks (as often occurs in distribution drift or concept drift), we have **continuous task-agnostic learning** (see e.g., [SKM21; Zen+21]). If there are well defined boundaries (i.e., discontinuous changes of the training distribution), then we

⁷ This example is from Mike Mozer.

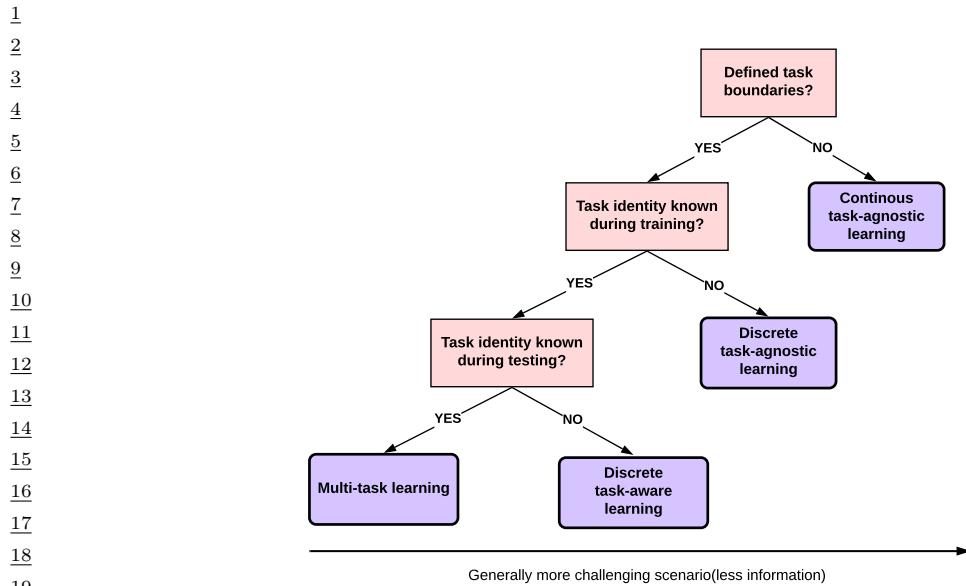


Figure 20.18: Different kinds of incremental learning. Adapted from Figure 1 of [Zen+18].

can distinguish two subcases. If the boundaries are not known during training (similar to detecting distribution shift), we have **discrete task-agnostic learning**. Finally, if the boundaries are given to the training algorithm, we have a **task-aware continual learning** problem.

A common experimental setup in the task-aware setting is to define each task to be a different version of the MNIST dataset, e.g., with all 10 classes present but with the pixels randomly permuted (this is called **permuted MNIST**) or with a subset of 2 classes present at each step (this is called **split MNIST**).⁸ In the task-aware setting, the task label may or may not be known at test time. If it is, the problem is essentially equivalent to multi-task learning (see Section 20.5.5). If it is not, the model must predict the task and corresponding class label within that task (which is a standard supervised problem with a hierarchical label space); this is commonly done by using a **multi-headed DNN**, with CT outputs, where C is the number of classes, and T is the number of tasks.

In the multi-headed approach, the number of “heads” is usually specified as input to the algorithm, because the softmax imposes a sum-to-one constraint that prevents incremental estimation of the output weights in the open-class setting. An alternative approach is to wait until a new class label is encountered for the first time, and then train the model with an enlarged output head. This requires storing past data from each class, as well as data for the new class (see e.g., [PTD20]). Alternatively, we can use generative classifiers where we do not need to worry about “output heads”. If we use a “deep” nearest neighbor classifier, with a shared feature extractor (embedding function), the main challenge is to efficiently update the stored prototypes for past classes as the feature extractor parameters

8. In the split MNIST setup, for task 1, digits (0,1) get labeled as (0,1), but in task 2, digits (2,3) get labeled as (0,1). So the “meaning” of the output label depends on what task we are solving. (It seems odd to “reuse” the same label for perceptually different things, but there are some problems where words can be ambiguous, so one could argue this problem setting is worth studying.)

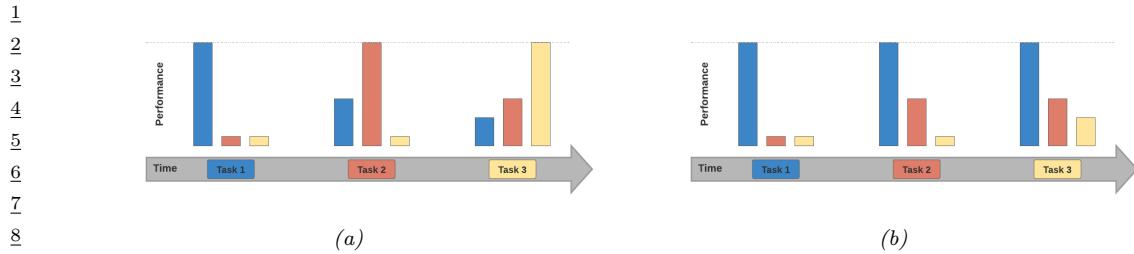


Figure 20.19: Some failure modes in class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) Catastrophic forgetting refers to the phenomenon in which performance on a previous task drops when trained on a new task. (b) Too little plasticity (e.g., due to too much regularization) refers to the phenomenon in which only the first task is learned. Adapted from Figure 2 of [Had+20].

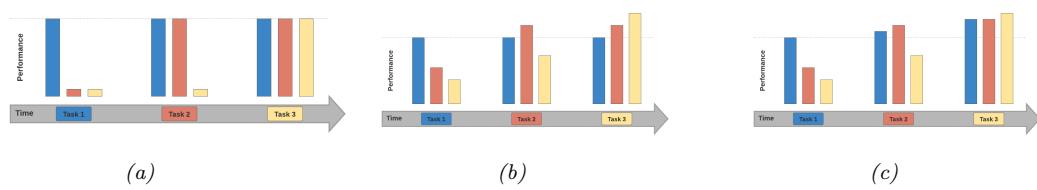


Figure 20.20: What success looks like for class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) No forgetting refers to the phenomenon in which performance on previous tasks does not degrade over time. (b) Forward transfer refers to the phenomenon in which training on past tasks improves performance on future tasks beyond what would have been obtained by training from scratch. (c) Backwards transfer refers to the phenomenon in which training on future tasks improves performance on past tasks beyond what would have been obtained by training from scratch. Adapted from Figure 2 of [Had+20].

change (see e.g., [DLT21]). If we fit a separate generative model per class (e.g., a VAE, as in [VLT21]), online learning becomes easier, but the method may be less sample efficient.

At the time of writing, most of the CL literature focuses on the task-aware setting. However, from a practical point of view, the assumption that task boundaries are provided at training or test time is very unrealistic. For example, consider the problem of training a robot to perform various activities: The data just streams in, and the robot must learn what to do, without anyone telling it that it is now being given an example from a new task or distribution (see e.g., [Fon+21; Woł+21]). Thus future research should focus on the task-agnostic setting, with either discrete or continuous changes.

20.7.4 Catastrophic forgetting

In the class incremental learning literature, it is common to train on a sequence of tasks, but to test (at each step) on all tasks. In this scenario, there are two main possible failure modes. The first possible problem is called “**catastrophic forgetting**” (see e.g., [Rob95b; Fre99; Kir+17]). This refers to the phenomenon in which performance on a previous task drops when trained on a new task (see Figure 20.19(a)). Another possible problem is that only the first task is learned, and

1 the model does not adapt to new tasks (see Figure 20.19(b)).

2 If we avoid these problems, we should expect to see the performance profile in Figure 20.20(a),
3 where performance of incremental training is equal to training on each task separately. However, we
4 might hope to do better by virtue of the fact that we are training on multiple tasks, which are often
5 assumed to be related. In particular, we might hope to see **forward transfer**, in which training on
6 past tasks improves performance on future tasks beyond what would have been obtained by training
7 from scratch (see Figure 20.20(b)). Additionally, we might hope to see **backwards transfer**, in
8 which training on future tasks improves performance on past tasks (see Figure 20.20(c)). We can
9 quantify the degree of transfer as follows, following [LPR17]. If R_{ij} is the performance on task j
10 after it was trained on task i , R_j^{ind} is the performance on task j when trained just on j , and there
11 are T tasks, then the amount of forward transfer is

$$\text{FWT} = \frac{1}{T} \sum_{j=1}^T R_{j,j} - R_j^{\text{ind}} \quad (20.40)$$

17 and the amount of backwards transfer is

$$\text{BWT} = \frac{1}{T} \sum_{j=1}^T R_{T,j} - R_{j,j} \quad (20.41)$$

23 There are many methods that have been devised to overcome the problem of catastrophic forgetting,
24 but we can group them into three main types. The first is **regularization methods**, which add a
25 loss to preserve information that is relevant to old tasks. (For example, online Bayesian inference is
26 of this type, since the posterior for the parameters is derived from the new data and the past prior;
27 see e.g., the **elastic weight consolidation** method discussed in Section 17.6.3, or the **variational**
28 **continual learning** method discussed in Section 10.3.9). The second is **memory methods**, which
29 rely on some kind of **experience replay** or **rehearsal** of past data (see e.g., [Hen+21]), or some
30 kind of generative model of past data. The third is **architectural methods**, that add capacity to
31 the network whenever a task boundary is encountered, such as a new class label (see e.g., [Rus+16]).

32 Of course, these techniques can be combined. For example, we can create a semi-parametric model,
33 in which we store some past data (exemplars) while also learning parameters online in a Bayesian
34 (regularized) way (see e.g., [Kur+20]). The “right” method depends, as usual, on what inductive bias
35 you want to use, and want your computational budget is in terms of time and memory.

37 20.7.5 Online learning

40 The problem of **online learning** refers to the setting in which the input distribution $p_t(\mathbf{x})$ and the
41 target function f_t can change at each step, but where we only evaluate performance on one-step-ahead
42 predictions. That is, at each step, the algorithm obtains an unlabeled sample, $\mathbf{x}_t \sim p_t(\mathbf{x})$, it makes a
43 prediction $\hat{\mathbf{y}}_{t|t-1} = \text{argmax } p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:t-1})$, and then incurs loss $\mathcal{L}_t = \ell(\hat{\mathbf{y}}_{t|t-1}, \mathbf{y}_t)$, where $\mathbf{y}_t = f_t(\mathbf{x}_t)$
44 is the true label. Finally, it updates its belief about the unknown prediction function, $p(\boldsymbol{\theta}_t | \mathcal{D}_{1:t})$. See
45 Figure 20.21 for an illustration. (This setup is also called **prequential inference** [DV99].)

46 In contrast to the continual learning scenarios studied above, the loss incurred at each step is what

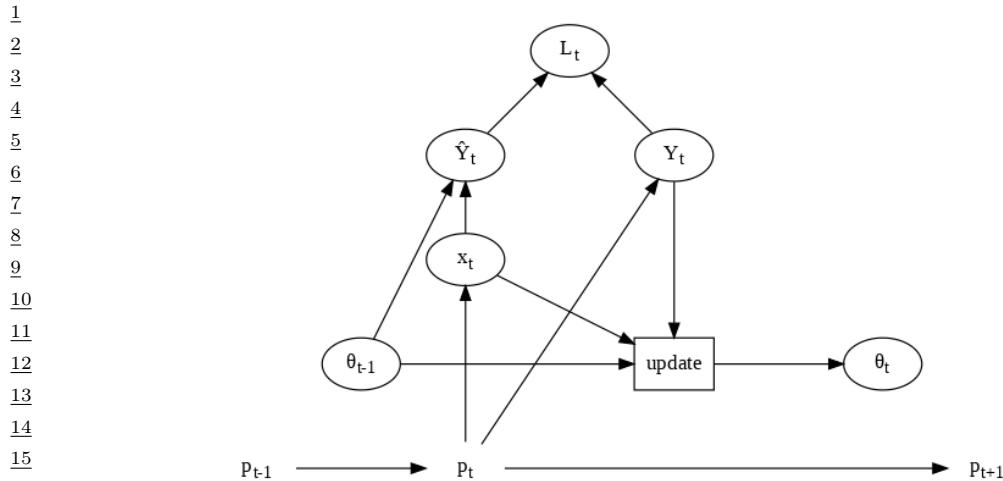


Figure 20.21: Online learning illustrated as a graphical model.

matters, rather than loss on a fixed test set. That is, we want to minimize

$$\mathcal{L} = \sum_{t=1}^T \mathcal{L}_t = \sum_{t=1}^T \ell(\hat{\mathbf{y}}_{t|t-1}, \mathbf{y}_t) \quad (20.42)$$

Since it is hard to interpret this number, it is common to compare it to the optimal value one could have obtained in hindsight. This yields a quantity called the **regret**:

$$\text{regret} = \sum_{t=1}^T [\ell(\hat{\mathbf{y}}_{t:t-1}, \mathbf{y}_t) - \ell(\hat{\mathbf{y}}_{t:T}, \mathbf{y}_t)] \quad (20.43)$$

where $\hat{\mathbf{y}}_{t|t-1} = \operatorname{argmax}_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:t-1})$ is the online prediction, and $\hat{\mathbf{y}}_{t:T} = \operatorname{argmax}_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:T})$ is the optimal estimate at the end of training. It is possible to convert bounds on regret, which are backwards looking, into bounds on risk (i.e., expected future loss), which is forwards looking. See [HT15] for details.

Online learning is very useful for decision and control problems, such as multi-armed bandits (Section 36.4) and reinforcement learning (see Chapter 37), where the agent “lives forever”, and where there is no fixed training phase followed by a test phase. However, it is possible to include the previous continual learning scenarios as special cases of online learning, by defining a suitable sequence of distributions. We also need to distinguish between steps in which we train (i.e., perform an update of our model) and steps in which we just test (with the model held fixed). Furthermore, we allow the training steps and test steps to work with minibatches of samples, instead of individual samples. We call this “**generalized online learning**”.

With this setup, we can capture class incremental learning as follows: at step 1, we train on samples from p_1 , and at step 1', we test on samples from $p_* = p_{1:T}$; at step 2, we train on samples from p_2 , and at step 2', we test on samples from p_* ; etc. This is a special case of generalized online

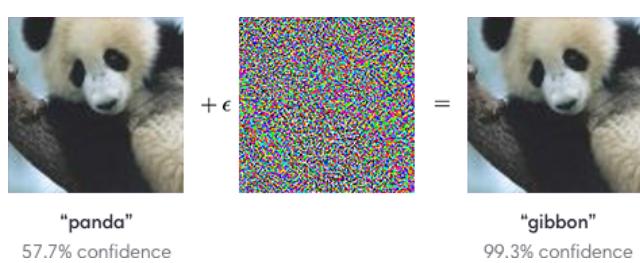


Figure 20.22: Example of an adversarial attack on an image classifier. Left column: original image which is correctly classified. Middle column: small amount of structured noise which is added to the input (magnitude of noise is magnified by $10\times$). Right column: new image, which is confidently misclassified as a “gibbon”, even though it looks just like the original “panda” image. Here $\epsilon = 0.007$. From Figure 1 of [GSS15]. Used with kind permission of Ian Goodfellow.

learning in which the sequence of distributions has the form $p_1, p_*, p_2, p_*, \dots$, and the training mode bit sequence has the form $1, 0, 1, 0, \dots$. This framing makes clear the assumption that we should only evaluate performance on all tasks, p_* , if we think they will occur again in the future. By contrast, if a task will not repeat again, it is okay to forget about it (something that commonly happens in human learning, due to finite memory and compute abilities).

Similarly, we can frame distribution shift and concept shift as special cases of (generalized) online learning. For distribution shift, we let $p_t(\mathbf{x})$ evolve, but keep f_t fixed, and we alternate between training on $p_t(\mathbf{x}, \mathbf{y})$ and testing on $p_*(\mathbf{x}, \mathbf{y})$. For concept drift, we keep $p_t(\mathbf{x})$ fixed, but let f_t evolve, and we alternate between training on $p_t(\mathbf{x}, \mathbf{y})$ and testing on $p_t(\mathbf{x}, \mathbf{y})$.

20.8 Adversarial examples

This section is coauthored with Justin Gilmer.

In Section 20.2, we discussed what happens to a predictive model when the input distribution shifts for some reason. In this section, we consider the case where an adversary deliberately chooses inputs to minimize the performance of a predictive model. That is, suppose an input \mathbf{x} is classified as belonging to class c . We then choose a new input \mathbf{x}_{adv} which minimizes the probability of this label, subject to the constraint that \mathbf{x}_{adv} is “perceptually similar” to the original input \mathbf{x} . This gives rise to the following objective:

$$\mathbf{x}_{\text{adv}} = \underset{\mathbf{x}' \in \Delta(\mathbf{x})}{\operatorname{argmin}} \log p(y = c | \mathbf{x}') \quad (20.44)$$

where $\Delta(\mathbf{x})$ is the set of images that are “similar” to \mathbf{x} (we discuss different notions of similarity below).

Equation (20.44) is an example of an **adversarial attack**. We illustrate this in Figure 20.22. The input image \mathbf{x} is on the left, and is predicted to be a panda with probability 57%. By adding a tiny amount of carefully chosen noise (shown in the middle) to the input, we generate the **adversarial image** \mathbf{x}_{adv} on the right: this “looks like” the input, but is now classified as a gibbon with probability 99%.

1 The ability to create adversarial images was first noted in [Sze+14]. It is surprisingly easy to create
2 such examples, which seems paradoxical, given the fact that modern classifiers seem to work so well
3 on normal inputs, and the perturbed images “look” the same to humans. We explain this paradox in
4 Section 20.8.5.
5

6 The existence of adversarial images also raises security concerns. For example, [Sha+16] showed
7 they could force a face recognition system to misclassify person A as person B , merely by asking
8 person A to wear a pair of sunglasses with a special pattern on them, and [Eyk+18] show that is
9 possible to attach small “**adversarial stickers**” to traffic signs to classify stop signs as speed limit
10 signs.

11 Below we briefly discuss how to create adversarial attacks, why they occur, and how we can try to
12 defend against them. We focus on the case of deep neural nets for images, although it is important
13 to note that many other kinds of models (including logistic regression and generative models) can
14 also suffer from adversarial attacks. Furthermore, this is not restricted to the image domain, but
15 occurs with many kinds of high dimensional inputs. For example, [Li+19] contains an audio attack
16 and [Dal+04; Jia+19] contains a text attack. More details on adversarial examples can be found in
17 e.g., [Wiy+19; Yua+19].
18

19 20.8.1 Whitebox (gradient-based) attacks 20

21 To create an adversarial example, we must find a “small” perturbation $\boldsymbol{\delta}$ to add to the input \mathbf{x} to
22 create $\mathbf{x}_{\text{adv}} = \mathbf{x} + \boldsymbol{\delta}$ so that $f(\mathbf{x}_{\text{adv}}) = y'$, where $f()$ is the classifier, and y' is the label we want to
23 force the system to output. This is known as a **targeted attack**. Alternatively, we may just want
24 to find a perturbation that causes the current predicted label to change from its current value to any
25 other value, so that $f(\mathbf{x} + \boldsymbol{\delta}) \neq f(\mathbf{x})$, which is known as **untargeted attack**.

26 In general, we define the objective for the adversary as *maximizing* the following loss:
27

$$\mathbf{x}_{\text{adv}} = \underset{\mathbf{x}' \in \Delta(\mathbf{x})}{\operatorname{argmax}} \mathcal{L}(\mathbf{x}', y; \boldsymbol{\theta}) \quad (20.45)$$

30 where y is the true label. For the untargeted case, we can define $\mathcal{L}(\mathbf{x}', y; \boldsymbol{\theta}) = -\log p(y|\mathbf{x}')$, so we
31 minimize the probability of the true label; and for the targeted case, we can define $\mathcal{L}(\mathbf{x}', y; \boldsymbol{\theta}) =$
32 $\log p(y'|\mathbf{x}')$, where we maximize the probability of the desired label $y' \neq y$.

34 To define what we mean by “small” perturbation, we impose the constraint that $\mathbf{x}_{\text{adv}} \in \Delta(\mathbf{x})$,
35 which is the set of “perceptually similar” images to the input \mathbf{x} . Most of the literature has focused
36 on a simplistic setting in which the adversary is restricted to making bounded l_p perturbations of a
37 clean input \mathbf{x} , that is

$$\Delta(\mathbf{x}) = \{\mathbf{x}' : \|\mathbf{x}' - \mathbf{x}\|_p < \epsilon\} \quad (20.46)$$

41 Typically people assume $p = 1$ or $p = 0$. We will discuss more realistic threat models in Section 20.8.3.

42 In this section, we assume that the attacker knows the model parameters $\boldsymbol{\theta}$; this is called a
43 **whitebox attack**, and lets us use gradient based optimization methods. We relax this assumption
44 in Section 20.8.2.)

45 To solve the optimization problem in Equation (20.45), we can use any kind of constrained
46 optimization method. In [Sze+14] they used bound-constrained BFGS. [GSS15] proposed the more
47

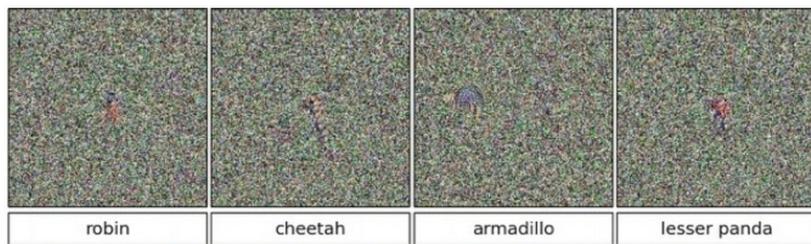


Figure 20.23: Images that look like random noise but which cause the CNN to confidently predict a specific class. From Figure 1 of [NYC15]. Used with kind permission of Jeff Clune.

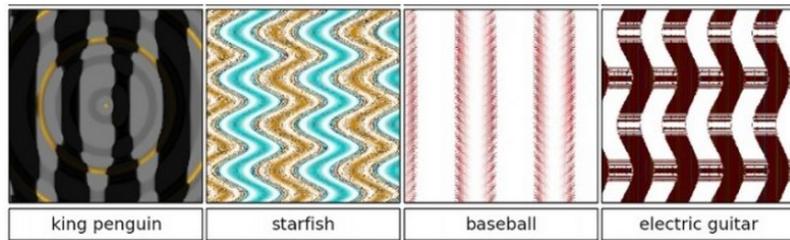


Figure 20.24: Synthetic images that cause the CNN to confidently predict a specific class. From Figure 1 of [NYC15]. Used with kind permission of Jeff Clune.

efficient **fast gradient sign (FGS)** method, which performs iterative updates of the form

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \boldsymbol{\delta}_t \quad (20.47)$$

$$\boldsymbol{\delta}_t = \epsilon \operatorname{sign}(\nabla_{\mathbf{x}} \log p(y'|\mathbf{x}, \boldsymbol{\theta})|_{\mathbf{x}_t}) \quad (20.48)$$

where $\epsilon > 0$ is a small learning rate. (Note that this gradient is with respect to the input pixels, not the model parameters.) Figure 20.22 gives an example of this process.

More recently, [Mad+18] proposed the more powerful **projected gradient descent (PGD)** attack; this can be thought of as an iterated version of FGS. There is no “best” variant of PGD for solving 20.45. Instead, what matters more is the implementation details, e.g. how many steps are used, the step size, and the exact form of the loss. To avoid local minima, we may use random restarts, choosing random points in the constraint space Δ to initialize the optimization. The algorithm should be carefully tuned to the specific problem, and the loss should be monitored to check for optimization issues. For best practices, see [Car+19].

20.8.2 Blackbox (gradient-free) attacks

In this section, we no longer assume that the adversary knows the parameters $\boldsymbol{\theta}$ of the predictive model f . This is known as a **black box attack**. In such cases, we must use derivative-free optimization methods (see Section 6.12).

Evolutionary algorithms (EA) are one class of DFO solvers. These were used in [NYC15] to create blackbox attacks. Figure 20.23 shows some images that were generated by applying an EA to a

¹ random noise image. These are known as **fooling images**, as opposed to adversarial images, since
² they are not visually realistic. Figure 20.24 shows some fooling images that were generated by
³ applying EA to the parameters of a compositional pattern-producing network [Sta07].⁹ By suitably
⁴ perturbing the CPPN parameters, it is possible to generate structured images with high fitness
⁵ (classifier score), but which do not look like natural images [Aue12].

⁶ In [SVK19], they used differential evolution to attack images by modifying a single pixel. This is
⁷ equivalent to bounding the ℓ_0 norm of the perturbation, so that $\|\mathbf{x}_{\text{adv}} - \mathbf{x}\|_0 = 1$.

⁸ In [Pap+17], they learned a differentiable surrogate model of the blackbox, by just querying its
⁹ predictions y for different inputs \mathbf{x} . They then used gradient-based methods to generate adversarial
¹⁰ attacks on their surrogate model, and then showed that these attacks transferred to the real model.
¹¹ In this way, they were able to attack various the image classification APIs of various cloud service
¹² providers, including Google, Amazon and MetaMind.

¹³

¹⁴ 20.8.3 Real world adversarial attacks

¹⁵ Typically, the space of possible adversarial inputs Δ can be quite large, and will be difficult to exactly
¹⁶ define mathematically as it will depend on semantics of the input based on the attacker's goals
¹⁷ [BR18]. (The set of variations Δ that we want the model to be invariant to is called the **threat**
¹⁸ **model**.)

¹⁹ Consider for example of the content constrained threat model discussed in [Gil+18a]. One instance
²⁰ of this threat model involves image spam, where the attacker wishes to upload an image attachment
²¹ in an email that will not be classified as spam by a detection model. In this case Δ is incredibly
²² large as it consists of all possible images which contain some semantic concept the attacker wishes to
²³ upload (in this case an advertisement). To explore Δ , spammers can utilize different fonts, word
²⁴ orientations or add random objects to the background as is the case of the adversarial example in
²⁵ Figure 20.25 (see [Big+11] for more examples). Of course, optimization based methods may still be
²⁶ used here to explore parts of Δ . However, in practice it may be preferable to design an adversarial
²⁷ input by hand as this can be significantly easier to execute with only limited-query black-box access
²⁸ to the underlying classifier.

²⁹

³⁰ 20.8.4 Defenses based on robust optimization

³¹ As discussed in Section 20.8.3, securing a system against adversarial inputs in more general threat
³² models seems extraordinarily difficult, due to the vast space of possible adversarial inputs Δ . However,
³³ there is a line of research focused on producing models which are invariant to perturbations within
³⁴ a small constraint set $\Delta(\mathbf{x})$, with a focus on l_p -robustness where $\Delta(\mathbf{x}) = \{\mathbf{x}' : \|\mathbf{x} - \mathbf{x}'\|_p < \epsilon\}$.
³⁵ Although solving this toy threat model has little application to security settings, enforcing smoothness
³⁶ priors have in some cases improved robustness to random image corruptions [SHS], lead to models
³⁷ which transfer better [Sal+20], and can bias models towards different features in the data [Yin+19a].
³⁸ Perhaps the most straightforward method for improving l_p -robustness is to directly optimize for
³⁹ it through **robust optimization** [BTEGN09], also known as **adversarial training** [GSS15]. We
⁴⁰

⁴¹ ⁴² ⁴³ ⁴⁴ ⁴⁵ ⁴⁶ 9. A CPPN is a set of elementary functions (such as linear, sine, sigmoid, and Gaussian) which can be composed in
⁴⁷ order to specify the mapping from each coordinate to the desired color value. CPPN was originally developed as a way
⁴⁸ to encode abstract properties such as symmetry and repetition, which are often seen during biological development.



Figure 20.25: An adversarially modified image to evade spam detectors. The image is constructed from scratch, and does not involve applying a small perturbation to any given image. This is an illustrative example of how large the space of possible adversarial inputs Δ can be when the attacker has full control over the input. From [Big+11]. Used with kind permission of Battista Biggio.

define the **adversarial risk** to be

$$\min_{\theta} \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p(\mathbf{x}, \mathbf{y})} \left[\max_{\mathbf{x}' \in \Delta(\mathbf{x})} L(\mathbf{x}', \mathbf{y}; \theta) \right] \quad (20.49)$$

The min max formulation in equation 20.49 poses unique challenges from an optimization perspective—it requires solving both the non-concave inner maximization and the non-convex outer minimization problems. Even worse, the inner max is NP-hard to solve in general [Kat+17]. However, in practice it may be sufficient to compute the gradient of the outer objective $\nabla_{\theta} L(\mathbf{x}_{\text{adv}}, \mathbf{y}; \theta)$ at an approximately maximal point in the inner problem $\mathbf{x}_{\text{adv}} \approx \text{argmax}_{\mathbf{x}} L(\mathbf{x}_{\text{adv}}, \mathbf{y}; \theta)$ [Mad+18]. Currently, best practice is to approximate the inner problem using a few steps of PGD.

Other methods seek to **certify** that a model is robust within a given region $\Delta(x)$. One method for certification uses randomized smoothing [CRK19]—a technique for converting a model robust to random noise into a model which is provably robust to bounded worst-case perturbations in the l_2 -metric. Another class of methods applies specifically for networks with ReLU activations, leveraging the property that the model is locally linear, and that certifying in region defined by linear constraints reduces to solving a series of linear programs, for which standard solvers can be applied [WK18].

20.8.5 Why models have adversarial examples

The existence of adversarial inputs is paradoxical, since modern classifiers seem to do so well on normal inputs. However, the existence of adversarial examples is a natural consequence of the general lack of robustness to distribution shift discussed in Section 20.2. To see this, suppose a model’s accuracy drops on some shifted distribution of inputs $p_{\text{te}}(\mathbf{x})$ that differs from the training distribution $p_{\text{tr}}(\mathbf{x})$; in this case, the model will necessarily be vulnerable to an adversarial attack: if errors exist, there must be a nearest such error. Furthermore, if the input distribution is high

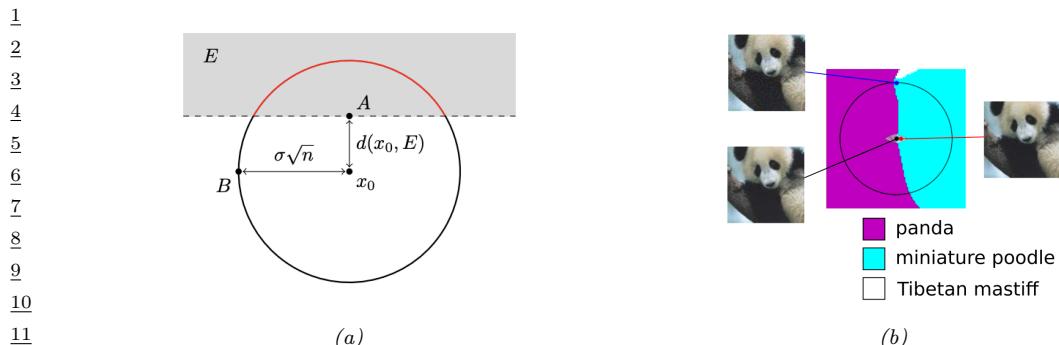


Figure 20.26: (a) When the input dimension n is large and the decision boundary is locally linear, even a small error rate in random noise will imply the existence of small adversarial perturbations. Here, $d(\mathbf{x}_0, E)$ denotes the distance from a clean input \mathbf{x}_0 to an adversarial example (A) while the distance from \mathbf{x}_0 to a random sample $N(0; \sigma^2 I)$ (B) will be approximately $\sigma\sqrt{n}$. As $n \rightarrow \infty$ the ratio of $d(\mathbf{x}_0, A)$ to $d(\mathbf{x}_0, B)$ goes to 0. (b) A 2d slice of the InceptionV3 decision boundary through three points: a clean image (black), an adversarial example (red), and an error in random noise (blue). The adversarial example and the error in noise lie in the same region of the error set which is misclassified as “miniature poodle”, which closely resembles a halfspace as in Figure (a). Used with kind permission of Justin Gilmer.

²² dimensional, then we should expect the nearest error to be significantly closer than errors which are
²³ sampled randomly from some out-of-distribution $p_{te}(\mathbf{x})$.

²⁴ A cartoon illustration of what is going on is shown in Figure 20.26a, where \mathbf{x}_0 is the clean input
²⁵ image, B is an image corrupted by Gaussian noise, and A is an adversarial image. If we assume a
²⁶ linear decision boundary, then the error set E is a half space a certain distance from \mathbf{x}_0 . We can
²⁷ relate the distance to the decision boundary $d(\mathbf{x}_0, E)$ with the error rate in noise at some input \mathbf{x}_0 ,
²⁸ denoted by $\mu = \mathbb{P}_{\delta \sim N(0, \sigma I)} [\mathbf{x}_0 + \delta \in E]$. With a linear decision boundary the relationship between
²⁹ these two quantities is determined by

$$\frac{31}{30} \quad d(x_0, E) = -\sigma \Phi^{-1}(\mu) \quad (20.50)$$

³³ where Φ^{-1} denotes the inverse cdf of the gaussian distribution. When the input dimension is large,
³⁴ this distance will be significantly smaller than the distance to a randomly sampled noisy image
³⁵ $\mathbf{x}_0 + \delta$ for $\delta \sim N(0, \sigma I)$, as the noise term will with high probability have norm $\|\delta\|_2 \approx \sigma\sqrt{d}$. As a
³⁶ concrete example consider the ImageNet dataset, where $d = 224 \times 224 \times 3$ and suppose we set $\sigma = .2$.
³⁷ Then if the error rate in noise is just $\mu = .01$, equation 20.50 will imply that $d(\mathbf{x}_0, E) = .5$. Thus the
³⁸ distance to an adversarial example will be more than 100 times closer than the distance to a typical
³⁹ noisy images, which will be $\sigma\sqrt{d} \approx 77.6$. This phenomenon of small volume error sets being close
⁴⁰ to most points in a data distribution $p(\mathbf{x})$ is called **concentration of measure**, and is a property
⁴¹ common among many high dimensional data distributions [MDM19; Gil+18b].

42 In summary, although the existence of adversarial examples is often discussed as an unexpected
43 phenomenon, there is nothing special about the existence of worst-case errors for ML classifiers—they
44 will always exist as long as errors exist.

PART IV

Generation

21 Generative models: an overview

21.1 Introduction

A **generative model** is a joint probability distribution $p(\mathbf{x})$, for $\mathbf{x} \in \mathcal{X}$. In some cases, the model may be conditioned on inputs or covariates $\mathbf{c} \in \mathcal{C}$, which gives rise to a **conditional generative model** of the form $p(\mathbf{x}|\mathbf{c})$.

There are many kinds of generative model. We give a brief summary in Section 21.2, and go into more detail in subsequent chapters. See also [Tom22] for a recent book on this topic that goes into more depth.

21.2 Types of generative model

There are many kinds of generative model, some of which we list in Table 21.1. At a high level, we can distinguish between **deep generative models** (DGM) — which use deep neural network to learn a complex mapping from a single latent vector \mathbf{z} to the observed data \mathbf{x} — and more “classical” **probabilistic graphical models** (PGM), that map a set of interconnected latent variables $\mathbf{z}_1, \dots, \mathbf{z}_L$ to the observed variables $\mathbf{x}_1, \dots, \mathbf{x}_D$ using simpler, often linear, mappings. Of course, many hybrids are possible. For example, PGMs can use neural networks, and DGMs can use structured state spaces. We discuss PGMs in general terms in Chapter 4, and give examples in Chapter 29, Chapter 30, Chapter 31, Chapter 32. In this part of the book, we mostly focus on DGMs.

The main kinds of DGM are: **variational autoencoders (VAE)**, **autoregressive (AR)** models, **normalizing flows**, **diffusion models**, **energy based models (EBM)**, and **generative adversarial networks (GAN)**. We can categorize these models in terms of the following criteria (see Figure 21.1 for a visual summary):

- Density: does the model support pointwise evaluation of the probability density function $p(\mathbf{x})$, and if so, is this fast or slow, exact, approximate or a bound, etc? For **implicit models**, such as GANs, there is no well-defined density $p(\mathbf{x})$. For other models, we can only compute a lower bound on the density (VAEs), or an approximation to the density (EBMs, UGMs).
- Sampling: does the model support generating new samples, $\mathbf{x} \sim p(\mathbf{x})$, and if so, is this fast or slow, exact or approximate? Directed PGMs, VAEs and GANs all support fast sampling. However, undirected PGMs, EBMs, AR, diffusion and flows are slow for sampling.
- Training: what kind of method is used for parameter estimation? For some models (such as AR, flows and directed PGMs), we can perform exact maximum likelihood estimation (MLE), although

Model	Chapter	Density	Sampling	Training	Latents	Architecture
PGM-D	Chapter 4	Exact, fast	Fast	MLE	Optional	Sparse DAG
PGM-U	Chapter 4	SA, slow	Slow	MLE-A	Optional	Sparse graph
FA	Chapter 29	Exact, fast	Fast	MLE	\mathbb{R}^D	Linear
HMM	Chapter 30	Exact, fast	Fast	MLE	$\{1, \dots, K\}^T$	Chain
SSM-LG	Chapter 31	Exact, fast	Fast	MLE	$\mathbb{R}^{L \times T}$	Chain
SSM-NLG	Chapter 31	SA, fast	Fast	MLE-A	$\mathbb{R}^{L \times T}$	Chain
VAE	Chapter 22	LB, fast	Fast	MLE-LB	\mathbb{R}^L	Encoder-Decoder
AR	Chapter 23	Exact, fast	Slow	MLE	None	Sequential
Flows	Chapter 24	Exact, slow/fast	Slow	MLE	\mathbb{R}^D	Invertible
EBM	Chapter 25	SA, slow	Slow	MLE-A	Optional	Discriminative
Diffusion	Chapter 26	LB	Slow	MLE-LB	\mathbb{R}^D	Encoder-Decoder
GAN	Chapter 27	NA	Fast	Min-max	\mathbb{R}^L	Generator-Discriminator

Table 21.1: Characteristics of common kinds of generative model. Models above the line are “classical” probabilistic graphical models; models below the line are considered to be “deep” generative models (even though some of the models above the line can also use neural networks). Here D is the dimensionality of the observed \mathbf{x} (for time series data, we assume \mathbf{x} is $D \times T$ dimensional), and L is the dimensionality of the latent \mathbf{z} , if present. Abbreviations: MLE-A = MLE (Approximate), AR = autoregressive, EBM = Energy Based Model, FA = factor analysis, GAN = generative adversarial network, HMM = hidden Markov model, LB = lower bound, MLE = maximum likelihood estimation, MLE-LB = maximizing lower bound of the likelihood, PGM-D = directed probabilistic graphical model, PGM-U = undirected probabilistic graphical model, SA = stochastic approximation, SSM = state space model, SSM-LG = linear Gaussian SSM, SSM-NLG = non-linear and/or non-Gaussian SSM, VAE = variational autoencoder.

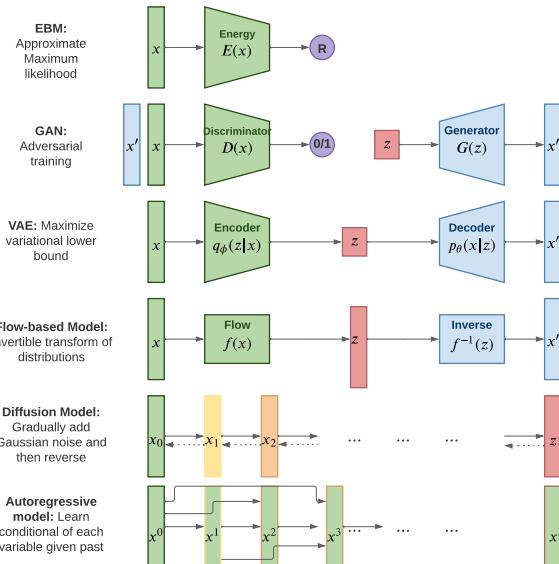


Figure 21.1: Summary of various kinds of deep generative model. Here \mathbf{x} is the observed data, \mathbf{z} is the latent code, and \mathbf{x}' is a sample from the model. AR models do not have a latent code \mathbf{z} . For diffusion models and flow models, the size of \mathbf{z} is the same as \mathbf{x} . For AR models, x^d is the d 'th dimension of \mathbf{x} . For diffusion models, \mathbf{x}_t is the t 'th “noised up version of \mathbf{x} , where $\mathbf{x}_0 = \mathbf{x}$ and $\mathbf{z} = \mathbf{x}_T$. Adapted from Figure 1 of [Wen21].



Figure 21.2: Synthetic faces from a score-based generative model (Section 25.3.5). From Figure 12 of [Son+21]. Used with kind permission of Yang Song.

the objective is usually non-convex, so we can only reach a local optimum. For other models, we cannot tractably compute the likelihood. In the case of VAEs, we maximize a lower bound on the likelihood; in the case of EBMs and UGMs, we maximize an approximation to the likelihood. For GANs we have to use min-max training, which can be unstable, and there is no clear objective function to monitor.

- Latents: does the model use a latent vector \mathbf{z} to generate \mathbf{x} or not, and if so, is it the same size as \mathbf{x} or is it a potentially compressed representation? For example, AR models do not use latents; flows and diffusion use latents, but they are not compressed. Graphical models, including EBMs, may or may not use latents.
- Architecture: what kind of neural network should we use, and are there restrictions? For flows, we are restricted to using invertible neural networks where each layer has a tractable Jacobian. For EBMs, we can use any model we like. The other models have different restrictions.

21.3 Goals of generative modeling

There are several different kinds of tasks that we can use generative models for, as we discuss below.

21.3.1 Generating data

One of the main goals of generative models is to generate (create) new data samples. For example, if we fit a model $p(\mathbf{x})$ to images of faces, we can sample new faces from it, as illustrated in Figure 21.2.¹ Similar methods can be used to create samples of text, audio, etc. When this technology is abused to make fake content, they are called **deep fakes**. For a review of this topic, see e.g., [Ngu+19].

¹ These images were made with a technique called score-based generative modeling (Section 25.3.5), although similar results can be obtained using many other techniques. See for example <https://this-person-does-not-exist.com/en> which shows results from a GAN model (Chapter 27).

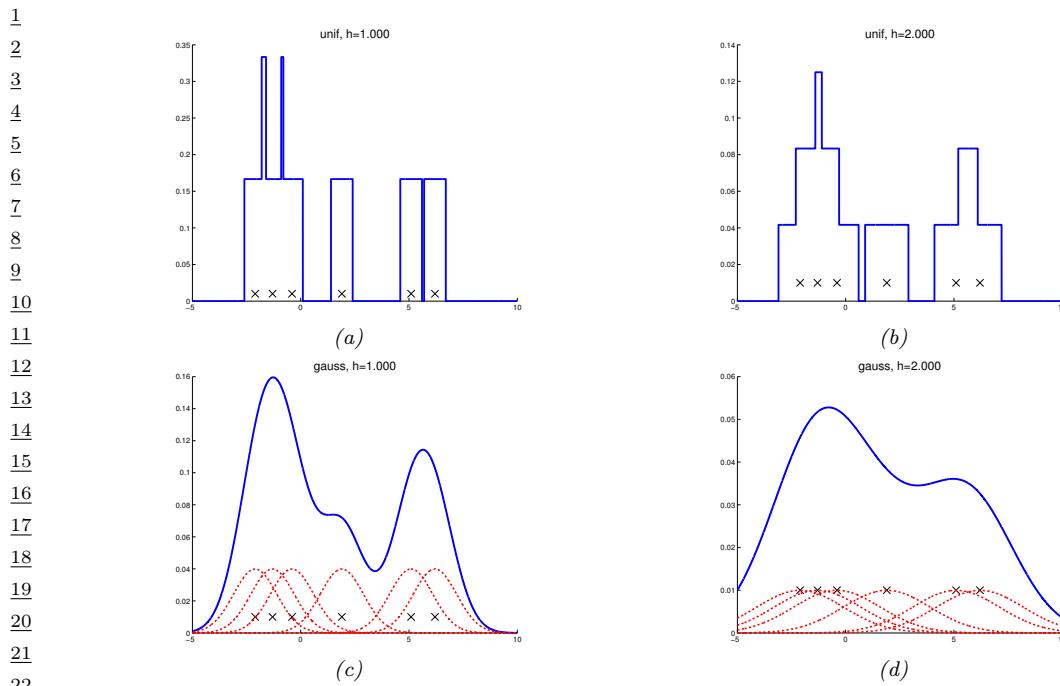


Figure 21.3: A nonparametric (Parzen) density estimator in 1d estimated from 6 data points, denoted by x .
 Top row: uniform kernel. Bottom row: Gaussian kernel. Left column: bandwidth parameter $h = 1$. Right column: bandwidth parameter $h = 2$. Adapted from http://en.wikipedia.org/wiki/Kernel_density_estimation. Generated by `parzen_window_demo2.py`.

To control what is generated, it is useful to use a conditional generative model of the form $p(\mathbf{x}|\mathbf{c})$. For example, \mathbf{c} might be a text prompt and \mathbf{x} might be an image, in which case $p(\mathbf{x}|\mathbf{c})$ is called a text-to-image model. Or \mathbf{c} might be a sequence of sounds and \mathbf{x} might be a sequence of letters, in which case $p(\mathbf{x}|\mathbf{c})$ is called a speech-to-text model, which is useful for **automatic speech recognition**. Or \mathbf{c} might be a sequence of English words and \mathbf{x} might be a sequence of French words, in which case $p(\mathbf{x}|\mathbf{c})$ is called a sequence-to-sequence model, which is useful for **machine translation**.

Note that, in the conditional case, we often denote the inputs by \mathbf{x} and the outputs by \mathbf{y} . In this case the model has the familiar form $p(\mathbf{y}|\mathbf{x})$. In the special case that \mathbf{y} denotes a low dimensional quantity, such as an integer class label, $y \in \{1, \dots, C\}$, we get a predictive (discriminative) model. The main difference between a discriminative model and a conditional generative model is this: in a discriminative model, we assume there is one correct output, whereas in a conditional generative model, we assume there may be multiple correct outputs. This makes it harder to evaluate generative models, as we discuss in Section 21.4.

21.3.2 Density estimation

The task of **density estimation** refers to evaluating the probability of an observed data vector, $p = p(\mathbf{x})$. This can be useful for outlier detection (Section 20.4.2), data compression (Section 5.4),

	Data sample	Variables			Missing values replaced by means		
		A	B	C	A	B	C
1	1	6	6	NA	2	6	7.5
2	2	NA	6	0	9	6	0
3	3	NA	6	NA	9	6	7.5
4	4	10	10	10	10	10	10
5	5	10	10	10	10	10	10
6	6	10	10	10	10	10	10
12	Average	9	8	7.5	9	8	7.5

Figure 21.4: Missing data imputation using the mean of each column.

generative classifiers, model comparison, etc.

A simple approach to this problem, which works in low dimensions, is to use **kernel density estimation** or **KDE**, which has the form

$$p(\mathbf{x}|\mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) \quad (21.1)$$

Here \mathcal{K}_h is a density kernel with **bandwidth** h , which is a function $\mathcal{K} : \mathbb{R} \rightarrow \mathbb{R}_+$ such that $\int \mathcal{K}(x)dx = 1$ and $\int x\mathcal{K}(x)dx = 0$. We give a 1d example of this in Figure 21.3: in the top row, we use a uniform (boxcar) kernel, and in the bottom row we use a Gaussian kernel.

In higher dimensions, KDE suffers from the **curse of dimensionality** (see e.g., [AHK01]), and we need to use parametric density models $p_\theta(\mathbf{x})$ of some kind.

21.3.3 Imputation

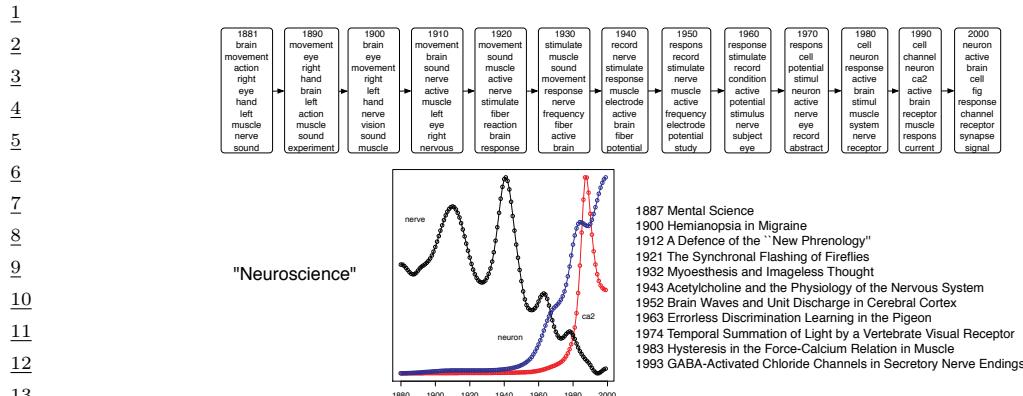
The task of **imputation** refers to “filling in” missing values of a data vector or data matrix. For example, suppose \mathbf{X} is an $N \times D$ matrix of data (think of a spreadsheet) in which some entries, call them \mathbf{X}_m , may be missing, while the rest, \mathbf{X}_o , are observed. A simple way to fill in the missing data is to use the mean value of each feature, $\mathbb{E}[x_d]$; this is called **mean value imputation**, and is illustrated in Figure 21.4. However, this ignores dependencies between the variables within each row, and does not return any measure of uncertainty.

We can generalize this by fitting a generative model to the observed data, $p(\mathbf{X}_o)$, and then computing samples from $p(\mathbf{X}_m|\mathbf{X}_o)$. This is called **multiple imputation**. A generative model can be used to fill in more complex data types, such as **in-painting** occluded pixels in an image.

See Section 22.3.5 for a more general discussion of missing data.

21.3.4 Structure discovery

Some kinds of generative models have latent variables \mathbf{z} , which are assumed to be the “causes” that generated the observed data \mathbf{x} . We can use Bayes rule to invert the model to compute



14 *Figure 21.5: Part of the output of the dynamic topic model when applied to articles from Science. At the top,*
15 *we show the top 10 words for the neuroscience topic over time. On the bottom left, we show the probability of*
16 *three words within this topic over time. On the bottom right, we list paper titles from different years that*
17 *contained this topic. From Figure 4 of [BL06]. Used with kind permission of David Blei.*

21 $p(\mathbf{z}|\mathbf{x}) \propto p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$. This can be useful for discovering latent, low-dimensional patterns in the data.

22 We give an example of this in Figure 21.5, where we show the output of a **dynamic topic model**
23 applied to 100 years of articles from *Science*. We see that the model has discovered various topics
24 (groups of related words), and can track their usage over time. For details on topic models, see
25 the supplementary material. For details on structural discovery using other kinds of latent variable
26 models, see Part V.

28 21.3.5 Latent space interpolation

30 One of the most interesting abilities of certain latent variable models is the ability to generate
31 samples that have certain desired properties by interpolating between existing data points in latent
32 space. To explain how this works, let \mathbf{x}_1 and \mathbf{x}_2 be two inputs (e.g. images), and let $\mathbf{z}_1 = e(\mathbf{x}_1)$ and
33 $\mathbf{z}_2 = e(\mathbf{x}_2)$ be their latent encodings. (The method used for computing these will depend on the
34 type of model; we discuss the details in later chapters.) We can regard \mathbf{z}_1 and \mathbf{z}_2 as two “anchors” in
35 latent space. We can now generate new images that interpolate between these points by computing
36 $\mathbf{z} = \lambda\mathbf{z}_1 + (1 - \lambda)\mathbf{z}_2$, where $0 \leq \lambda \leq 1$, and then decoding by computing $\mathbf{x}' = d(\mathbf{z})$, where $d()$ is the
37 decoder. This is called **latent space interpolation**, and will generate data that combines semantic
38 features from both \mathbf{x}_1 and \mathbf{x}_2 . (The justification for taking a linear interpolation is that the learned
39 manifold often has approximately zero curvature, as shown in [SKTF18]. However, sometimes it
40 is better to use nonlinear interpolation [MB21; Fad+20].) We give an example of this process in
41 Figure 21.6, where we use a simple VAE (Chapter 22) fit to some face images. (Higher quality
42 samples are possible by using other kinds of model.)

43 In some cases, we can go beyond interpolation, and can perform arithmetic in latent space, in
44 which we can increase or decrease the amount of a desired “semantic factor of variation”. This was
45 first shown in the **word2vec** model [Mik+13], but it also is possible in other latent variable models.
46 For example, consider our VAE model fit to **CelebA** dataset, which has faces of celebrities and some

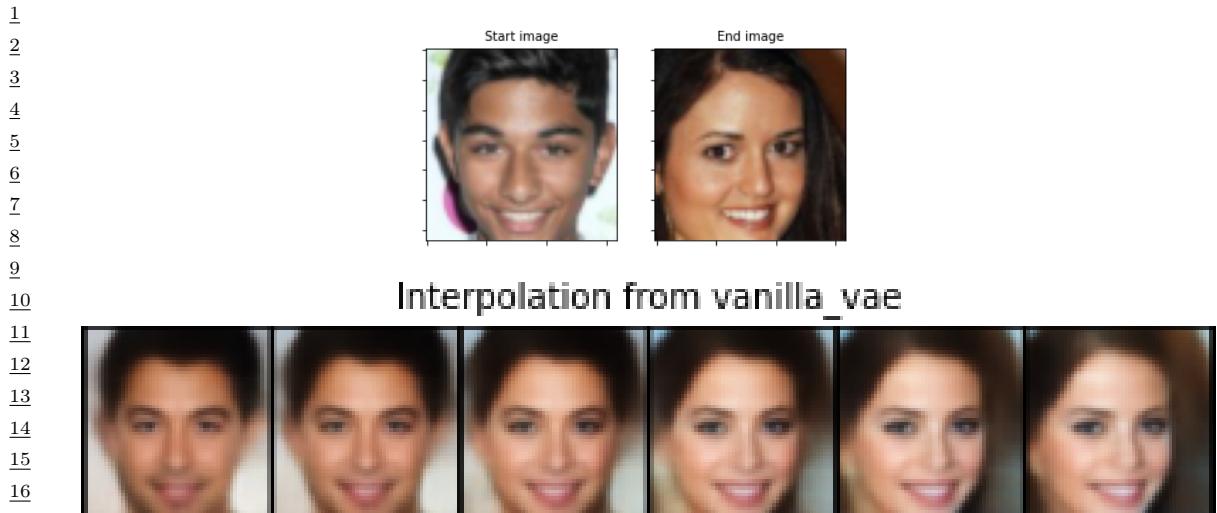


Figure 21.6: Interpolation between two images (top row) in the latent space of a VAE. Generated by [vae_compare_results.ipynb](#).

corresponding attributes. Consider the attribute of wearing sunglasses. Let \mathbf{X}_i^+ be a set of images which have attribute i , and \mathbf{X}_i^- be a set of images which do not have this attribute. Let \mathbf{Z}_i^+ and \mathbf{Z}_i^- be the corresponding embeddings, and $\bar{\mathbf{z}}^+$ and $\bar{\mathbf{z}}^-$ be the average of these embeddings. We define the offset vector as $\Delta = \bar{\mathbf{z}}^+ - \bar{\mathbf{z}}^-$. If we add some positive multiple of Δ to a new point \mathbf{z} , we increase the amount of the sunglass factor; if we subtract some multiple of Δ , we decrease the amount of the sunglass factor [Whi16]. We give an example of this in Figure 21.7. The j 'th reconstruction is computed using $\hat{\mathbf{x}}_j = d(\mathbf{z} + s_j\Delta)$, where $\mathbf{z} = e(\mathbf{x})$ is the encoding of the original image, and s_j is a scale factor. For the VAE, we see that we can remove sunglasses by setting $s = -4$, or make the sunglasses bigger by setting $s = 4$.

21.3.6 Representation learning

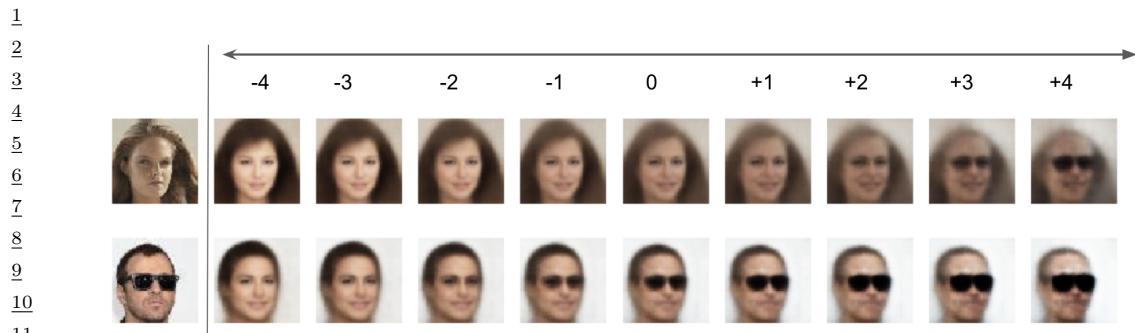
Representation learning refers to learning (possibly uninterpretable) latent factors \mathbf{z} that generate the observed data \mathbf{x} . The primary goal is for these features to be used in “**downstream**” supervised tasks. This is discussed in Chapter 34.

21.4 Evaluating generative models

This section is coauthored with Mihaela Rosca, Shakir Mohamed and Balaji Lakshminarayanan.

Evaluating generative models requires metrics which capture

- **sample quality** - are samples generated by the model part of the data distribution?
- **sample diversity** - are samples from the model distribution capturing all modes of the data



13 Figure 21.7: Arithmetic in the latent space of a VAE . The first column is an input image, with embedding \mathbf{z} .
14 Subsequent columns show the decoding of $\mathbf{z} + s\Delta$, where $s \in \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$ and $\Delta = \bar{\mathbf{z}}^+ - \bar{\mathbf{z}}^-$
15 is the difference in the average embeddings of images with or without a certain attribute (here, wearing
16 sunglasses). Generated by [vae_celeba_lightning.ipynb](#).

17

18 distribution?, and
19

- 20 • **generalization** - is the model generalizing beyond the training data?

21 There is no known metric which meets all these desiderata, but various metrics have been proposed
22 to capture different aspects of the learned distribution, some of which we discuss below.
23

24 21.4.1 Likelihood

26 A standard way to measure how close a model q is to a true distribution p is in terms of the KL
27 divergence (Section 5.1):

$$\frac{29}{30} D_{\text{KL}}(p\|q) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} = -\mathbb{H}(p) + \mathbb{H}(p, q) \quad (21.2)$$

31 where $\mathbb{H}(p)$ is a constant, and $\mathbb{H}(p, q)$ is the cross entropy. If we approximate $p(\mathbf{x})$ by the empirical
32 distribution, we can evaluate the cross entropy in terms of the empirical **negative log likelihood**
33 on the dataset:

$$\frac{35}{36} \text{NLL} = -\frac{1}{N} \sum_{n=1}^N \log q(\mathbf{x}_n) \quad (21.3)$$

38 Usually we care about negative log likelihood on a held-out test set.²

39

40 21.4.1.1 Evaluating log-likelihood

41 For models of discrete data, such as language models, it is easy to compute the (negative) log
42 likelihood. However, it is common to measure performance using a quantity called **perplexity**, which
43 is defined as 2^H , where $H = \text{NLL}$ is the cross entropy or negative log likelihood.
44

45 2. In some applications, we report **bits per dimension**, which is the NLL using log base 2, divided by the dimensionality
46 of \mathbf{x} . (To compute this metric, recall that $\log_2 L = \frac{\log_e L}{\log_2 e}$.)

47

For image and audio models, one complication is that the model is usually a continuous distribution $p(\mathbf{x}) \geq 0$ but the data is usually discrete (e.g., $\mathbf{x} \in \{0, \dots, 255\}^D$ if we use one byte per pixel). Consequently the average log likelihood can be arbitrary large, since the pdf can be bigger than 1. To avoid this it is standard practice to use **uniform dequantization** [TOB16], in which we add uniform random noise to the discrete data, and then treat it as continuous-valued data. This gives a lower bound on the average log likelihood of the discrete model on the original data.

To see this, let \mathbf{z} be a continuous latent variable, and \mathbf{x} be a vector of binary observations computed by rounding, so $p(\mathbf{x}|\mathbf{z}) = \delta(\mathbf{x} - \text{round}(\mathbf{z}))$, computed elementwise. We have $p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$. Let $q(\mathbf{z}|\mathbf{x})$ be a probabilistic inverse of \mathbf{x} , that is, it has support only on values where $p(\mathbf{x}|\mathbf{z}) = 1$. In this case, Jensen's inequality gives

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x})] \quad (21.4)$$

$$= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x})] \quad (21.5)$$

Thus if we model the density of $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$, which is a dequantized version of \mathbf{x} , we will get a lower bound on $p(\mathbf{x})$.

21.4.1.2 Challenges with using likelihood

Unfortunately, there are several challenges with using likelihood to evaluate generative models, some of which we discuss below.

21.4.1.3 Likelihood can be hard to compute

For many models, computing the likelihood can be computationally expensive, since it requires knowing the normalization constant of the probability model. One solution is to use variational inference (Chapter 10), which provides a way to efficiently compute lower (and sometimes upper) bounds on the log likelihood. Another solution is to use annealed importance sampling (Section 11.5.4.1), which provides a way to estimate the log likelihood using Monte Carlo sampling. However, in the case of implicit generative models, such as GANs (Chapter 27), the likelihood is not even defined, so we need to find evaluation metrics that do not rely on likelihood.

21.4.1.4 Likelihood is not related to sample quality

A more subtle concern with likelihood is that it is often uncorrelated with the perceptual quality of the samples, at least for real-valued data, such as images and sound. In particular, a model can have great log-likelihood but create poor samples and vice versa.

To see why a model can have good likelihoods but create bad samples, consider the following argument from [TOB16]. Suppose q_0 is a density model for D -dimensional data \mathbf{x} which performs arbitrarily well as judged by average log-likelihood, and suppose q_1 is a bad model, such as white noise. Now consider samples generated from the mixture model

$$q_2(\mathbf{x}) = 0.01q_0(\mathbf{x}) + 0.99q_1(\mathbf{x}) \quad (21.6)$$

Clearly 99% of the samples will be poor. However, the log-likelihood per pixel will hardly change between q_2 and q_0 if D is large, since

$$\log q_2(\mathbf{x}) = \log[0.01q_0(\mathbf{x}) + 0.99q_1(\mathbf{x})] \geq \log[0.01q_0(\mathbf{x})] = \log q_0(\mathbf{x}) - 100 \quad (21.7)$$

1 For high-dimensional data, $|\log q_0(\mathbf{x})| \sim D \gg 100$, so $\log q_2(\mathbf{x}) \approx \log q_0(\mathbf{x})$, and hence mixing in the
 2 poor sampler does not significantly impact the log likelihood.
 3

4 Now consider a case where the model has good samples but bad likelihoods. To achieve this,
 5 suppose q is a GMM centered on the training images:

$$6 \quad q(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^N \mathcal{N}(\mathbf{x} | \mathbf{x}_n, \epsilon^2 \mathbf{I}) \quad (21.8)$$

9 If ϵ is small enough that the Gaussian noise is imperceptible, then samples from this model will look
 10 good, since they correspond to the training set of real images. But this model will almost certainly
 11 have poor likelihood on the test set due to overfitting. (In this case we say the model has effectively
 12 just memorized the training set.)
 13

14 21.4.2 Distances and divergences in feature space

15 Due to the challenges associated with comparing distributions in high dimensional spaces, and the
 16 desire to compare distributions in a semantically meaningful way, it is common to use domain-specific
 17 **perceptual distance metrics**, that measure how similar data vectors are to each other or to the
 18 training data. However, most metrics used to evaluate generative models do not directly compare
 19 raw data (e.g. pixels) but use a neural network to obtain features from the raw data and compare
 20 the feature distribution obtained from model samples with the feature distribution obtained from
 21 the dataset. The neural network used to obtain features can be trained solely for the purpose of
 22 evaluation, or can be pretrained; a common choice is to use a pretrained classifier (see e.g., [Sal+16;
 23 Heu+17b; Bin+18; Kyn+19; SSG18a]).

24 The **Inception Score** [Sal+16] measures the average KL divergence between the marginal distribution of class labels obtained from the samples $p_{\theta}(y) = \int p(y|\mathbf{x})p_{\theta}(\mathbf{x})$ and the distribution $p(y|\mathbf{x})$ obtained from a sample $\mathbf{x} \sim p_{\theta}(\mathbf{x})$. This leads to the following score:

$$25 \quad IS = \exp [\mathbf{E}_{p_{\theta}(\mathbf{x})} \mathbb{KL}(p(y|\mathbf{x}) || p_{\theta}(y))] \quad (21.9)$$

26 If a model produces high quality samples from all classes in the dataset, then $p_{\theta}(y)$ should be close
 27 to uniform, while $p(y|\mathbf{x})$ should be a sharp distribution corresponding to the class associated with \mathbf{x} ;
 28 this leads to a high $\mathbb{KL}(p(y|\mathbf{x}) || p_{\theta}(y))$ and thus a high Inception Score.

29 The Inception Score solely relies on class labels, and thus does not measure overfitting or sample
 30 diversity outside the predefined dataset classes. For example, a model which generates one perfect
 31 example per class would get a perfect Inception Score, despite not capturing the variety of examples
 32 inside a class, as shown in Figure 21.8a. To address this drawback, the **Fréchet Inception Distance**
 33 or **FID** score [Heu+17b] measures the Fréchet distance between two Gaussian distributions on sets
 34 of features of a pre-trained classifier. One Gaussian is obtained by passing model samples through a
 35 pretrained classifier, and the other by passing samples the dataset through the same classifier. If we
 36 assume that the mean and covariance obtained from model features are μ_m and Σ_m and those from
 37 the data are μ_d and Σ_d , then the FID is
 38

$$39 \quad FID = \|\mu_m - \mu_d\|_2^2 + \text{trace}(\Sigma_d + \Sigma_m - 2(\Sigma_d \Sigma_m)^{1/2}) \quad (21.10)$$

40 Since it uses features instead of class logits, the Fréchet distance captures more than modes captured
 41 by class labels, as shown in Figure 21.8b. Unlike the Inception score, a lower score is better since we
 42 want the two distributions to be as close as possible.
 43

44

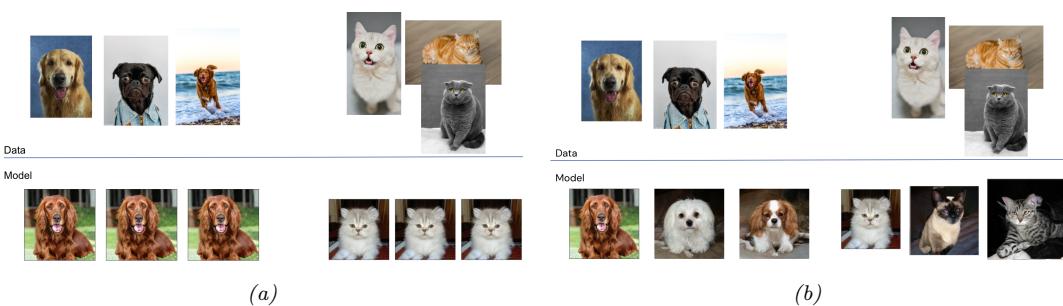


Figure 21.8: (a) Model samples with good (high) inception score are visually realistic. (b) Model samples with good (low) FID score are visually realistic and diverse.

Unfortunately, the Fréchet distance has been shown to have a high bias, with results varying widely based on the number of samples used to compute the score. To mitigate this issue, the **Kernel Inception Distance** has been introduced [Bin+18], which measures the squared MMD (Section 2.9.3) between the features obtained from the data and features obtained from model samples.

21.4.3 Precision and recall metrics

Since the FID only measures the distance between the data and model distributions, it is difficult to use it as a diagnostic tool: a bad (high) FID can indicate that the model is not able to generate high quality data, or that it puts too much mass around the data distribution (e.g. in Figure 27.7a), or that the model only captures a subset of the data (e.g. in Figure 27.7d). Trying to disentangle between these two failure modes has been the motivation to seek individual precision (sample quality) and recall (sample diversity) metrics in the context of generative models [LPO17; Kyn+19]. (The diversity question is especially important in the context of GANs, where mode collapse (Section 27.3.3) can be an issue.)

A common approach taken is to use nearest neighbors in the feature space of a pretrained classifier to define precision and recall [Kyn+19]. To formalize this, let us define

$$f_k(\phi, \Phi) = \begin{cases} 1 & \text{if } \exists \phi' \in \Phi \text{ s.t. } \|\phi - \phi'\|_2^2 \leq \|\phi' - \text{NN}_k(\phi', \Phi)\|_2^2 \\ 0 & \text{otherwise} \end{cases} \quad (21.11)$$

where $\text{NN}_k(\phi', \Phi)$ is a function returning the k -th nearest neighbor of ϕ' in Φ , where Φ is a set of feature vectors. We now define precision and recall as follows:

$$\text{precision}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{model}|} \sum_{\phi \in \Phi_{model}} f_k(\phi, \Phi_{data}); \quad (21.12)$$

$$\text{recall}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{data}|} \sum_{\phi \in \Phi_{data}} f_k(\phi, \Phi_{model}); \quad (21.13)$$

Precision and recall are always between 0 and 1. Intuitively, the precision metric measures whether samples are as close to data as data is to other data examples, while recall measures whether data

1 is as close to model samples as model samples are to other samples. The parameter k controls
2 how lenient the metrics will be – the higher k , the higher both precision and recall will be. As in
3 classification, precision and recall in generative models can be used to construct a trade-off curve
4 between different models which allow practitioners to make an informed decision regarding which
5 model they want to use.
6

7

8 **21.4.4 Statistical tests**

9 Statistical tests have been long used to determine whether two set of samples have been generated
10 from the same distribution; these types of statistical tests are called **two sample tests**. Let us
11 define the null hypothesis to represent that the set of samples are from the same distribution. We
12 then compute a statistic from the data and compare it to a threshold, and based on this we decide
13 whether to reject the null hypothesis. In the context of evaluating implicit generative models such as
14 GANs, statistics based on classifiers [Saj+18] and the MMD [Liu+20b] have been used. To adapt to
15 the high dimensional input spaces, which are ubiquitous in the era of deep learning, two sample tests
16 have been adapted to use learned features instead of raw data.
17

18 Like all other evaluation metrics for generative models, statistical tests have their own advantages
19 and disadvantages: while users can specify Type 1 error – the chance they allow that the null
20 hypothesis is wrongly rejected – statistical tests tend to be computationally expensive and thus
21 cannot be used to monitor progress in training, but rather ought just to be used to compare fully
22 trained models.
23

24 **21.4.5 Challenges with using pretrained classifiers**

25 While popular and convenient, evaluation metrics that rely on pretrained classifiers (such as IS, FID,
26 nearest neighbors in feature space, and statistical tests in feature space) have significant drawbacks.
27 One might not have a pretrained classifier available for the dataset at hand, so classifiers trained on
28 other datasets are used. Given the well known challenges with neural network generalization (see
29 Section 17.5), the features of a classifier trained on images from one dataset might not be reliable
30 enough to provide a fine grained signal of quality for samples obtained from a model trained on a
31 different dataset. If the generative model is trained on the same dataset as the pre-trained classifier
32 but the model is not capturing the data distribution perfectly, we are presenting the pre-trained
33 classifier with out-of-distribution data and relying on its features to obtain score to evaluate our
34 models. Far from being purely theoretical concerns, these issues have been studied extensively and
35 have been shown to affect evaluation in practice [RV19; BS18].
36

37 **21.4.6 Using model samples to train classifiers**

38 Instead of using pretrained classifiers to evaluate samples, one can *train* a classifier on samples from
39 conditional generative models , and then see how good these classifiers are at classifying data. For
40 example, does adding synthetic (sampled) data to the real data help? This is closer to a reliable
41 evaluation of generative model samples, since ultimately, the performance of generative models is
42 dependent on the downstream task they are trained for. If used for semi supervised learning, one
43 should assess how much adding samples to a classifier dataset helps with test accuracy. If used for
44 model based reinforcement learning, one should assess how much the generative model helps with
45 agent performance. For examples of this approach, see e.g., [SSM18; SSA18; RV19; SS20].
46

47



Figure 21.9: Illustration of nearest neighbors in feature space: in the top left we have the query sample generated using BigGAN, and the rest of the images are its nearest neighbors from the dataset. The nearest neighbors search is done in the feature space of a pretrained classifier. From Figure 13 of [BDS18]. Used with kind permission of Andy Brock.

21.4.7 Assessing overfitting

Many of the metrics discussed so far capture the sample quality and diversity, but do not capture overfitting to the training data. To capture overfitting, often a visual inspection is performed: a set of samples is generated from the model and for each sample its closest K nearest neighbors in the feature space of a pretrained classifier are obtained from the dataset. While this approach requires manually assessing samples, it is a simple way to test whether a model is simply memorizing the data. We show an example in Figure 21.9: since the model sample in the top left is quite different than its neighbors from the dataset (remaining images), we can conclude the sample is not simply memorised from the dataset. Similarly, sample diversity can be measured by approximating the support of the learned distribution by looking for similar samples in a large sample pool — as in the pigeonhole principle — but it is expensive and often requires manual human assessment [AZ17].

For likelihood-based models — such as variational autoencoders Chapter 22, autoregressive models Chapter 23, and normalising flows Chapter 24 — we can assess memorisation by seeing how much the log-likelihood of a model changes when a sample is included in the model’s training set or not [BW21].

21.4.8 Human evaluation

One approach to evaluate generative models is to use human evaluation, by presenting samples from the model along side samples from the data distribution, and ask human raters to compare

1 the quality of the samples [Zho+19b]. Human evaluation is a suitable metric if the model is used
2 to create art or other data for human display, or if reliable automated metrics are hard to obtain.
3 However, human evaluation can be difficult to standardize, hard to automate and can be expensive
4 or cumbersome to set up.
5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

22 Variational autoencoders

22.1 Introduction

In this chapter, we discuss generative models of the form

$$\mathbf{z} \sim p_{\theta}(\mathbf{z}) \tag{22.1}$$

$$\mathbf{x}|\mathbf{z} \sim \text{Expfam}(\mathbf{x}|d_{\theta}(\mathbf{z})) \tag{22.2}$$

where $p(\mathbf{z})$ is some kind of prior on the latent code \mathbf{z} , $d_{\theta}(\mathbf{z})$ is a deep neural network, known as the **decoder**, and $\text{Expfam}(\mathbf{x}|\boldsymbol{\eta})$ is an exponential family distribution, such as a Gaussian or product of Bernoullis. This is called a **deep latent variable model** or **DLVM**. When the prior is Gaussian (as is often the case), this model is called a **deep latent Gaussian model** or **DLGM**.

Posterior inference (i.e., computing $p_{\theta}(\mathbf{z}|\mathbf{x})$) is computationally intractable, as is computing the marginal likelihood

$$p_{\theta}(\mathbf{x}) = \int p_{\theta}(\mathbf{x}|\mathbf{z})p_{\theta}(\mathbf{z}) d\mathbf{z} \tag{22.3}$$

Hence we need to resort to approximate inference. For most of this chapter, we will use **amortized inference**, which we discussed in Section 10.3.7. This trains another model, $q_{\phi}(\mathbf{z}|\mathbf{x})$, called the **recognition network** or **inference network**, simultaneously with the generative model to do approximate posterior inference. This combination is called a **variational autoencoder** or **VAE** [KW14; RMW14b; KW19a], since it can be thought of as a probabilistic version of a deterministic autoencoder.

In this chapter, we introduce the basic VAE, as well as some extensions. Note that the literature on VAE-like methods is vast¹, so we will only discuss a small subset of the ideas that have been explored.

22.2 VAE basics

In this section, we discuss the basics of variational autoencoders.

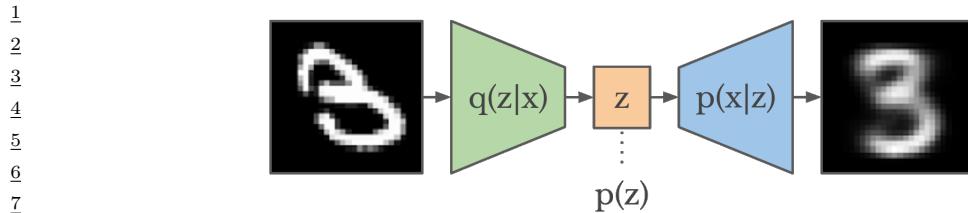


Figure 22.1: Schematic illustration of a VAE. From a figure in [Haf18]. Used with kind permission of Danijar Hafner.

22.2.1 Modeling assumptions

In the simplest setting, a VAE defines a generative model of the form

$$p_{\theta}(z, x) = p_{\theta}(z)p_{\theta}(x|z) \quad (22.4)$$

where $p_{\theta}(z)$ is usually a Gaussian, and $p_{\theta}(x|z)$ is usually a product of exponential family distributions (e.g., Gaussians or Bernoullis), with parameters computed by a neural network decoder, $d_{\theta}(z)$. For example, for binary observations, we can use

$$p_{\theta}(x|z) = \prod_{d=1}^D \text{Ber}(x_d | \sigma(d_{\theta}(z))) \quad (22.5)$$

In addition, a VAE fits a recognition model

$$q_{\phi}(z|x) = q(z|e_{\phi}(x)) \approx p_{\theta}(z|x) \quad (22.6)$$

to perform approximate posterior inference. Here $q_{\phi}(z|x)$ is usually a Gaussian, with parameters computed by a neural network encoder $e_{\phi}(x)$:

$$q_{\phi}(z|x) = \mathcal{N}(z|\mu, \text{diag}(\exp(\ell))) \quad (22.7)$$

$$(\mu, \ell) = e_{\phi}(x) \quad (22.8)$$

where $\ell = \log \sigma$. The model can be thought of as encoding the input x into a stochastic latent bottleneck z and then decoding it to approximately reconstruct the input, as shown in Figure 22.1.

The idea of training an inference network to “invert” a generative network, rather than running an optimization algorithm to infer the latent code, is called amortized inference, and is discussed in Section 10.3.7. This idea was first proposed in the **Helmholtz machine** [Day+95]. However, that paper did not present a single unified objective function for inference and generation, but instead used the wake sleep (Section 22.7) method for training. By contrast, the VAE optimizes a variational lower bound on the log-likelihood, which means that convergence to a locally optimal MLE of the parameters is guaranteed.

We can use other approaches to fitting the DLGM (see e.g., [Hof17; DF19]). However, learning an inference network to fit the DLGM is often faster and can have some regularization benefits (see e.g., [KP20]). Combining a generative model with an inference model in this way results in what Jacob Andreas, in his blog, called a **monference**, i.e., model-inference hybrid.²

⁴⁵ 1. For example, the website <https://github.com/matthewvowels1/Awesome-VAEs> lists over 900 papers.

⁴⁶ 2. See <http://blog.jacobandreas.net/monference.html>.

22.2.2 Evidence lower bound

When fitting the model, our goal is to maximize the marginal likelihood

$$p_{\theta}(\mathbf{x}) = \int p_{\theta}(\mathbf{x}|z) p_{\theta}(z) dz \quad (22.9)$$

Unfortunately, computing this quantity is intractable, because if we could compute it, we could then easily compute the posterior as follows:

$$p_{\theta}(z|\mathbf{x}) = \frac{p_{\theta}(z, \mathbf{x})}{p_{\theta}(\mathbf{x})} \quad (22.10)$$

(Note that the numerator, $p_{\theta}(z, \mathbf{x})$, is always tractable in a directed graphical model.) For a DLVM, $p_{\theta}(z|\mathbf{x})$ is intractable. However, we can use the inference network to compute an approximate posterior, $q_{\phi}(z|\mathbf{x})$, and hence a lower bound to the marginal likelihood. This idea is discussed in Section 10.1.2, but we repeat the argument here, using slightly different notation.

First note that we have the following decomposition:

$$\log p_{\theta}(\mathbf{x}) = \mathbb{E}_{q_{\phi}(z|\mathbf{x})} [\log p_{\theta}(\mathbf{x})] \quad (22.11)$$

$$= \mathbb{E}_{q_{\phi}(z|\mathbf{x})} \left[\log \left(\frac{p_{\theta}(\mathbf{x}, z)}{p_{\theta}(z|\mathbf{x})} \right) \right] \quad (22.12)$$

$$= \mathbb{E}_{q_{\phi}(z|\mathbf{x})} \left[\log \left(\frac{p_{\theta}(\mathbf{x}, z)}{q_{\phi}(z|\mathbf{x})} \frac{q_{\phi}(z|\mathbf{x})}{p_{\theta}(z|\mathbf{x})} \right) \right] \quad (22.13)$$

$$= \underbrace{\mathbb{E}_{q_{\phi}(z|\mathbf{x})} \left[\log \left(\frac{p_{\theta}(\mathbf{x}, z)}{q_{\phi}(z|\mathbf{x})} \right) \right]}_{\mathcal{L}_{\theta, \phi}(\mathbf{x})} + \underbrace{\mathbb{E}_{q_{\phi}(z|\mathbf{x})} \left[\log \left(\frac{q_{\phi}(z|\mathbf{x})}{p_{\theta}(z|\mathbf{x})} \right) \right]}_{D_{\text{KL}}(q_{\phi}(z|\mathbf{x}) \| p_{\theta}(z|\mathbf{x}))} \quad (22.14)$$

The second term in Equation (22.14) is non-negative, and hence

$$\mathcal{L}_{\theta, \phi}(\mathbf{x}) \leq \log p_{\theta}(\mathbf{x}) \quad (22.15)$$

The quantity $\log p_{\theta}(\mathbf{x})$ is the log marginal likelihood, also called the **evidence**. Hence $\mathcal{L}_{\theta, \phi}(\mathbf{x})$ is called the **evidence lower bound** or **ELBO**. Our goal is to *maximize* this quantity. (Thus we use the symbol \mathcal{L} rather than \mathcal{L} , since the latter denotes a loss we want to minimize.)

We can rewrite the ELBO as follows:

$$\mathcal{L}_{\theta, \phi}(\mathbf{x}) = \mathbb{E}_{q_{\phi}(z|\mathbf{x})} [\log p_{\theta}(\mathbf{x}, z) - \log q_{\phi}(z|\mathbf{x})] \quad (22.16)$$

$$= \mathbb{E}_{q_{\phi}(z|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|z) + \log p_{\theta}(z) - \log q_{\phi}(z|\mathbf{x})] \quad (22.17)$$

$$= \mathbb{E}_{q_{\phi}(z|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|z)] - D_{\text{KL}}(q_{\phi}(z|\mathbf{x}) \| p_{\theta}(z)) \quad (22.18)$$

We can interpret this objective as the expected log likelihood plus a regularization term, that ensures the (per-sample) posterior is “well behaved” (does not deviate too far from the prior in terms of KL divergence).

The tightness of this lower bound is controlled by the **variational gap**, which is given by $D_{\text{KL}}(q_{\phi}(z|\mathbf{x}) \| p_{\theta}(z|\mathbf{x}))$. A better approximate posterior results in a tighter bound. When the KL goes to zero, the posterior is exact, so any improvements to the ELBO directly translate to improvements in the likelihood of the data, as in the EM algorithm (see Section 6.7.3).

22.2.3 Optimization

The ELBO for a single datapoint \mathbf{x} is given in Equation (22.18). The ELBO for the whole dataset, scaled by $N = |\mathcal{D}|$, the number of examples, is given by

$$\mathbb{L}_{\theta, \phi}(\mathcal{D}) = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{D}} \mathbb{L}_{\theta, \phi}(\mathbf{x}) = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{D}} [\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))] \quad (22.19)$$

Our goal is to *maximize* this wrt θ and ϕ using stochastic gradient ascent. Alternatively, we can try to *minimize* the **negative ELBO**, also called the **variational free energy**:

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{D}} [\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [-\log p_\theta(\mathbf{x}|\mathbf{z})] + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))] \quad (22.20)$$

We can create an unbiased minibatch approximation of this objective by sampling examples \mathbf{x} , and then computing the objective for a given \mathbf{x} . So now we focus on a fixed \mathbf{x} , for brevity.

If we assume $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$ and $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}))$ we can use Equation (5.67) to compute the KL in closed form:

$$D_{\text{KL}}(q||p) = -\frac{1}{2} \sum_{k=1}^K [\log \sigma_k^2 - \sigma_k^2 - \mu_k^2 + 1] \quad (22.21)$$

This just leaves us with the expected log likelihood term. We can approximate this by sampling $\mathbf{z}^s \sim q_\phi(\mathbf{z}|\mathbf{x})$, to get

$$\mathbb{L}_{\theta, \phi}(\mathbf{x}) = \log p_\theta(\mathbf{x}|\mathbf{z}^s) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) \quad (22.22)$$

It is easy to take gradients of this wrt θ , using automatic differentiation. Unfortunately taking gradients wrt ϕ is harder, since we need to take into account that the sampling process itself depends on ϕ . We discuss a solution to this in Section 22.2.4.

22.2.4 The reparameterization trick

In this section, we discuss how to compute gradients of the (single sample) ELBO

$$\mathbb{L}_{\theta, \phi}(\mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\theta(\mathbf{z}) - q_\phi(\mathbf{z}|\mathbf{x})] \quad (22.23)$$

(We write it this way, rather than in terms of a KL penalty, since for non-Gaussian distributions, the KL will be hard to compute.)

The gradient wrt the generative parameters θ is easy to compute, since we can push gradients inside the expectation, and use a single Monte Carlo sample:

$$\nabla_\theta \mathbb{L}_{\theta, \phi}(\mathbf{x}) = \nabla_\theta \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})] \quad (22.24)$$

$$= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\nabla_\theta \{\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\}] \quad (22.25)$$

$$\approx \nabla_\theta \log p_\theta(\mathbf{x}, \mathbf{z}^s) \quad (22.26)$$

where $\mathbf{z}^s \sim q_\phi(\mathbf{z}|\mathbf{x})$. This is an unbiased estimate of the gradient, so can be used with SGD.

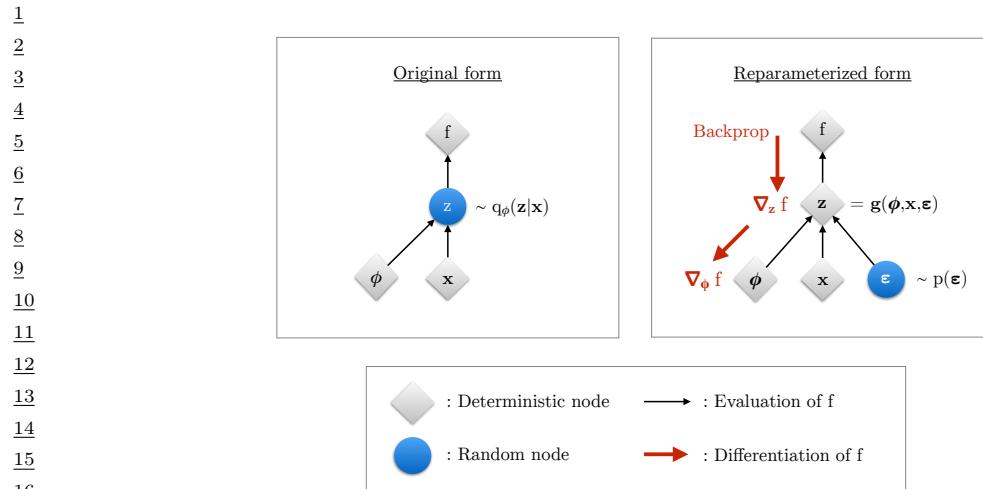


Figure 22.2: Illustration of the reparameterization trick. The objective f depends on the variational parameters ϕ , the observed data x , and the latent random variable $z \sim q_\phi(z|x)$. On the left, we show the standard form of the computation graph. On the right, we show a reparameterized form, in which we move the stochasticity into the noise source ϵ , and compute z deterministically, $z = f(\phi, x, \epsilon)$. The rest of the graph is deterministic, so we can backpropagate the gradient of the scalar f wrt ϕ through z and into ϕ . From Figure 2.3 of [KW19a]. Used with kind permission of Durk Kingma.

The gradient wrt the inference parameters ϕ is harder to compute since

$$\nabla_\phi \mathbb{L}_{\theta, \phi}(x) = \nabla_\phi \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x, z) - \log q_\phi(z|x)] \quad (22.27)$$

$$\neq \mathbb{E}_{q_\phi(z|x)} [\nabla_\phi \{\log p_\theta(x, z) - \log q_\phi(z|x)\}] \quad (22.28)$$

However, for continuous latent variables z , we can use the **reparameterization trick** to make the randomness independent of ϕ , which lets us pass gradients through. We explain this in detail in Section 6.6.4, but we summarize the basic idea here.

The key trick is to rewrite the random variable $z \sim q_\phi(z|x)$ as some differentiable (and invertible) transformation r of another random variable $\epsilon \sim p(\epsilon)$, which does not depend on ϕ , i.e., we assume we can write

$$z = r(\epsilon, \phi, x) \quad (22.29)$$

For example,

$$z \sim \mathcal{N}(\mu, \text{diag}(\sigma)) \iff z = \mu + \epsilon \odot \sigma, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (22.30)$$

Using this, we have

$$\mathbb{E}_{q_\phi(z|x)} [f(z)] = \mathbb{E}_{p(\epsilon)} [f(z)] \quad \text{s.t. } z = r(\epsilon, \phi, x) \quad (22.31)$$

where we define

$$f(z) = \log p_\theta(x, z) - \log q_\phi(z|x) \quad (22.32)$$

1
2 Hence

3 $\nabla_{\phi} \mathbb{E}_{q_{\phi}(z|x)} [f(z)] = \nabla_{\phi} \mathbb{E}_{p(\epsilon)} [f(z)] = \mathbb{E}_{p(\epsilon)} [\nabla_{\phi} f(z)]$ (22.33)
4

5 which we can approximate with a single Monte Carlo sample. This lets us propagate gradients back
6 through the f function and then into the DNN transformation function r that is used to compute
7 $z = r(\epsilon, \phi, x)$. See Figure 22.2 for an illustration.
8

9 22.2.5 Computing the reparameterized ELBO

10
11 Since we are now working with the random variable ϵ , we need to use the change of variables formula
12 to compute

13
14 $\log q_{\phi}(z|x) = \log p(\epsilon) - \log \left| \det \left(\frac{\partial z}{\partial \epsilon} \right) \right|$ (22.34)
15

16 where $\frac{\partial z}{\partial \epsilon}$ is the Jacobian:
17

18
19 $\frac{\partial z}{\partial \epsilon} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \dots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \dots & \frac{\partial z_k}{\partial \epsilon_k} \end{pmatrix}$ (22.35)
20
21

22 We design the transformation $z = r(\epsilon)$ such that this Jacobian is tractable to compute. We give
23 some examples below.
24

25 22.2.5.1 Fully factorized Gaussian

26 Suppose we have a fully factorized Gaussian posterior:

27 $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (22.36)
28

29 $z = \mu + \sigma \odot \epsilon$ (22.37)
30

31 $(\mu, \log \sigma) = e_{\phi}(x)$ (22.38)
32

33 Then the Jacobian is

34 $\frac{\partial z}{\partial \epsilon} = \text{diag}(\sigma)$ (22.39)
35

36 so

37
38 $\log q_{\phi}(z|x) = \sum_{k=1}^K \log \mathcal{N}(\epsilon_k | 0, 1) - \log \sigma_k = \sum_{k=1}^K -\frac{1}{2} \log(2\pi) - \frac{1}{2} \epsilon_k^2 - \log \sigma_k$ (22.40)
39
40

41 22.2.5.2 Full covariance Gaussian

42 Now consider a full covariance Gaussian posterior:

43 $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (22.41)
44

45 $z = \mu + \mathbf{L}\epsilon$ (22.42)
46

47

where \mathbf{L} is a lower triangular matrix with non-zero entries on the diagonal, which satisfies

$$\Sigma = \mathbf{L}\mathbf{L}^\top \quad (22.43)$$

The Jacobian of this affine transformation is

$$\frac{\partial \mathbf{z}}{\partial \epsilon} = \mathbf{L} \quad (22.44)$$

Since \mathbf{L} is a triangular matrix, its determinant is the product of its diagonals, so

$$\log |\det \frac{\partial \mathbf{z}}{\partial \epsilon}| = \sum_{k=1}^K \log |L_{kk}| \quad (22.45)$$

We need to make the parameters of the transformation r be a function of the inputs \mathbf{x} . One way to do this is to define

$$(\mu, \log \sigma, \mathbf{L}') = e_\phi(\mathbf{x}) \quad (22.46)$$

$$\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\sigma) \quad (22.47)$$

where \mathbf{M} is a masking matrix with 0s on and above the diagonal, and 1s below the diagonal. With this construction, the diagonal entries of \mathbf{L} are given by σ , so

$$\log |\det \frac{\partial \mathbf{z}}{\partial \epsilon}| = \sum_{k=1}^K \log |L_{kk}| = \sum_{k=1}^K \log \sigma_k \quad (22.48)$$

See Algorithm 26 for the corresponding pseudo code for computing the reparameterized ELBO.

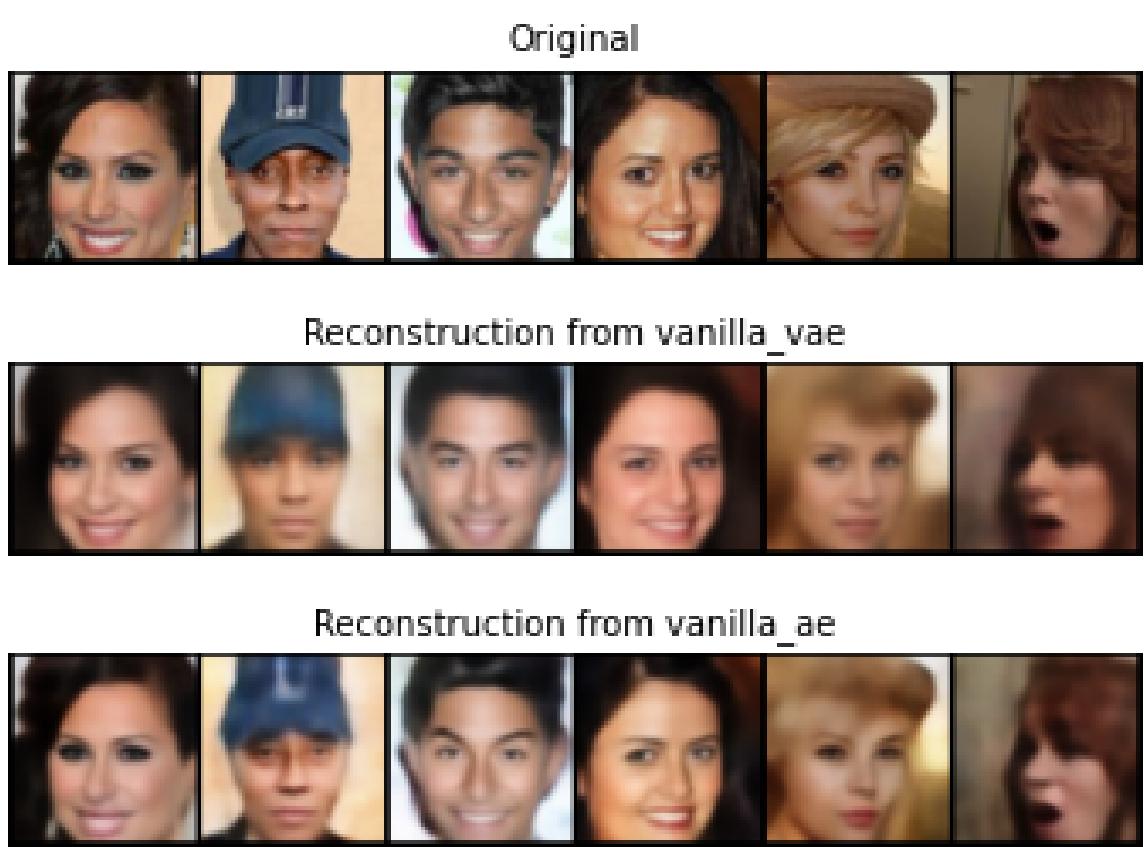
Algorithm 26: Computing a single sample unbiased estimate of the reparameterized ELBO for a VAE with full covariance Gaussian posterior, and factorized Bernoulli likelihood. Based on Algorithm 2 of [KW19a].

```

1  $(\mu, \log \sigma, \mathbf{L}') = e_\phi(\mathbf{x})$  ;
2  $\mathbf{M} = \text{np.triu}(\text{np.ones}(K), -1)$  ;
3  $\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\sigma)$  ;
4  $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  ;
5  $\mathbf{z} = \mathbf{L}\epsilon + \mu$  ;
6  $\mathbf{p} = d_\theta(\mathbf{z})$  ;
7  $\mathcal{L}_{\log qz} = -\sum_{k=1}^K [\frac{1}{2}\epsilon_k^2 + \frac{1}{2}\log(2\pi) + \log \sigma_k]$  // from  $q_\phi(\mathbf{z}|\mathbf{x})$  ;
8  $\mathcal{L}_{\log pz} = -\sum_{k=1}^K [\frac{1}{2}z_k^2 + \frac{1}{2}\log(2\pi)]$  // from  $p_\theta(\mathbf{z})$  ;
9  $\mathcal{L}_{\log px} = -\sum_{d=1}^D [x_d \log p_d + (1-x_d) \log(1-p_d)]$  // from  $p_\theta(\mathbf{x}|\mathbf{z})$  ;
10  $\mathcal{L} = \mathcal{L}_{\log px} + \mathcal{L}_{\log pz} - \mathcal{L}_{\log qz}$ 
```

22.2.5.3 Inverse autoregressive flows

In Section 10.4.3, we discuss how to use inverse autoregressive flows to learn more expressive posteriors $q_\phi(\mathbf{z}|\mathbf{x})$, leveraging the tractability of the Jacobian of this nonlinear transformation.



22.2.6 Comparison of VAEs and autoencoders

VAEs are very similar to deterministic autoencoders (DAE). In particular, the generative model, $p_{\theta}(\mathbf{x}|\mathbf{z})$ acts like the decoder, and the inference network, $q_{\phi}(\mathbf{z}|\mathbf{x})$, acts like the encoder. To illustrate this, we fit a a convolutional VAE and DAE to the **CelebA** dataset [Liu+15].³ In Figure 22.3, we see that both DAE and VAE can reconstruct the input images reasonably well, although the VAE reconstructions are somewhat more blurry, for reasons we discuss in Section 22.3.1.

The main advantage of a VAE over a deterministic autoencoder is that it defines a proper generative model, that can create sensible-looking novel images by decoding prior samples $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. By contrast, an autoencoder only knows how to decode latent codes derived from the training set, not

³ 3. CelebA contains about 200k images of famous celebrities. The images are also annotated with 40 attributes. We reduce the resolution of the images to 64x64, as is conventional.

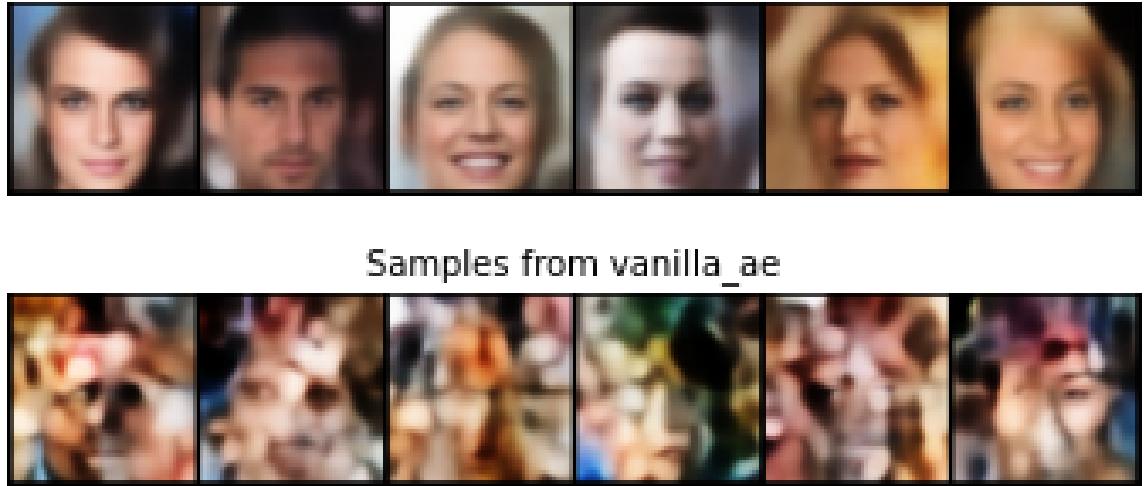


Figure 22.4: Illustration of image sampling. (a) Variational autoencoder. (b) Deterministic autoencoder. Generated by [vae_compare_results.ipynb](#).

novel samples. This is illustrated in Figure 22.4.

22.2.7 VAEs optimize in an augmented space

In this section, we derive several alternative expressions for the ELBO which shed light on how VAEs work.

First, let us define the joint generative distribution

$$p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{z})p_{\theta}(\mathbf{x}|\mathbf{z}) \quad (22.49)$$

and the joint inference distribution

$$q_{\phi}(\mathbf{z}, \mathbf{x}) = p_{\mathcal{D}}(\mathbf{x})q_{\phi}(\mathbf{z}|\mathbf{x}) \quad (22.50)$$

where

$$p_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^N \delta(\mathbf{x}_n - \mathbf{x}) \quad (22.51)$$

is the empirical distribution. Let us also define the data marginal

$$p_{\theta}(\mathbf{x}) = \int_{\mathbf{z}} p_{\theta}(\mathbf{x}, \mathbf{z}) d\mathbf{z} \quad (22.52)$$

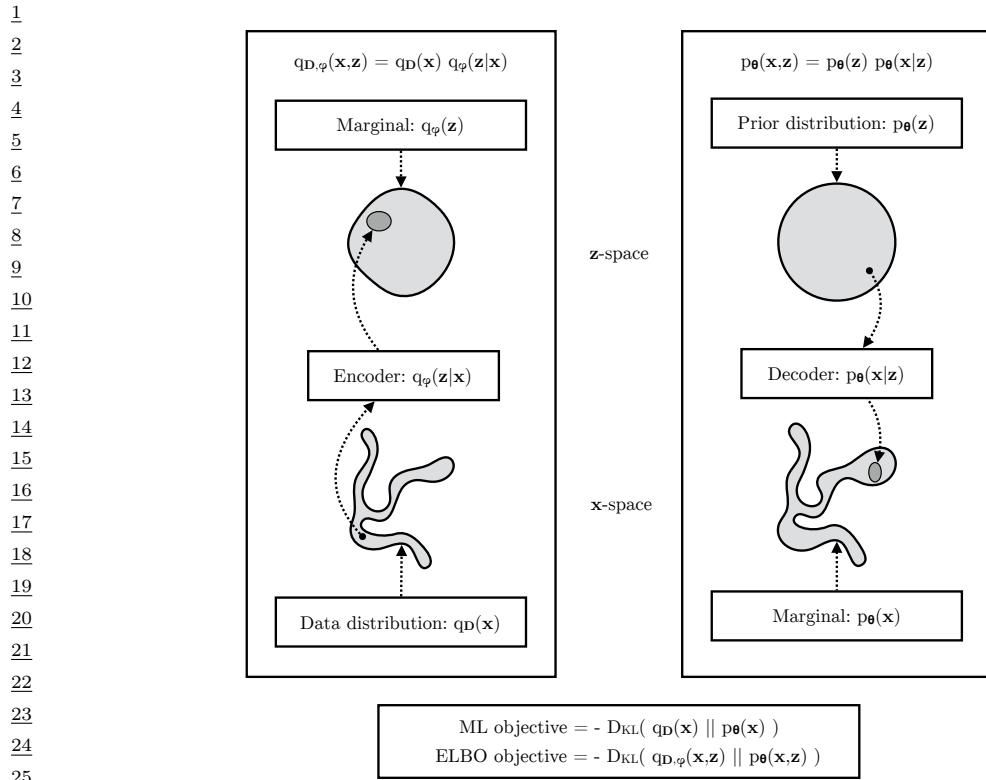


Figure 22.5: The maximum likelihood (ML) objective can be viewed as the minimization of $D_{\text{KL}}(p_D(x) \parallel p_\theta(x))$, while the ELBO objective is minimization of $D_{\text{KL}}(q_{D,\phi}(x,z) \parallel p_\theta(x,z))$, which upper bounds $D_{\text{KL}}(q_D(x) \parallel p_\theta(x))$. From Figure 2.4 of [KW19a]. Used with kind permission of Durk Kingma.

and the inference marginal, also called the **aggregated posterior**:

$$q_\phi(z) = \int_x q_\phi(x, z) dx \quad (22.53)$$

Finally, we define the conditionals $p_\theta(z|x) = p_\theta(x, z)/p_\theta(x)$ and $q_\phi(x|z) = q_\phi(x, z)/q_\phi(z)$. See Figure 22.5 for a visual illustration.

Having defined our terms, we can now derive various alternative versions of the ELBO, following [ZSE19]. First note that

$$\mathcal{L} = \mathbb{E}_{p_D(x)} [\mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)]] - \mathbb{E}_{p_D(x)} [D_{\text{KL}}(q_\phi(z|x) \parallel p_\theta(z))] \quad (22.54)$$

$$= \mathbb{E}_{q_\phi(x,z)} [\log p_\theta(x|z) + \log p_\theta(z) - \log q_\phi(z|x)] \quad (22.55)$$

$$= \mathbb{E}_{q_\phi(x,z)} \left[\log \frac{p_\theta(x, z)}{q_\phi(x, z)} + \log p_D(x) \right] \quad (22.56)$$

$$= -D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) + \mathbb{E}_{p_D(x)} [\log p_D(x)] \quad (22.57)$$

If we define $\stackrel{c}{=}$ to mean equal up to additive constants, we can rewrite the above as

$$\mathcal{L} \stackrel{c}{=} -D_{\text{KL}}(q_{\phi}(\mathbf{x}, \mathbf{z}) \| p_{\theta}(\mathbf{x}, \mathbf{z})) \quad (22.58)$$

Now note that, from the chain rule for KL divergence,

$$D_{\text{KL}}(q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z}) \| p_{\theta}(\mathbf{x}, \mathbf{z})) = D_{\text{KL}}(q_{\mathcal{D}}(\mathbf{x}) \| p_{\theta}(\mathbf{x})) + \mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}[D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p_{\theta}(\mathbf{z}|\mathbf{x}))] \quad (22.59)$$

Hence

$$\mathcal{L} \stackrel{c}{=} -D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\theta}(\mathbf{x})) - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p_{\theta}(\mathbf{z}|\mathbf{x}))] \quad (22.60)$$

Thus maximizing the ELBO requires minimizing the two KL terms. The first KL term is minimized by MLE, and the second KL term is minimized by fitting the true posterior. Thus if the posterior family is limited, there may be a conflict between these objectives.

Finally, we note that the ELBO can also be written as

$$\mathcal{L} \stackrel{c}{=} -D_{\text{KL}}(q_{\phi}(\mathbf{z}) \| p_{\theta}(\mathbf{z})) - \mathbb{E}_{q_{\phi}(\mathbf{z})}[D_{\text{KL}}(q_{\phi}(\mathbf{x}|\mathbf{z}) \| p_{\theta}(\mathbf{x}|\mathbf{z}))] \quad (22.61)$$

We see from Equation (22.61) that VAEs are trying to minimize both the aggregated posterior $D_{\text{KL}}(q_{\phi}(\mathbf{z}) \| p_{\theta}(\mathbf{z}))$ and the expected likelihood $D_{\text{KL}}(q_{\phi}(\mathbf{x}|\mathbf{z}) \| p_{\theta}(\mathbf{x}|\mathbf{z}))$. Since \mathbf{x} is typically of much higher dimensionality than \mathbf{z} , the latter term usually dominates. Consequently, if there is a conflict between these two objectives (e.g., due to limited modeling power), the VAE will favor reconstruction accuracy over posterior inference. Thus the learned posterior may not be a very good approximation to the true posterior (see [ZSE19] for further discussion).

22.3 VAE generalizations

In this section, we discuss some variants of the basic VAE model.

22.3.1 σ -VAE

It is often the case that VAEs generate somewhat blurry images, as illustrated in Figure 22.3, Figure 22.4 and Figure 21.6. This is not the case for models that optimize the exact likelihood, such as pixelCNNs (Section 23.3.2) and flow models (Chapter 24). To see why VAEs are different, consider the VAE objective:

$$\mathcal{L} = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})]] - R(q_{\phi}) \quad (22.62)$$

where $R()$ is the KL regularizer. Now consider the common case where the decoder is a Gaussian with fixed variance, so

$$\log p_{\theta}(\mathbf{x}|\mathbf{z}) = -\frac{1}{2\sigma^2}\|\mathbf{x} - d_{\theta}(\mathbf{z})\|_2^2 \quad (22.63)$$

Let $e_{\phi}(\mathbf{x}) = \mathbb{E}[q_{\phi}(\mathbf{z}|\mathbf{x})]$ be the encoding of \mathbf{x} , and $\mathcal{X}(\mathbf{z}) = \{\mathbf{x} : e_{\phi}(\mathbf{x}) = \mathbf{z}\}$ be the set of inputs that get mapped to \mathbf{z} . For a fixed inference network, the optimal setting of the generator parameters is to

1 ensure $d_{\theta}(\mathbf{z}) = \mathbb{E}[\mathbf{x} : \mathbf{x} \in \mathcal{X}(\mathbf{z})]$. Thus the decoder should predict the average of all inputs \mathbf{x} that
2 map to that \mathbf{z} . So unless the encoder is lossless, the outputs will be blurry.
3

4 We can solve this problem by increasing the expressive power of the posterior approximation
5 (avoiding the merging of distinct inputs into the same latent code), or of the generator (by adding
6 back information that is missing from the latent code), or both. However, an even simpler solution is
7 to optimize the noise variance σ^2 , which controls the degree of blurring.

8 To do this, consider a VAE with a Gaussian decoder with mean $\hat{\mathbf{x}} = d_{\theta}(\mathbf{z})$ and spherical covariance
9 $\sigma^2 \mathbf{I}$. The corresponding negative log likelihood has the form

$$\begin{aligned} \underline{11} \quad -\log p(\mathbf{x}|\mathbf{z}) &= \frac{D}{2\sigma^2} \text{mse}(d_{\theta}(\mathbf{z}), \mathbf{x}) + D \log \sqrt{2\pi} \end{aligned} \quad (22.64)$$

12 where

$$\begin{aligned} \underline{15} \quad \text{mse}(\hat{\mathbf{x}}, \mathbf{x}) &= \frac{1}{D} \sum_{d=1}^D (\hat{x}_d - x_d)^2 \end{aligned} \quad (22.65)$$

18 is the mean squared error. Hence the negative ELBO is given by

$$\begin{aligned} \underline{20} \quad \mathcal{L}(\theta, \phi, \sigma) &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} \left[\frac{D}{2\sigma^2} \text{mse}(d_{\theta}(\mathbf{z}), \mathbf{x}) \right] + D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p_{\theta}(\mathbf{z})) \right] \end{aligned} \quad (22.66)$$

22 We can learn the parameter σ just like we learn the other parameters, by minimizing the above
23 objective. However, we can sometimes get better results, and faster training, if we first fit θ and ϕ ,
24 and then estimate the optimal σ based on the mean of the residual squared errors, just like we do for
25 linear regression, i.e.,
26

$$\begin{aligned} \underline{27} \quad \sigma^* &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [\text{mse}(d_{\theta}(\mathbf{z}), \mathbf{x})]] \end{aligned} \quad (22.67)$$

29 (In practice, the inner expectation is approximated with a single latent sample, and the outer
30 expectation is approximated using the current minibatch sample.) This method is called the σ -
31 VAE [RDL21], and can (sometimes) result in improved sample quality (compare top two rows of
32 Figure 22.6).

33 We can also use this approach to estimate a diagonal covariance matrix, which associates one
34 variance term per pixel (and per color channel). However, this can easily overfit. To see why, consider
35 an image with mostly black or white pixels; we can set the corresponding variance terms to 0 and
36 drive the log likelihood to infinity, since we are modeling a constant value with a delta function. This
37 pathology can be avoided by tying the variance parameter across output dimensions.
38

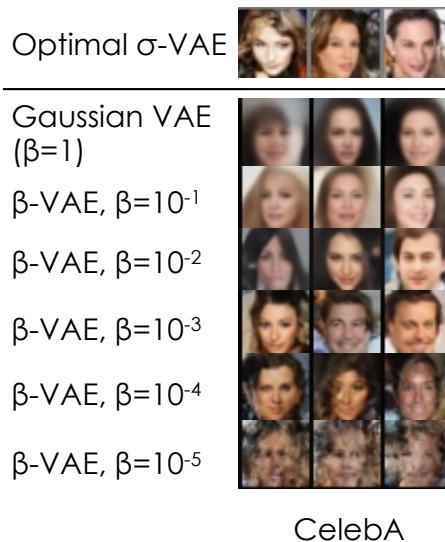
39 22.3.2 β -VAE

41 The negative ELBO loss (which we want to minimize) can be written as follows:

$$\begin{aligned} \underline{43} \quad \mathcal{L}(\theta, \phi | \mathbf{x}) &= \underbrace{-\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})]}_{\mathcal{L}_E} + \underbrace{D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p_{\theta}(\mathbf{z}))}_{\mathcal{L}_R} \end{aligned} \quad (22.68)$$

45 where \mathcal{L}_E is the reconstruction error (negative log likelihood), and \mathcal{L}_R is the KL regularizer.

47



CelebA

Figure 22.6: Comparison of σ -VAE (top row), standard VAE (second row), and β -VAE (remaining rows). The model is a convolutional hierarchical VAE (Section 22.5) with Gaussian prior and Gaussian likelihood. From Figure 2 of [RDL21]. Used with kind permission of Oleh Rybkin.

It is natural to consider a generalization, in which we weight the KL term by an adjustable constant, $\beta > 0$. This gives the **β -VAE** objective [Hig+17]:

$$\mathcal{L}_\beta = \mathcal{L}_E + \beta \mathcal{L}_R \quad (22.69)$$

If we set $\beta = 1$, we recover the objective used in standard VAEs; if we set $\beta = 0$, we recover the objective used in standard autoencoders.

By varying β from 0 to infinity, we can reach different points on the **rate distortion curve**, as discussed in Section 5.4.2. These points make different tradeoffs between reconstruction error (distortion) and how much information is stored in the latents about the input (rate of the corresponding code).

22.3.2.1 Connection with σ -VAE

The β -VAE is usually combined with a Gaussian decoder with unit variance. The loss function in this case becomes

$$\mathcal{L}_\beta = \frac{D}{2} \text{mse}(\hat{\mathbf{x}}, \mathbf{x}) + \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{x})) \quad (22.70)$$

We see that this is a special case of the σ -VAE from Section 22.3.1 where $\sigma^2 = \beta/2$. It is common to tune the β parameter in order to get good-looking samples, but obviously we could just tune σ^2 instead, which often works better, as shown in Figure 22.6.

1 **22.3.2.2 Disentangled representations**

3 One advantage of using $\beta > 1$ is that it encourages the learning of a latent representation that is
4 “**disentangled**”. Intuitively this means that each latent dimension represents a different **factor of**
5 **variation** in the input. This is often formalized in terms of the total correlation (Section 5.3.5.1),
6 which is defined as follows:

7

$$\text{TC}(\mathbf{z}) = \sum_k \mathbb{H}(z_k) - \mathbb{H}(\mathbf{z}) = D_{\text{KL}} \left(p(\mathbf{z}) \middle\| \prod_k p_k(z_k) \right) \quad (22.71)$$

8

9 This is zero iff the components of \mathbf{z} are all mutually independent, and hence disentangled. In [AS18],
10 they prove that using $\beta > 1$ will decrease the TC.

11 Unfortunately, in [Loc+18] they prove that nonlinear latent variable models are unidentifiable, and
12 therefore for any disentangled representation, there is an equivalent fully entangled representation
13 with exactly the same likelihood. Thus it is not possible to recover the correct latent representation
14 without choosing the appropriate inductive bias, via the encoder, decoder, prior, dataset, or learning
15 algorithm, i.e., merely adjusting β is not sufficient.

16

17 **22.3.2.3 Connection with information bottleneck**

18 In this section, we show that the β -VAE is an unsupervised version of the information bottleneck
19 (IB) objective from Section 5.6. If the input is \mathbf{x} , the hidden bottleneck is \mathbf{z} , and the target outputs
20 are $\tilde{\mathbf{x}}$, then the unsupervised IB objective becomes

21

$$\mathcal{L}_{\text{UIB}} = \beta \mathbb{I}(\mathbf{z}; \mathbf{x}) - \mathbb{I}(\mathbf{z}; \tilde{\mathbf{x}}) \quad (22.72)$$

22

$$= \beta \mathbb{E}_{p(\mathbf{x}, \mathbf{z})} \left[\log \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{x})p(\mathbf{z})} \right] - \mathbb{E}_{p(\mathbf{z}, \tilde{\mathbf{x}})} \left[\log \frac{p(\mathbf{z}, \tilde{\mathbf{x}})}{p(\mathbf{z})p(\tilde{\mathbf{x}})} \right] \quad (22.73)$$

23

24 where

25

$$p(\mathbf{x}, \mathbf{z}) = p_{\mathcal{D}}(\mathbf{x})p(\mathbf{z}|\mathbf{x}) \quad (22.74)$$

26

$$p(\mathbf{z}, \tilde{\mathbf{x}}) = \int p_{\mathcal{D}}(\mathbf{x})p(\mathbf{z}|\mathbf{x})p(\tilde{\mathbf{x}}|\mathbf{z})d\mathbf{x} \quad (22.75)$$

27

28 Intuitively, the objective in Equation (22.72) means we should pick a representation \mathbf{z} that can
29 predict $\tilde{\mathbf{x}}$ reliably, while not memorizing too much information about the input \mathbf{x} . The tradeoff
30 parameter is controlled by β .

31 From Equation (5.163), we have the following variational upper bound on this unsupervised
32 objective:

33

$$\mathcal{L}_{\text{UVIB}} = -\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})q_{\phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})] + \beta \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \| p_{\theta}(\mathbf{z}))] \quad (22.76)$$

34

35 which matches Equation (22.69) when averaged over \mathbf{x} .

36

37 **22.3.3 InfoVAE**

38

39 In Section 22.2.7, we discussed some drawbacks of the standard ELBO objective for training VAEs,
40 namely the tendency to ignore the latent code when the decoder is powerful (Section 22.4), and the
41

tendency to learn a poor posterior approximation due to the mismatch between the KL terms in data space and latent space (Section 22.2.7). We can fix these problems to some degree by using a generalized objective of the following form:

$$\mathcal{L}(\theta, \phi | \mathbf{x}) = -\lambda D_{\text{KL}}(q_\phi(\mathbf{z}) \| p_\theta(\mathbf{z})) - \mathbb{E}_{q_\phi(\mathbf{z})} [D_{\text{KL}}(q_\phi(\mathbf{x}|\mathbf{z}) \| p_\theta(\mathbf{x}|\mathbf{z}))] + \alpha \mathbb{I}_q(\mathbf{x}; \mathbf{z}) \quad (22.77)$$

where $\alpha \geq 0$ controls much we weight the mutual information $\mathbb{I}_q(\mathbf{x}; \mathbf{z})$ between \mathbf{x} and \mathbf{z} , and $\lambda \geq 0$ controls the tradeoff between \mathbf{z} -space KL and \mathbf{x} -space KL. This is called the **InfoVAE** objective [ZSE19]. If we set $\alpha = 0$ and $\lambda = 1$, we recover the standard ELBO, as shown in Equation (22.61).

Unfortunately, the objective in Equation (22.77) cannot be computed as written, because of the intractable MI term:

$$\mathbb{I}_q(\mathbf{x}; \mathbf{z}) = \mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} \left[\log \frac{q_\phi(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{x})q_\phi(\mathbf{z})} \right] = \mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} \left[\log \frac{q_\phi(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right] \quad (22.78)$$

However, using the fact that $q_\phi(\mathbf{x}|\mathbf{z}) = p_{\mathcal{D}}(\mathbf{x})q_\phi(\mathbf{z}|\mathbf{x})/q_\phi(\mathbf{z})$, we can rewrite the objective as follows:

$$\mathcal{L} = \mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} \left[-\lambda \log \frac{q_\phi(\mathbf{z})}{p_\theta(\mathbf{z})} - \log \frac{q_\phi(\mathbf{x}|\mathbf{z})}{p_\theta(\mathbf{x}|\mathbf{z})} - \alpha \log \frac{q_\phi(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right] \quad (22.79)$$

$$= \mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} \left[\log p_\theta(\mathbf{x}|\mathbf{z}) - \log \frac{q_\phi(\mathbf{z})^{\lambda+\alpha-1} p_{\mathcal{D}}(\mathbf{x})}{p_\theta(\mathbf{z})^\lambda q_\phi(\mathbf{z}|\mathbf{x})^{\alpha-1}} \right] \quad (22.80)$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z})]] - (1 - \alpha) \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))] \\ - (\alpha + \lambda - 1) D_{\text{KL}}(q_\phi(\mathbf{z}) \| p_\theta(\mathbf{z})) - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\log p_{\mathcal{D}}(\mathbf{x})] \quad (22.81)$$

where the last term is a constant we can ignore. The first two terms can be optimized using the reparameterization trick. Unfortunately, the last term requires computing $q_\phi(\mathbf{z}) = \int_{\mathbf{x}} q_\phi(\mathbf{x}, \mathbf{z}) d\mathbf{x}$, which is intractable. Fortunately, we can easily sample from this distribution, by sampling $\mathbf{x} \sim p_{\mathcal{D}}(\mathbf{x})$ and $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. Thus $q_\phi(\mathbf{z})$ is an **implicit probability model**, similar to a GAN (see Chapter 27).

As long as we use a strict divergence, meaning $D(q, p) = 0$ iff $q = p$, then one can show that this does not affect the optimality of the procedure. In particular, proposition 2 of [ZSE19] tells us the following:

Theorem 1. Let \mathcal{X} and \mathcal{Z} be continuous spaces, and $\alpha < 1$ (to bound the MI) and $\lambda > 0$. For any fixed value of $\mathbb{I}_q(\mathbf{x}; \mathbf{z})$, the approximate InfoVAE loss, with any strict divergence $D(q_\phi(\mathbf{z}), p_\theta(\mathbf{z}))$, is globally optimized if $p_\theta(\mathbf{x}) = p_{\mathcal{D}}(\mathbf{x})$ and $q_\phi(\mathbf{z}|\mathbf{x}) = p_\theta(\mathbf{z}|\mathbf{x})$.

22.3.3.1 Connection with MMD VAE

If we set $\alpha = 1$, the InfoVAE objective simplifies to

$$\mathcal{L} \stackrel{c}{=} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z})]] - \lambda D_{\text{KL}}(q_\phi(\mathbf{z}) \| p_\theta(\mathbf{z})) \quad (22.82)$$

The **MMD VAE**⁴ replaces the KL divergence in the above term with the (squared) maximum mean discrepancy or **MMD** divergence defined in Section 2.9.3. (This is valid based on the above theorem.)

4. Proposed in <https://ermongroup.github.io/blog/a-tutorial-on-mmd-variational-autoencoders/>.

1 The advantage of this approach over standard InfoVAE is that the resulting objective is tractable. In
2 particular, if we set $\lambda = 1$ and swap the sign we get
3

$$\underline{4} \quad \mathcal{L} = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [-\log p_{\theta}(\mathbf{x}|\mathbf{z})]] + \text{MMD}(q_{\phi}(\mathbf{z}), p_{\theta}(\mathbf{z})) \quad (22.83)$$

5 As we discuss in Section 2.9.3, we can compute the MMD as follows:
6

$$\underline{7} \quad \text{MMD}(p, q) = \mathbb{E}_{p(\mathbf{z}), p(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] + \mathbb{E}_{q(\mathbf{z}), q(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] - 2\mathbb{E}_{p(\mathbf{z}), q(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] \quad (22.84)$$

8 where $\mathcal{K}()$ is some kernel function, such as the RBF kernel, $\mathcal{K}(\mathbf{z}, \mathbf{z}') = \exp(-\frac{1}{2\sigma^2} \|\mathbf{z} - \mathbf{z}'\|_2^2)$. Intuitively
9 the MMD measures the similarity (in latent space) between samples from the prior and samples from
10 the aggregated posterior.
11

12 In practice, we can implement the MMD objective by using the posterior predicted mean $\mathbf{z}_n =$
13 $e_{\phi}(\mathbf{x}_n)$ for all B samples in the current minibatch, and comparing this to B random samples from
14 the $\mathcal{N}(\mathbf{0}, \mathbf{I})$ prior.

15 If we use a Gaussian decoder with fixed variance, the negative log likelihood is just a squared error
16 term:

$$\underline{17} \quad -\log p_{\theta}(\mathbf{x}|\mathbf{z}) = \|\mathbf{x}_n - d_{\theta}(\mathbf{z}_n)\|_2^2 \quad (22.85)$$

18 Thus the entire model is deterministic, and just predicts the means in latent space and visible space.
19

21 22.3.3.2 Connection with β -VAEs

22 If we set $\alpha = 0$ and $\lambda = 1$, we get back the original ELBO. If $\lambda > 0$ is freely chosen, but we use
23 $\alpha = 1 - \lambda$, we get the β -VAE.
24

25 22.3.3.3 Connection with adversarial autoencoders

26 If we set $\alpha = 1$ and $\lambda = 1$, and D is chosen to be the Jensen Shannon divergence (which can be
27 minimized by training a binary discriminator, as explained in Section 27.2.2), then we get a model
28 known as an **adversarial autoencoder** [Mak+15a].
29

30

31 22.3.3.4 Example

32 In this section, we give a simple example of InfoVAE (MMD version) applied to MNIST. We use
33 just $L = 2$ latent dimensions, so we can plot the latent space. We use the squared MMD divergence
34 (Section 2.9.3) as an alternative to $D_{\text{KL}}(q_{\phi}(\mathbf{z})\|p_{\theta}(\mathbf{z}))$. We implement MMD using an RBF kernel,
35 using the bandwidth parameter $\sigma^2 = 2/L$.⁵ The neural architecture is an MLP with one hidden
36 layer. In Figure 22.7(a), we show the result of fitting this model using the standard ELBO. In
37 Figure 22.7(b), we show the result of fitting this model using the InfoVAE loss, with $\alpha = 1$ and
38 $\alpha + \lambda - 1 = 1$. We see that the latent space for the InfoVAE model shows better class separation,
39 suggesting that it has captured more of the “semantics” of the data.
40

41

42 22.3.4 Multi-modal VAEs

43 It is possible to extend VAEs to create joint distributions over different kinds of variables, such as
44 images and text. This is sometimes called a **multimodal VAE** or **MVAE**. Let us assume there are
45

46 5. This is the hyperparameter recommended in the MMD blog post.
47

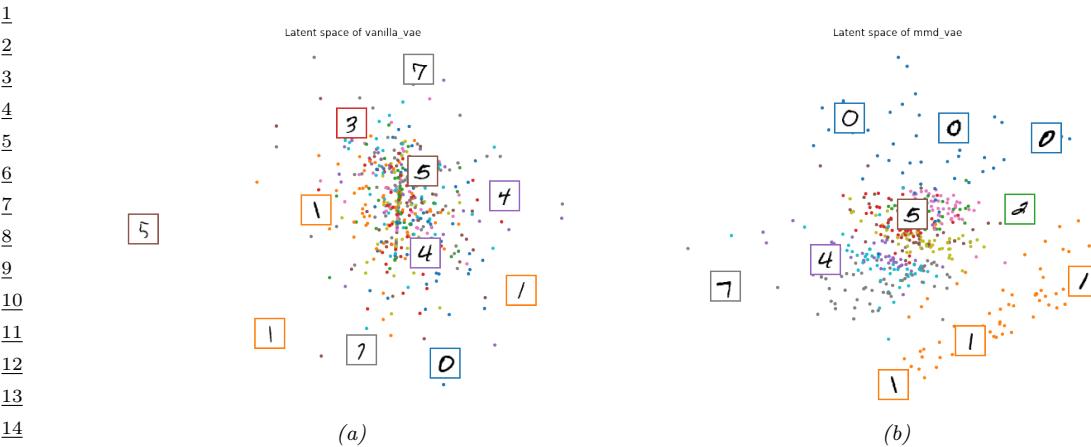


Figure 22.7: Illustration of 2d embedding space for (a) VAE and (b) MMD-VAE fit to MNIST. Generated by `vae_latent_space.ipynb`.

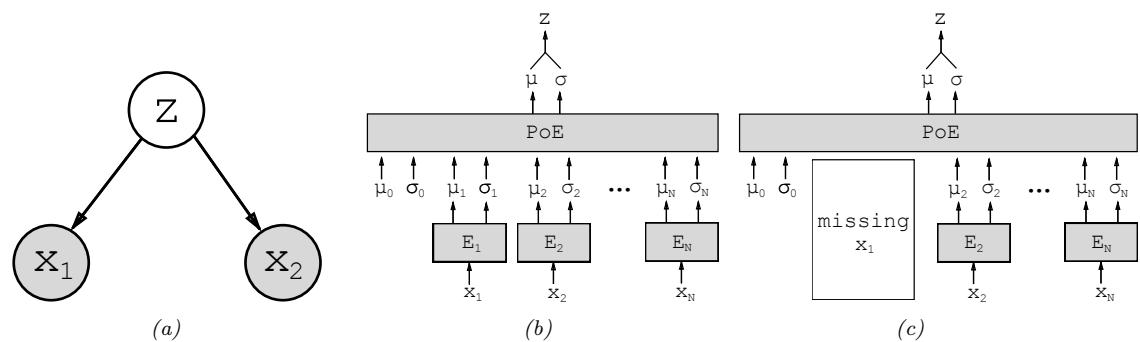


Figure 22.8: Illustration of multi-modal VAE. (a) The generative model with $N = 2$ modalities. (b) The product of experts (PoE) inference network is derived from N individual Gaussian experts E_i . μ_0 and σ_0 are parameters of the prior. (c) If a modality is missing, we omit its contribution to the posterior. From Figure 1 of [WG18]. Used with kind permission of Mike Wu.

M modalities. We assume they are conditionally independent given the latent code, and hence the generative model has the form

$$p_{\theta}(\mathbf{x}_1, \dots, \mathbf{x}_M, \mathbf{z}) = p(\mathbf{z}) \prod_{m=1}^M p_{\theta}(\mathbf{x}_m | \mathbf{z}) \quad (22.86)$$

where we treat $p(\mathbf{z})$ as a fixed prior. See Figure 22.8(a) for an illustration.

The standard ELBO is given by

$$\mathcal{L}_{\theta, \phi}(\mathbf{X}) = \mathbb{E}_{q_{\phi}(\mathbf{z} | \mathbf{X})} \left[\sum_m \log p_{\theta}(\mathbf{x}_m | \mathbf{z}) \right] - D_{\text{KL}}(q_{\phi}(\mathbf{z} | \mathbf{X}) \| p(\mathbf{z})) \quad (22.87)$$

where $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_M)$ is the observed data. However, the different likelihood terms $p(\mathbf{x}_m|\mathbf{z})$ may have different dynamic ranges (e.g., Gaussian pdf for pixels, and categorical pmf for text), so we introduce weight terms $\lambda_m \geq 0$ for each likelihood. In addition, let $\beta \geq 0$ control the amount of KL regularization. This gives us a weighted version of the ELBO, as follows:

$$\mathcal{L}_{\theta, \phi}(\mathbf{X}) = \mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{X})} \left[\sum_m \lambda_m \log p_{\theta}(\mathbf{x}_m|\mathbf{z}) \right] - \beta D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{X}) \| p(\mathbf{z})) \quad (22.88)$$

Often we don't have a lot of paired (aligned) data from all M modalities. For example, we may have a lot of images (modality 1), and a lot of text (modality 2), but very few (image, text) pairs. So it is useful to generalize the loss so it fits the marginal distributions of subsets of the features. Let $O_m = 1$ if modality m is observed (i.e., \mathbf{x}_m is known), and let $O_m = 0$ if it is missing or unobserved. Let $\mathbf{X} = \{\mathbf{x} : O_m = 1\}$ be the visible features. We now use the following objective:

$$\mathcal{L}_{\theta, \phi}(\mathbf{X}) = \mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{X})} \left[\sum_{m:O_m=1} \lambda_m \log p_{\theta}(\mathbf{x}_m|\mathbf{z}) \right] - \beta D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{X}) \| p(\mathbf{z})) \quad (22.89)$$

The key problem is how to compute the posterior $q_{\phi}(\mathbf{z}|\mathbf{X})$ given different subsets of features. In general this can be hard, since the inference network is a discriminative model that assumes all inputs are available. For example, if it is trained on (image, text) pairs, $q_{\phi}(\mathbf{z}|\mathbf{x}_1, \mathbf{x}_2)$, how can we compute the posterior just given an image, $q_{\phi}(\mathbf{z}|\mathbf{x}_1)$, or just given text, $q_{\phi}(\mathbf{z}|\mathbf{x}_2)$? (This issue arises in general with VAE when we have missing inputs; we discuss the general case in Section 22.3.5.)

Fortunately, based on our conditional independence assumption between the modalities, we can compute the optimal form for $q_{\phi}(\mathbf{z}|\mathbf{X})$ given set of inputs by computing the exact posterior under the model, which is given by

$$p(\mathbf{z}|\mathbf{X}) = \frac{p(\mathbf{z})p(\mathbf{x}_1, \dots, \mathbf{x}_M|\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} = \frac{p(\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} \prod_{m=1}^M p(\mathbf{x}_m|\mathbf{z}) \quad (22.90)$$

$$= \frac{p(\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} \prod_{m=1}^M \frac{p(\mathbf{z}|\mathbf{x}_m)p(\mathbf{x}_m)}{p(\mathbf{z})} \quad (22.91)$$

$$\propto p(\mathbf{z}) \prod_{m=1}^M \frac{p(\mathbf{z}|\mathbf{x}_m)}{p(\mathbf{z})} \approx p(\mathbf{z}) \prod_{m=1}^M \tilde{q}(\mathbf{z}|\mathbf{x}_m) \quad (22.92)$$

This can be viewed as a product of experts (Section 25.1.1), where each $\tilde{q}(\mathbf{z}|\mathbf{x}_m)$ is an “expert” for the m ’th modality, and $p(\mathbf{z})$ is the prior. We can compute the above posterior for any subset of modalities for which we have data by modifying the product over m . If we use Gaussian distributions for the prior $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0^{-1})$ and marginal posterior ratio $\tilde{q}(\mathbf{z}|\mathbf{x}_m) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1})$, then we can compute the product of Gaussians using the result from Equation (2.94):

$$\prod_{m=0}^M \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1}) \propto \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \boldsymbol{\Sigma} = (\sum_m \boldsymbol{\Lambda}_m)^{-1}, \quad \boldsymbol{\mu} = \boldsymbol{\Sigma}(\sum_m \boldsymbol{\Lambda}_m \boldsymbol{\mu}_m) \quad (22.93)$$

Thus the overall posterior precision is the sum of individual expert posterior precisions, and the overall posterior mean is the precision weighted average of the individual expert posterior means.

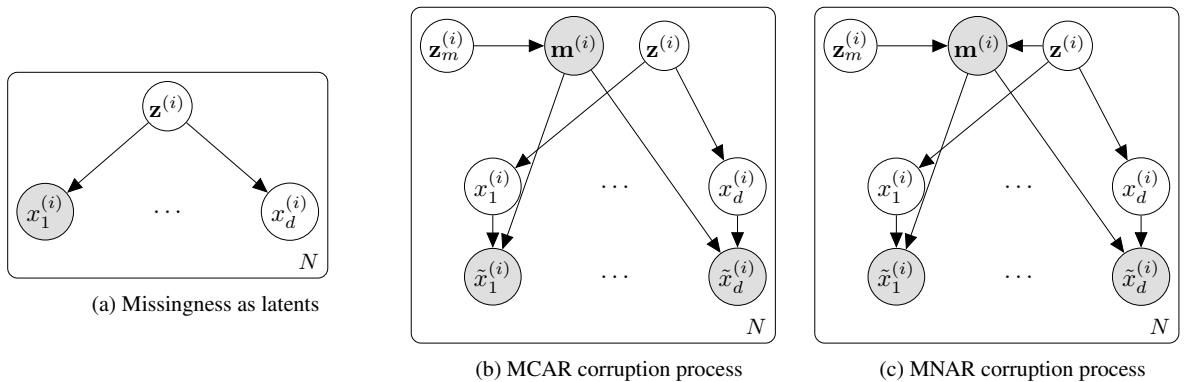


Figure 22.9: Illustration of different VAE variants for handling missing data. From Figure 1 of [CNW20]. Used with kind permission of Mark Collier.

See Figure 22.8(b) for an illustration. For a linear Gaussian (factor analysis) model, we can ensure $q(\mathbf{z}|\mathbf{x}_m) = p(\mathbf{z}|\mathbf{x}_m)$, in which case the above solution is the exact posterior [WN18], but in general it will be an approximation.

We need to train the individual expert recognition models $q(\mathbf{z}|\mathbf{x}_m)$ as well as the joint model $q(\mathbf{z}|\mathbf{X})$, so the model knows what to do with fully observed as well as partially observed inputs at test time. In [Ved+18], they propose a somewhat complex “triple ELBO” objective. In [WG18], they propose the simpler approach of optimizing the ELBO for the fully observed feature vector, all the marginals, and a set of J randomly chosen joint modalities:

$$\mathcal{L}_{\theta, \phi}(\mathbf{X}) = \mathcal{L}_{\theta, \phi}(\mathbf{x}_1, \dots, \mathbf{x}_M) + \sum_{m=1}^M \mathcal{L}_{\theta, \phi}(\mathbf{x}_m) + \sum_{j=1}^J \mathcal{L}_{\theta, \phi}(\mathbf{X}_j) \quad (22.94)$$

This generalizes nicely to the semi-supervised setting, in which we only have a few aligned (“labeled”) examples from the joint, but have many unaligned (“unlabeled”) examples from the individual marginals. See Figure 22.8(c) for an illustration.

Note that the above scheme can only handle the case of a fixed number of missingness patterns; we generalize to allow for arbitrary missingness in Section 22.3.5.

22.3.5 VAEs with missing data

Sometimes we may have **missing data**, in which parts of the data vector $\mathbf{x} \in \mathbb{R}^D$ may be unknown. In Section 22.3.4 we saw a special case of this when we discussed multimodal VAEs. In this section we allow for arbitrary patterns of missingness.

To model the missing data, let $\mathbf{m} \in \{0, 1\}^D$ be a binary vector where $m_j = 1$ is x_j is missing, and $m_j = 0$ otherwise. Let $\mathbf{X} = \{\mathbf{x}^{(n)}\}$ and $\mathbf{M} = \{\mathbf{m}^{(n)}\}$ be $N \times D$ matrices. Furthermore, let \mathbf{X}_o be the observed parts of \mathbf{X} and \mathbf{X}_h be the hidden parts. If we assume $p(\mathbf{M}|\mathbf{X}_o, \mathbf{X}_h) = p(\mathbf{M})$, we say the data is **missing completely at random** or **MCAR**, since the missingness does not depend on the hidden or observed features. If we assume $p(\mathbf{M}|\mathbf{X}_o, \mathbf{X}_h) = p(\mathbf{M}|\mathbf{X}_o)$, we say the data is **missing at**

1 random or MAR, since the missingness does not depend on the hidden features, but may depend
2 on the visible features. If neither of these assumptions hold, we say the data is **not missing at**
3 **random or NMAR.**

4 In the MCAR and MAR cases, we can ignore the missingness mechanism, since it tells us nothing
5 about the hidden features. However, in the NMAR case, we need to model the **missing data**
6 **mechanism**, since the lack of information may be informative. For example, the fact that someone
7 did not fill out an answer to a sensitive question on a survey (e.g., “Do you have COVID?”) could be
8 informative about the underlying value. See e.g., [LR87; Mar08] for more information on missing
9 data models.

10 In the context of VAEs, we can model the MCAR scenario by treating the missing values as latent
11 variables. This is illustrated in Figure 22.9(a). Since missing leaf nodes in a directed graphical model
12 do not affect their parents, we can simply ignore them when computing the posterior $p(\mathbf{z}^{(i)}|\mathbf{x}_o^{(i)})$,
13 where $\mathbf{x}_o^{(i)}$ are the observed parts of example i . However, when using an amortized inference network,
14 it can be difficult to handle missing inputs, since the model is usually trained to compute $p(\mathbf{z}^{(i)}|\mathbf{x}_{1:d}^{(i)})$.
15 One solution to this is to use the product of experts approach discussed in the context of multi-modal
16 VAEs in Section 22.3.4. However, this is designed for the case where whole blocks (corresponding to
17 different modalities) are missing, and will not work well if there are arbitrary missing patterns (e.g.,
18 pixels that get dropped out due to occlusion or scratches on the lens). In addition, this method will
19 not work for the MNAR case.

20 An alternative approach, proposed in [CNW20], is to explicitly include the missingness indicators
21 into the model, as shown in Figure 22.9(b). We assume the model always generates each \mathbf{x}_j for
22 $j = 1 : d$, but we only get to see the “corrupted” versions $\tilde{\mathbf{x}}_j$. If $m_j = 0$ then $\tilde{\mathbf{x}}_j = \mathbf{x}_j$, but if $m_j = 1$,
23 then $\tilde{\mathbf{x}}_j$ is a special value, such as 0, unrelated to \mathbf{x}_j . We can model any correlation between the
24 missingness elements (components of \mathbf{m}) by using another latent variable \mathbf{z}_m . This model can easily
25 be extended to the MNAR case by letting \mathbf{m} depend on the latent factors for the observed data, \mathbf{z} ,
26 as well as the usual missingness latent factors \mathbf{z}_m , as shown in Figure 22.9(c).

27 We modify the VAE to be conditional on the missingness pattern, so the VAE decoder has the
28 form $p(\mathbf{x}_o|\mathbf{z}, \mathbf{m})$, and the encoder has the form $q(\mathbf{z}|\mathbf{x}_o, \mathbf{m})$. However, we assume the prior is $p(\mathbf{z})$
29 as usual, independent of \mathbf{m} . We can compute a lower bound on the log marginal likelihood of the
30 observed data, given the missingness, as follows:

31

$$\log p(\mathbf{x}_o|\mathbf{m}) = \log \int \int p(\mathbf{x}_o, \mathbf{x}_m|\mathbf{z}, \mathbf{m}) p(\mathbf{z}) d\mathbf{x}_m d\mathbf{z} \quad (22.95)$$

$$= \log \int p(\mathbf{x}_o|\mathbf{z}, \mathbf{m}) p(\mathbf{z}) d\mathbf{z} \quad (22.96)$$

$$= \log \int p(\mathbf{x}_o|\mathbf{z}, \mathbf{m}) p(\mathbf{z}) \frac{q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})}{q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})} d\mathbf{z} \quad (22.97)$$

$$= \log \mathbb{E}_{q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})} \left[p(\mathbf{x}_o|\mathbf{z}, \mathbf{m}) \frac{p(\mathbf{z})}{q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})} \right] \quad (22.98)$$

$$\geq \mathbb{E}_{q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})} [\log p(\mathbf{x}_o|\mathbf{z}, \mathbf{m}) - D_{\text{KL}}(q(\mathbf{z}|\tilde{\mathbf{x}}, \mathbf{m})||p(\mathbf{z}))] \quad (22.99)$$

43

44 We can fit this model in the usual way. See Figure 22.10 for an example.

45

46

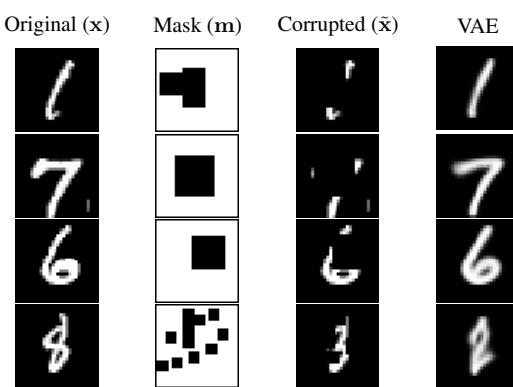


Figure 22.10: Imputing missing pixels given a masked out image using a VAE using a MCAR assumption. From Figure 2 of [CNW20]. Used with kind permission of Mark Collier.

22.3.6 Semi-supervised VAEs

In this section, we discuss how to extend VAEs to the **semi-supervised learning** setting in which we have both labeled data, $\mathcal{D}_L = \{(\mathbf{x}_n, y_n)\}$, and unlabeled data, $\mathcal{D}_U = \{(\mathbf{x}_n)\}$. We focus on the **M2** model, proposed in [Kin+14].

The generative model has the following form:

$$p_{\theta}(\mathbf{x}, y) = p_{\theta}(y)p_{\theta}(\mathbf{x}|y) = p_{\theta}(y) \int p_{\theta}(\mathbf{x}|y, \mathbf{z})p_{\theta}(\mathbf{z})d\mathbf{z} \quad (22.100)$$

where \mathbf{z} is a latent variable, $p_{\theta}(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$ is the latent prior, $p_{\theta}(y) = \text{Cat}(y|\boldsymbol{\pi})$ the label prior, and $p_{\theta}(\mathbf{x}|y, \mathbf{z}) = p(\mathbf{x}|f_{\theta}(y, \mathbf{z}))$ is the likelihood, such as a Gaussian, with parameters computed by f (a deep neural network). The main innovation of this approach is to assume that data is generated according to both a latent class variable y as well as the continuous latent variable \mathbf{z} . The class variable y is observed for labeled data and unobserved for unlabeled data.

To compute the likelihood for the *labeled data*, $p_{\theta}(\mathbf{x}, y)$, we need to marginalize over \mathbf{z} , which we can do by using an inference network of the form

$$q_{\phi}(\mathbf{z}|y, \mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\phi}(y, \mathbf{x}), \text{diag}(\sigma_{\phi}^2(\mathbf{x}))) \quad (22.101)$$

We then use the following variational lower bound

$$\log p_{\theta}(\mathbf{x}, y) \geq \mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x}, y)} [\log p_{\theta}(\mathbf{x}|y, \mathbf{z}) + \log p_{\theta}(y) + \log p_{\theta}(\mathbf{z}) - \log q_{\phi}(\mathbf{z}|\mathbf{x}, y)] = -\mathcal{L}(\mathbf{x}, y) \quad (22.102)$$

as is standard for VAEs (see Section 22.2). The only difference is that we observe two kinds of data: \mathbf{x} and y .

To compute the likelihood for the *unlabeled data*, $p_{\theta}(\mathbf{x})$, we need to marginalize over \mathbf{z} and y , which we can do by using an inference network of the form

$$q_{\phi}(\mathbf{z}, y|\mathbf{x}) = q_{\phi}(\mathbf{z}|\mathbf{x})q_{\phi}(y|\mathbf{x}) \quad (22.103)$$

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\phi}(\mathbf{x}), \text{diag}(\sigma_{\phi}^2(\mathbf{x}))) \quad (22.104)$$

$$q_{\phi}(y|\mathbf{x}) = \text{Cat}(y|\boldsymbol{\pi}_{\phi}(\mathbf{x})) \quad (22.105)$$

1 Note that $q_\phi(y|\mathbf{x})$ acts like a discriminative classifier, that imputes the missing labels. We then use
2 the following variational lower bound:
3

$$\log p_{\theta}(\mathbf{x}) \geq \mathbb{E}_{q_{\phi}(\mathbf{z}, y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|y, \mathbf{z}) + \log p_{\theta}(y) + \log p_{\theta}(\mathbf{z}) - \log q_{\phi}(\mathbf{z}, y|\mathbf{x})] \quad (22.106)$$

$$= - \sum_y q_{\phi}(y|\mathbf{x}) \mathcal{L}(\mathbf{x}, y) + \mathbb{H}(q_{\phi}(y|\mathbf{x})) = -\mathcal{U}(\mathbf{x}) \quad (22.107)$$

9 Note that the discriminative classifier $q_{\phi}(y|\mathbf{x})$ is only used to compute the log-likelihood of the
10 unlabeled data, which is undesirable. We can therefore add an extra classification loss on the
11 supervised data, to get the following overall objective function:

$$\mathcal{L}(\theta) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}_L} [\mathcal{L}(\mathbf{x}, y)] + \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_U} [\mathcal{U}(\mathbf{x})] + \alpha \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}_L} [-\log q_{\phi}(y|\mathbf{x})] \quad (22.108)$$

14 where α is a hyperparameter that controls the relative weight of generative and discriminative
15 learning.
16

17 22.3.7 VAEs with sequential encoders/decoders

19 In this section, we discuss VAEs for sequential data, such as text and biosequences, in which the
20 data \mathbf{x} is a variable-length sequence, but we have a fixed-sized latent variable $\mathbf{z} \in \mathbb{R}^K$. (We consider
21 the more general case in which \mathbf{z} is a variable-length sequence of latents — known as **sequential**
22 **VAE** or **dynamic VAE** — in Section 31.5.) All we have to do is modify the decoder $p(\mathbf{x}|\mathbf{z})$ and
23 encoder $q(\mathbf{z}|\mathbf{x})$ to work with sequences.
24

25 22.3.7.1 Models

27 If we use an RNN for the encoder and decoder of a VAE, we get a model which is called a **VAE-RNN**,
28 as proposed in [Bow+16a]. In more detail, the generative model is $p(\mathbf{z}, \mathbf{x}_{1:T}) = p(\mathbf{z}) \text{RNN}(\mathbf{x}_{1:T}|\mathbf{z})$,
29 where \mathbf{z} can be injected as the initial state of the RNN, or as an input to every time step. The
30 inference model is $q(\mathbf{z}|\mathbf{x}_{1:T}) = \mathcal{N}(\mathbf{z}|\mu(\mathbf{h}), \Sigma(\mathbf{h}))$, where $\mathbf{h} = [\mathbf{h}_T^\rightarrow, \mathbf{h}_1^\leftarrow]$ is the output of a bidirectional
31 RNN applied to $\mathbf{x}_{1:T}$. See Figure 22.11 for an illustration.

32 More recently, people have tried to combine transformers with VAEs. For example, in the **Optimus**
33 model of [Li+20a], they use a BERT model for the encoder. In more detail, the encoder $q(\mathbf{z}|\mathbf{x})$ is
34 derived from the embedding vector associated with a dummy token corresponding to the “class label”
35 which is appended to the input sequence \mathbf{x} . The decoder is a standard autoregressive model (similar
36 to GPT), with one additional input, namely the latent vector \mathbf{z} . They consider two ways of injecting
37 the latent vector. The simplest approach is to add \mathbf{z} to the embedding layer of every token in the
38 decoding step, by defining $\mathbf{h}'_i = \mathbf{h}_i + \mathbf{W}\mathbf{z}$, where $\mathbf{h}_i \in \mathbb{R}^H$ is the original embedding for the i 'th
39 token, and $\mathbf{W} \in \mathbb{R}^{H \times K}$ is a decoding matrix, where K is the size of the latent vector. However, they
40 get better results in their experiments by letting all the layers of the decoder attend to the latent
41 code \mathbf{z} . An easy way to do this is to define the memory vector $\mathbf{h}_m = \mathbf{W}\mathbf{z}$, where $\mathbf{W} \in \mathbb{R}^{LH \times K}$,
42 where L is the number of layers in the decoder, and then to append $\mathbf{h}_m \in \mathbb{R}^{L \times H}$ to all the other
43 embeddings at each layer.

44 An alternative approach, known as **transformer VAE**, was proposed in [Gre20]. This model uses
45 a **funnel transformer** [Dai+20b] as the encoder, and the **T5** [Raf+19] conditional transformer for
46 the decoder. In addition, it uses an MMD VAE (Section 22.3.3.1) to avoid posterior collapse.
47

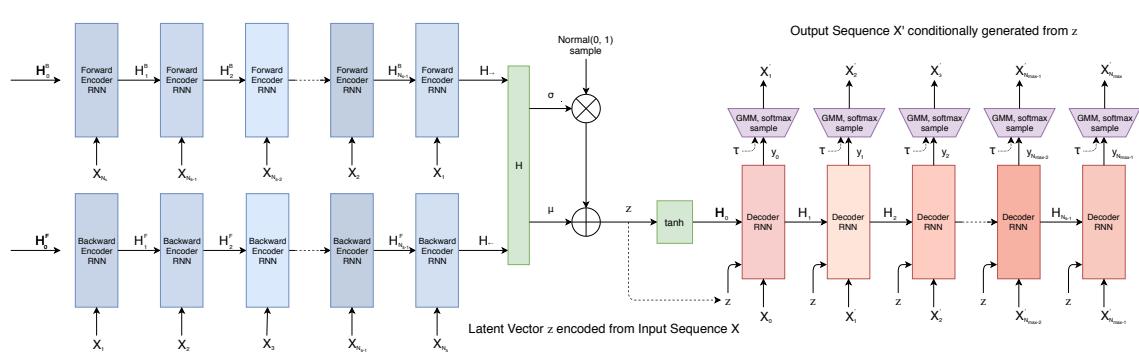


Figure 22.11: Illustration of a VAE with a bidirectional RNN encoder and a unidirectional RNN decoder. The output generator can use a GMM and/or softmax distribution. From Figure 2 of [HE18]. Used with kind permission of David Ha.

he was silent for a long moment .
 he was silent for a moment .
 it was quiet for a moment .
 it was dark and cold .
 there was a pause .
 it was my turn .

i went to the store to buy some groceries .
 i store to buy some groceries .
 i were to buy any groceries .
 horses are to buy any groceries .
 horses are to buy any animal .
 horses the favorite any animal .
 horses the favorite favorite animal .
 horses are my favorite animal .

(a)

(b)

Figure 22.12: (a) Samples from the latent space of a VAE text model, as we interpolate between two sentences (on first and last line). Note that the intermediate sentences are grammatical, and semantically related to their neighbors. From Table 8 of [Bow+16b]. (b) Same as (a), but now using a deterministic autoencoder (with the same RNN encoder and decoder). From Table 1 of [Bow+16b]. Used with kind permission of Sam Bowman.

22.3.7.2 Applications

In this section, we discuss some applications of VAEs to sequence data.

Text

In [Bow+16b], they apply the VAE-RNN model to natural language sentences. (See also [MB16; SSB17] for related work.) Although this does not improve performance in terms of the standard perplexity measures (predicting the next word given the previous words), it does provide a way to infer a semantic representation of the sentence. This can then be used for latent space interpolation, as discussed in Section 21.3.5. The results of doing this with the VAE-RNN are illustrated in Figure 22.12a. (Similar results are shown in [Li+20a], using a VAE-transformer.) By contrast, if we use a standard deterministic autoencoder, with the same RNN encoder and decoder networks, we learn a much less meaningful space, as illustrated in Figure 22.12b. The reason is that the deterministic autoencoder has “holes” in its latent space, which get decoded to nonsensical outputs.

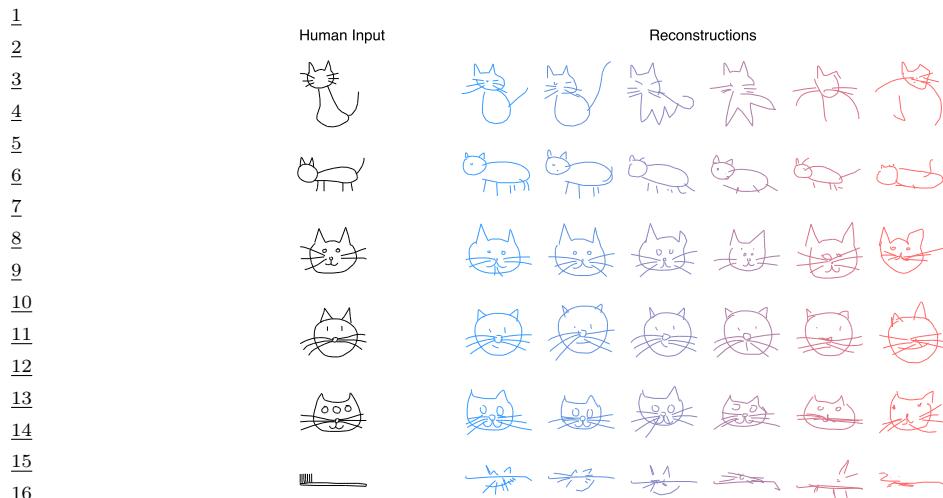


Figure 22.13: Conditional generation of cats from sketch-RNN model. We increase the temperature parameter from left to right. From Figure 5 of [HE18]. Used with kind permission of David Ha.

However, because RNNs (and transformers) are powerful decoders, we need to address the problem of posterior collapse, which we discuss in Section 22.4. One common way to avoid this problem is to use KL annealing, but a more effective method is to use the InfoVAE method of Section 22.3.3, which includes adversarial autoencoders (used in [She+20] with an RNN decoder) and MMD autoencoders (used in [Gre20] with a transformer decoder).

Sketches

In [HE18], they apply the VAE-RNN model to generate sketches (line drawings) of various animals and hand-written characters. They call their model **sketch-rnn**. The training data records the sequence of (x, y) pen positions, as well as whether the pen was touching the paper or not. The emission model used a GMM for the real-valued location offsets, and a categorical softmax distribution for the discrete state.

Figure 22.13 shows some samples from various class-conditional models. We vary the temperature parameter τ of the emission model to control the stochasticity of the generator. (More precisely, we multiply the GMM variances by τ , and divide the discrete probabilities by τ before renormalizing.) When the temperature is low, the model tries to reconstruct the input as closely as possible. However, when the input is untypical of the training set (e.g., a cat with three eyes, or a toothbrush), the reconstruction is “regularized” towards a canonical cat with two eyes, while still keeping some features of the input.

47

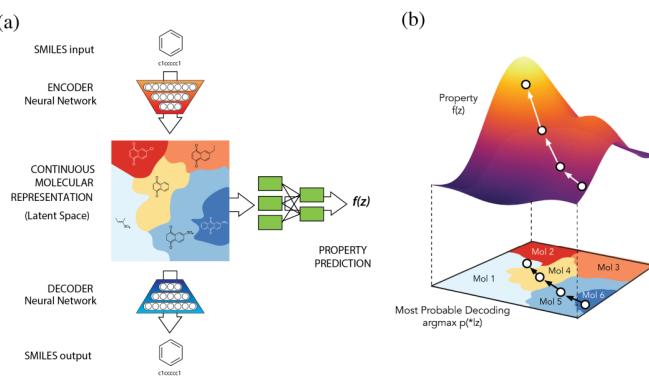


Figure 22.14: Application of VAE-RNN to molecule design. (a) The VAE-RNN model is trained on a sequence representation of molecules known as SMILES. We can fit an MLP to map from the latent space to properties of the molecule, such as its “fitness” $f(\mathbf{z})$. (b) We can perform gradient ascent in $f(\mathbf{z})$ space, and then decode the result to a new molecule with high fitness. From Figure 1 of [GB+18]. Used with kind permission of Rafael Gomez-Bombarelli.

Molecular design

In [GB+18], they use VAE-RNNs to model molecular graph structure, represented as a string using the SMILES representation.⁶ It is also possible to learn a mapping from the latent space to some scalar quantity of interest, such as the solubility or drug efficacy of a molecule. We can then perform gradient-based optimization in the continuous latent space to try to generate new graphs which maximize this quantity. See Figure 22.14 for a sketch of this approach.

The main problem is to ensure that points in latent space decode to valid strings/ molecules. There are various solutions to this, including using a **grammar VAE**, where the RNN decoder is replaced by a stochastic context free grammar. See [KPHL17] for details.

22.4 Avoiding posterior collapse

If the decoder $p_{\theta}(\mathbf{x}|\mathbf{z})$ is sufficiently powerful (e.g., a pixel CNN, or an RNN for text), then the VAE does not need to use the latent code \mathbf{z} for anything. This is called **posterior collapse** or **variational overpruning** (see e.g., [Che+17b; Ale+18; Hus17a; Phu+18; TT17; Yeu+17; Luc+19; DWW19; WBC21]). To see why this happens, consider Equation (22.60). If there exists a parameter setting for the generator θ^* such that $p_{\theta^*}(\mathbf{x}|\mathbf{z}) = p_D(\mathbf{x})$ for every \mathbf{z} , then we can make $D_{\text{KL}}(p_D(\mathbf{x})||p_{\theta}(\mathbf{x})) = 0$. Since the generator is independent of the latent code, we have $p_{\theta}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{z})$. The prior $p_{\theta}(\mathbf{z})$ is usually a simple distribution, such as a Gaussian, so we can find a setting of the inference parameters so that $q_{\phi^*}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{z})$, which ensures $D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x})||p_{\theta}(\mathbf{z}|\mathbf{x})) = 0$. Thus we have successfully minimized the ELBO, but we have not learned any useful latent representation of the data, which is

⁶ See https://en.wikipedia.org/wiki/Simplified_molecular_input_line-entry_system.

1 one of the goals of latent variable modeling.⁷

2 We discuss some solutions to posterior collapse below.

3

4 5 22.4.1 KL annealing

6 A common approach to solving this problem, proposed in [Bow+16a], is to use **KL annealing**, in
7 which the KL penalty term in the ELBO is scaled by β , which is increased from 0.0 (corresponding
8 to an autoencoder) to 1.0 (which corresponds to standard MLE training). (Note that, by contrast,
9 the β -VAE model in Section 22.3.2 uses $\beta > 1$.)

10 KL annealing can work well, but requires tuning the schedule for β . A standard practice [Fu+19]
11 is to use **cyclical annealing**, which repeats the process of increasing β multiple times. This ensures
12 the progressive learning of more meaningful latent codes, by leveraging good representations learned
13 in a previous cycle as a way to warmstart the optimization.

14

15

16 22.4.2 Lower bounding the rate

17 An alternative approach is to stick with the original unmodified ELBO objective, but to prevent
18 the rate (i.e., the $D_{\text{KL}}(q\|p)$ term) from collapsing to 0, by limiting the flexibility of q . For example,
19 [XD18; Dav+18] use a von Mises-Fisher (Section 2.4.2) prior and posterior, instead of a Gaussian,
20 and they constrain the posterior to have a fixed concentration, $q(\mathbf{z}|\mathbf{x}) = \text{vMF}(\mathbf{z}|\boldsymbol{\mu}(\mathbf{x}), \kappa)$. Here
21 the parameter κ controls the rate of the code. The δ -VAE method [Oor+19] uses a Gaussian
22 autoregressive prior and a diagonal Gaussian posterior. We can ensure the rate is at least δ by
23 adjusting the regression parameter of the AR prior.

24

25

26 22.4.3 Free bits

27

28 In this section, we discuss the method of **free bits** [Kin+16], which is another way of lower bounding
29 the rate. To explain this, consider a fully factorized posterior in which the KL penalty has the form

30

$$\mathcal{L}_R = \sum_i D_{\text{KL}}(q_\phi(z_i|\mathbf{x})\|p_\theta(z_i)) \quad (22.109)$$

31

32

33 where z_i is the i 'th dimension of \mathbf{z} . We can replace this with a hinge loss, that will give up driving
34 down the KL for dimensions that are already beneath a target compression rate λ :

35

$$\mathcal{L}'_R = \sum_i \max(\lambda, D_{\text{KL}}(q_\phi(z_i|\mathbf{x})\|p_\theta(z_i))) \quad (22.110)$$

36

37 Thus the bits where the KL is sufficiently small “are free”, since the model does not have to “pay” to
38 encode them according to the prior.

39

40

41 7. Note that [Luc+19; DWW20] show that posterior collapse can also happen in linear VAE models, where the ELBO
42 corresponds to the exact marginal likelihood, so the problem is not only due to powerful (nonlinear) decoders, but is
43 also related to spurious local maxima in the objective.

44

45

46

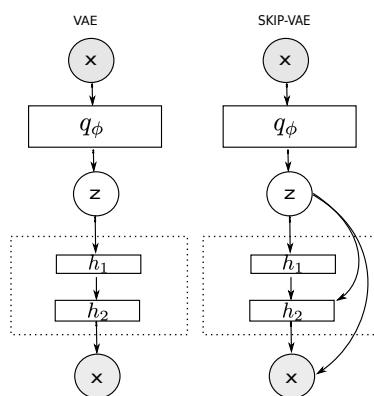


Figure 22.15: (a) VAE. (b) Skip-VAE. From Figure 1 of [Die+19a]. Used with kind permission of Adji Dieng.

22.4.4 Adding skip connections

One reason for latent variable collapse is that the latent variables z are not sufficiently “connected to” the observed data x . One simple solution is to modify the architecture of the generative model by adding **skip connections**, similar to a residual network (Section 16.2.4), as shown in Figure 22.15. This is called a **skip-VAE** [Die+19a].

22.4.5 Improved variational inference

The posterior collapse problem is caused in part by the poor approximation to the posterior. In [He+19], they proposed to keep the the model and VAE objective unchanged, but to more aggressively update the inference network before each step of generative model fitting. This enables the inference network to capture the current true posterior more faithfully, which will encourage the generator to use the latent codes when it is useful to do so.

However, this only addresses the part of posterior collapse that is due to the amortization gap [CLD18], rather than the more fundamental problem of variational pruning, in which the KL term penalizes the model if its posterior deviates too far from the prior, which is often too simple to match the aggregated posterior.

Another way to ameliorate variational pruning is to use lower bounds than are tighter than the vanilla ELBO (Section 10.5), or more accurate posterior approximations (Section 10.4), or more accurate (hierarchical) generative models (Section 22.5).

22.4.6 Alternative objectives

An alternative to the above methods is to replace the ELBO objective with other objectives, such as the InfoVAE objective discussed in Section 22.3.3, which includes adversarial autoencoders and MMD autoencoders as special cases. The InfoVAE objective includes a term to explicit enforce non-zero mutual information between x and z , which effectively solves the problem of posterior collapse.

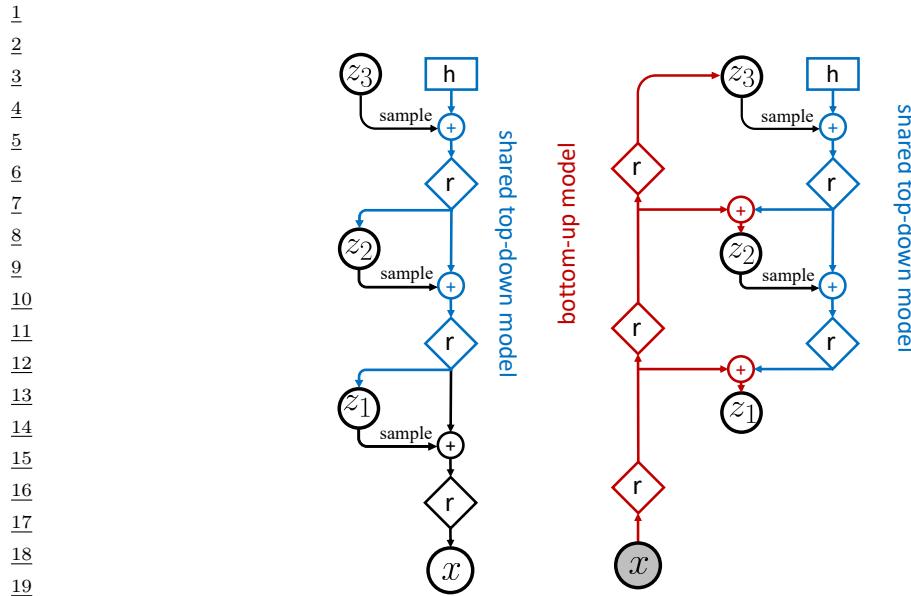


Figure 22.16: Hierarchical VAEs with 3 stochastic layers. Left: Generative model. Right: Inference network. Diamond is a residual network, \oplus is feature combination (e.g., concatenation), and h is a trainable parameter. We first do bottom-up inference, by propagating x up to z_3 to compute $z_3^s \sim q_\phi(z_3|x)$, and then we perform top-down inference by computing $z_2^s \sim q_\phi(z_2|x, z_3^s)$ and then $z_1^s \sim q_\phi(z_1|x, z_{2:3}^s)$. From Figure 2 of [VK20]. Used with kind permission of Arash Vahdat.

22.4.7 Enforcing identifiability

In [WBC21], they show that posterior collapse happens whenever the latent variables are not uniquely identifiable given the data. This can happen even in simple models, like a GMM, and even using exact inference methods. To see why, recall that non-identifiability of the latent variable means for each z , and all θ , there is some $z' \neq z$ such that $p(x|z, \theta) = p(x|z', \theta) = p(x|\theta)$, so the likelihood is invariant to the latent code.⁸ Consequently the posterior collapses to the prior:

$$p(z|x, \theta) \propto p(z)p(x|z, \theta) = p(z)p(x|\theta) \propto p(z) \quad (22.111)$$

To enforce identifiability of z , the authors propose to use monotone transport maps, and input-convex neural networks. Unfortunately this makes inference slower. See the paper for details.

22.5 VAEs with hierarchical structure

We define a **hierarchical VAE**, with L stochastic layers, to be the following generative model:⁹

$$p_{\theta}(\mathbf{x}, \mathbf{z}_{1:L}) = p_{\theta}(\mathbf{z}_L) \left[\prod_{l=L-1}^1 p_{\theta}(\mathbf{z}_l | \mathbf{z}_{l+1}) \right] p_{\theta}(\mathbf{x} | \mathbf{z}_1) \quad (22.112)$$

We can improve on the above model by making it non-Markovian, i.e., letting each \mathbf{z}_l depend on all the higher level stochastic variables, $\mathbf{z}_{l+1:L}$, not just the preceding level, i.e.,

$$p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{z}_L) \left[\prod_{l=L-1}^1 p_{\theta}(\mathbf{z}_l | \mathbf{z}_{l+1:L}) \right] p_{\theta}(\mathbf{x} | \mathbf{z}_{1:L}) \quad (22.113)$$

Note that the likelihood is now $p_{\theta}(\mathbf{x} | \mathbf{z}_{1:L})$ instead of just $p_{\theta}(\mathbf{x} | \mathbf{z}_1)$. This is analogous to adding skip connections from all preceding variables to all their children. It is easy to implement this by using a deterministic “backbone” of residual connections, that accumulates all stochastic decisions, and propagates them down the chain, as illustrated in Figure 22.16(left). We discuss how to perform inference and learning in such models below.

22.5.1 Bottom-up vs top-down inference

To perform inference in a hierarchical VAE, we could use a **bottom-up inference model** of the form

$$q_{\phi}(\mathbf{z} | \mathbf{x}) = q_{\phi}(\mathbf{z}_1 | \mathbf{x}) \prod_{l=2}^L q_{\phi}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{1:l-1}) \quad (22.114)$$

However, a better approach is to use a **top-down inference model** of the form

$$q_{\phi}(\mathbf{z} | \mathbf{x}) = q_{\phi}(\mathbf{z}_L | \mathbf{x}) \prod_{l=L-1}^1 q_{\phi}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L}) \quad (22.115)$$

Inference for \mathbf{z}_l combines bottom-up information from \mathbf{x} with top-down information from higher layers, $\mathbf{z}_{>l} = \mathbf{z}_{l+1:L}$. See Figure 22.16(right) for an illustration.¹⁰

With the above model, the ELBO can be written as follows:

$$\mathcal{L}_{\theta, \phi}(\mathbf{x}) = \mathbb{E}_{q_{\phi}(\mathbf{z} | \mathbf{x})} [\log p_{\theta}(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q_{\phi}(\mathbf{z}_L | \mathbf{x}) \| p_{\theta}(\mathbf{z}_L)) \quad (22.116)$$

$$- \sum_{l=L-1}^1 \mathbb{E}_{q_{\phi}(\mathbf{z}_{>l} | \mathbf{x})} [D_{\text{KL}}(q_{\phi}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{>l}) \| p_{\theta}(\mathbf{z}_l | \mathbf{z}_{>l}))] \quad (22.117)$$

⁸. Note that identifiability of \mathbf{z} is a weaker requirement compared to the whole model being identifiable, including the parameters.

⁹. There is a split in the literature about whether to label the top level as \mathbf{z}_L or \mathbf{z}_1 . We adopt the former convention, since we view lower numbered layers, such as \mathbf{z}_1 , as being “closer to the data”, and higher numbered layers, such as \mathbf{z}_L , as being “more abstract”.

¹⁰. Note that it is also possible to have a stochastic bottom-up encoder and a stochastic top-down encoder, as discussed in the **BIVA** paper [Maa+19]. (BIVA stands for “Bidirectional-Inference Variational Autoencoder.”)

1 where
2

3

$$\underline{4} \quad q_{\phi}(\mathbf{z}_{>l|\mathbf{x}}) = \prod_{i=l+1}^L q_{\phi}(\mathbf{z}_i|\mathbf{x}, \mathbf{z}_{>i}) \quad (22.118)$$

5

6 is the approximate posterior above layer l (i.e., the parents of \mathbf{z}_l).
7

8 The reason the top-down inference model is better is that it more closely approximates the true
9 posterior of a given layer, which is given by

10

$$\underline{11} \quad p_{\theta}(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L}) \propto p_{\theta}(\mathbf{z}_l|\mathbf{z}_{l+1:L}) p_{\theta}(\mathbf{x}|\mathbf{z}_l, \mathbf{z}_{l+1:L}) \quad (22.119)$$

12

13 Thus the posterior combines the top-down prior term $p_{\theta}(\mathbf{z}_l|\mathbf{z}_{l+1:L})$ with the bottom-up likelihood
14 term $p_{\theta}(\mathbf{x}|\mathbf{z}_l, \mathbf{z}_{l+1:L})$. We can approximate this posterior by defining

15

$$\underline{16} \quad q_{\phi}(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L}) \propto p_{\theta}(\mathbf{z}_l|\mathbf{z}_{l+1:L}) \tilde{q}_{\phi}(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L}) \quad (22.120)$$

17

18 where $\tilde{q}_{\phi}(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L})$ is a learned Gaussian approximation to the bottom-up likelihood. If both prior
19 and likelihood are Gaussian, we can compute this product in closed form, as proposed in the **ladder**
20 **network** paper [Sn+16; Søn+16].¹¹ A more flexible approach is to let $q_{\phi}(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L})$ be learned,
21 but to force it to share some of its parameters with the learned prior $p_{\theta}(\mathbf{z}_l|\mathbf{z}_{l+1:L})$, as proposed in
22 [Kin+16]. This reduces the number of parameters in the model, and ensures that the posterior and
23 prior remain somewhat close.

24 22.5.2 Example: Very deep VAE

25 There have been many papers exploring different kinds of HVAE models (see e.g., [Kin+16; Sn+16;
26 Chi21; VK20; Maa+19]), and we do not have space to discuss them all. Here we focus on the “**very**
27 **deep VAE**” or **VD-VAE** model of [Chi21], since it is simple but yields state of the art results (at
28 the time of writing).

29 The architecture is a simple convolutional VAE with bidirectional inference, as shown in Figure 22.17.
30 For each layer, the prior and posterior are diagonal Gaussians. The author found that nearest-neighbor
31 upsampling (in the decoder) worked much better than transposed convolution, and avoided posterior
32 collapse. This enabled training with the vanilla VAE objective, without needing any of the tricks
33 discussed in Section 22.5.4.

34 The low-resolution latents (at the top of the hierarchy) capture a lot of the global structure of
35 each image; the remaining high-resolution latents are just used to fill in details, that make the image
36 look more realistic, and improve the likelihood. This suggests the model could be useful for lossy
37 compression, since a lot of the low-level details can be drawn from the prior (i.e., “hallucinated”),
38 rather than having to be sent by the encoder. We illustrate this in Figure 22.18 using a small model
39 trained on CIFAR-10, which is a dataset of 60,000 32x32 color images from 10 classes. If we just use
40 the top level code (of size $1 \times 1 \times C$, where $C = 384$ channels), we cannot reliably encode the input
41 image. (For example, the jet fighter (airplane) gets reconstructed as a car.) However, we start to get
42 “plausible hallucinations” after using just the top 3 levels (of size $8 \times 8 \times C$). (Similar results are
43 shown in [VK20].)

44

⁴⁵ 11. The term “ladder network” arises from the horizontal “rungs” in Figure 22.16(right). Note that a similar idea was
46 independently proposed in [Sal16].

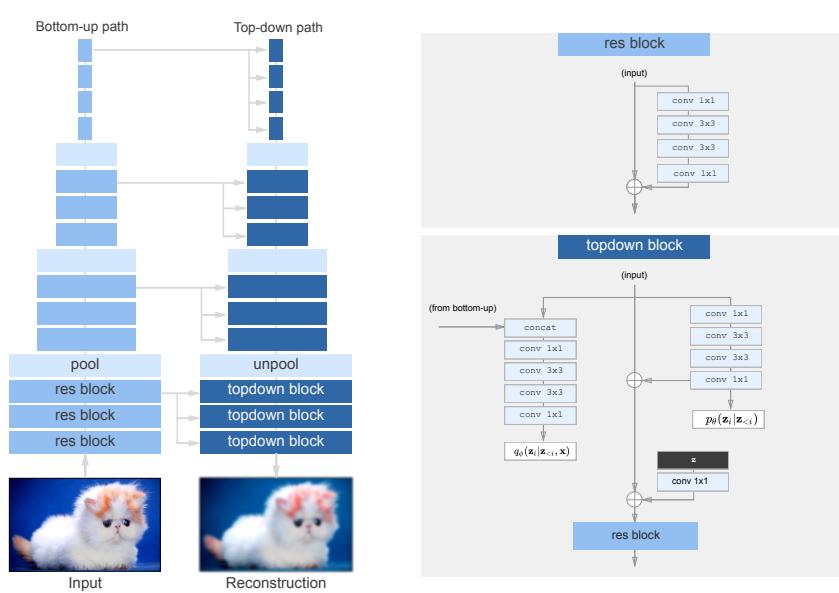


Figure 22.17: The top-down encoder used by the hierarchical VAE in [Chi21]. Each convolution is preceded by the GELU nonlinearity. The model uses average pooling and nearest-neighbor upsampling for the pool and unpool layers. The posterior q_ϕ and prior p_θ are diagonal Gaussians. From Figure 3 of [Chi21]. Used with kind permission of Ronan Child.

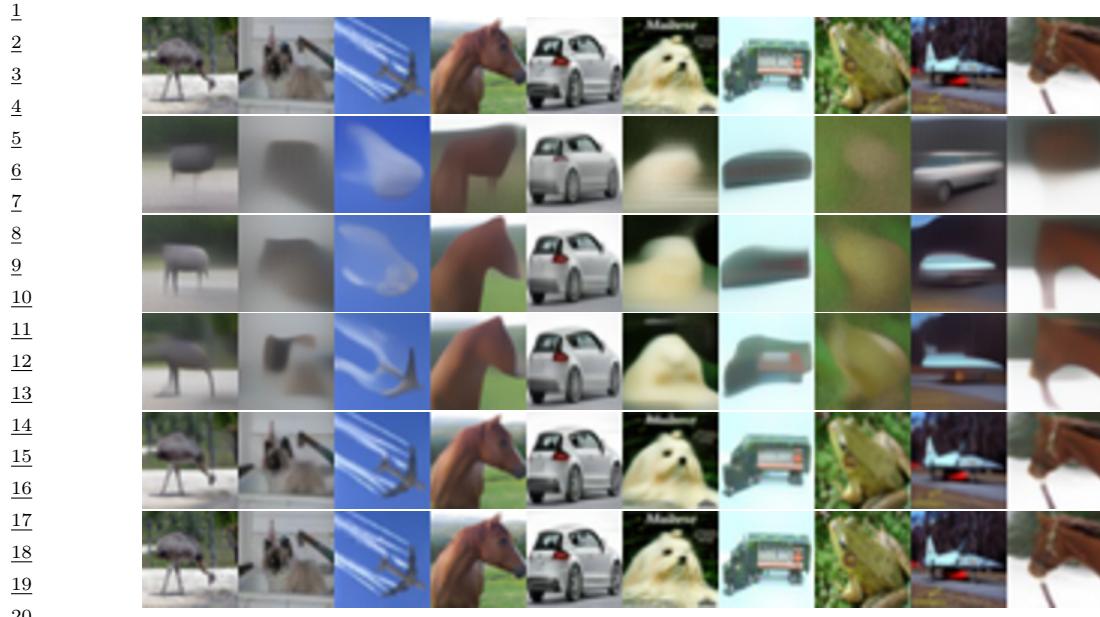
We can also use the model for unconditional sampling at multiple resolutions. This is illustrated in Figure 22.19, using a model with 78 stochastic layers trained on the FFHQ-256 dataset.¹²

22.5.3 Connection with autoregressive models

Until recently, most hierarchical VAEs only had a small number of stochastic layers. Consequently the images they generated have not looked as good, or had as high likelihoods, as images produced by other models, such as the autoregressive PixelCNN model (see Section 23.3.2). However, by endowing VAEs with many more stochastic layers, it is possible to outperform AR models in terms of likelihood and sample quality, while using fewer parameters and much less computing power [Chi21; VK20; Maa+19].

To see why this is possible, note that we can represent any AR model as a degenerate VAE, as shown in Figure 22.20(left). The idea is simple: the encoder copies the input into latent space by setting $\mathbf{z}_{1:D} = \mathbf{x}_{1:D}$ (so $q_\phi(\mathbf{z}_i = \mathbf{x}_i | \mathbf{z}_{>i}, \mathbf{x}) = 1$), then the model learns an autoregressive prior $p_\theta(\mathbf{z}_{1:D}) = \prod_d p(\mathbf{z}_d | \mathbf{z}_{1:d-1})$, and finally the likelihood function just copies the latent vector to output space, so $p_\theta(\mathbf{x}_i = \mathbf{z}_i | \mathbf{z}) = 1$. Since the encoder computes the exact (albeit degenerate) posterior, we

¹² 12. This is a 256² version of the Flickr-Faces High Quality dataset from <https://github.com/NVlabs/ffhq-dataset>, which has 80k images at 1024² resolution.



21 *Figure 22.18: Reconstruction from the VD-VAE model trained on CIFAR10 using different numbers of latent
22 layers. The latent code is a sample from the posterior for the first $L_1 < L$ layers, and then is sampled from
23 the prior (ignoring higher levels) at a low temperature; this emulates models of different stochastic depth.
24 Top row: input images. Subsequent rows: reconstructing down to resolutions 1, 4, 8, 16, 32. Generated by
25 vdvae_flax_demo_cifar.ipynb.*



33 *Figure 22.19: Samples from a VDVAE model (trained on FFHQ dataset) from different levels of the hierarchy.
34 From Figure 1 of [Chi21]. Used with kind permission of Ronan Child.*

37 have $q_{\phi}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{z}|\mathbf{x})$, so the ELBO is tight and reduces to the log likelihood,
38

$$39 \quad \log p_{\theta}(\mathbf{x}) = \log p_{\theta}(\mathbf{z}) = \sum_d \log p_{\theta}(x_d | \mathbf{x}_{<d}) \quad (22.121)$$

42 Thus we can emulate any AR model with a VAE providing it has at least D stochastic layers, where
43 D is the dimensionality of the observed data.

44 In practice data usually lives in a lower-dimensional manifold (see e.g., [DW19]), which can allow
45 for a much more compact latent code. For example, Figure 22.20(right) shows a hierarchical code
46 in which the latent factors at the lower level are conditionally independent given the higher level,
47

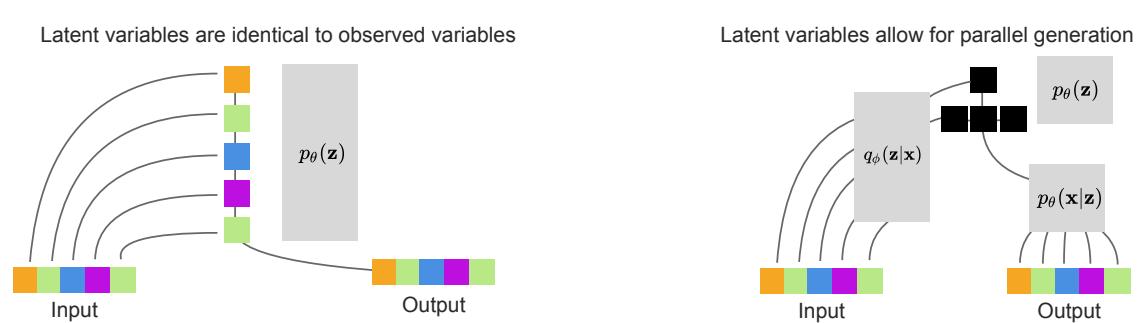


Figure 22.20: Left: a hierarchical VAE which emulates an autoregressive model using an identity encoder, autoregressive prior, and identity decoder. Right: a hierarchical VAE with a 2 layer hierarchical latent code. The bottom hidden nodes (black) are conditionally independent given the top layer. From Figure 2 of [Chi21]. Used with kind permission of Rewon Child.

and hence can be generated in parallel. Such a tree-like structure can enable sample generation in $O(\log D)$ time, whereas an autoregressive model always takes $O(D)$ time. (Recall that for an image D is the number of pixels, so it grows quadratically with image resolution. For example, even a tiny 32x32 image has $D = 3072$.)

In addition to speed, hierarchical models also require many fewer parameters than “flat” models. The typical architecture used for generating images is a **multi-scale** approach: the model starts from a small, spatially arranged set of latent variables, and at each subsequent layer, the spatial resolution is increased (usually by a factor of 2). This allows the high level to capture global, long-range correlations (e.g., the symmetry of a face, or overall skin tone), while letting lower levels capture fine-grained details.

22.5.4 Variational pruning

A common problem with hierarchical VAEs is that the higher level latent layers are often ignored, so the model does not learn interesting high level semantics. This is caused by **variational pruning**. This problem is analogous to the issue of latent variable collapse, which we discussed in Section 22.4 in the context of a single layer of latents with expressive decoders, such as RNNs; in the context of HVAEs, the “expressive decoder” corresponds to lower layers of the hierarchy.

A common heuristic to mitigate this problem is to use KL balancing coefficients [Che+17b], to ensure that an equal amount of information is encoded in each layer. That is, we use the following penalty:

$$\sum_{l=1}^L \gamma_l \mathbb{E}_{q_\phi(z_{>l}|\mathbf{x})} [D_{\text{KL}}(q_\phi(z_l|\mathbf{x}, z_{>l}) \| p_\theta(z_l|z_{>l}))] \quad (22.122)$$

The balancing term γ_l is set to a small value when the KL penalty is small (on the current minibatch), to encourage use of that layer, and is set to a large value when the KL term is large. (This is only done during the “warmup period.”) Concretely, [VK20] proposes to set the coefficients γ_l to be

1 proportional to the size of the layer, s_l , and the average KL loss:

2

$$\gamma_l \propto s_l \mathbb{E}_{\mathbf{x} \sim \mathcal{B}} [\mathbb{E}_{q_{\phi}(\mathbf{z}_{>l} | \mathbf{x})} [D_{\text{KL}}(q_{\phi}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{>l}) \| p_{\theta}(\mathbf{z}_l | \mathbf{z}_{>l}))]] \quad (22.123)$$

3 where \mathcal{B} is the current minibatch.

4 22.5.5 Other optimization difficulties

5 A common problem when training (hierarchical) VAEs is that the loss can become unstable. The
6 main reason for this is that the KL term is unbounded (can become infinitely large). In [Chi21], they
7 tackle the problem in two ways. First, ensure the initial random weights of the final convolutional
8 layer in each residual bottleneck block get scaled by $1/\sqrt{L}$. Second, skip an update step if the norm
9 of the gradient of the loss exceeds some threshold.

10 In the **Nouveau VAE** method of [VK20], they use some more complicated measures to ensure
11 stability. First they use batch normalization, but with various tweaks. Second they use spectral
12 regularization for the encoder. Specifically they add the penalty $\beta \sum_i \lambda_i$, where λ_i is the largest
13 singular value of the i 'th convolutional layer (estimated using a single power iteration step), and
14 $\beta \geq 0$ is a tuning parameter. Third, they use inverse autoregressive flows (Section 24.2.4.3) in each
15 layer, instead of a diagonal Gaussian approximation. Fourth, they represent the posterior using a
16 residual representation. In particular, let us assume the prior for the i 'th variable in layer l is

17

$$p_{\theta}(z_l^i | \mathbf{z}_{>l}) = \mathcal{N}(z_l^i | \mu_i(\mathbf{z}_{>l}), \sigma_i(\mathbf{z}_{>l})) \quad (22.124)$$

18 They propose the following posterior approximation:

19

$$q_{\phi}(z_l^i | \mathbf{x}, \mathbf{z}_{>l}) = \mathcal{N}(z_l^i | \mu_i(\mathbf{z}_{>l}) + \Delta\mu_i(\mathbf{z}_{>l}, \mathbf{x}), \sigma_i(\mathbf{z}_{>l}) \cdot \Delta\sigma_i(\mathbf{z}_{>l}, \mathbf{x})) \quad (22.125)$$

20 where the Δ terms are the relative changes computed by the encoder. The corresponding KL penalty
21 reduces to the following (dropping the l subscript for brevity):

22

$$D_{\text{KL}}(q_{\phi}(z^i | \mathbf{x}, \mathbf{z}_{>l}) \| p_{\theta}(z^i | \mathbf{z}_{>l})) = \frac{1}{2} \left(\frac{\Delta\mu_i^2}{\sigma_i^2} + \Delta\sigma_i^2 - \log \Delta\sigma_i^2 - 1 \right) \quad (22.126)$$

23 So as long as σ_i is bounded from below, the KL term can be easily controlled just by adjusting the
24 encoder parameters.

25 22.6 Vector quantization VAE

26 In this section, we describe **VQ-VAE**, which stands for “vector quantized VAE” [OVK17; ROV19].

27 This is like a standard VAE except it uses a set of discrete latent variables.

28

29 22.6.1 Autoencoder with binary code

30 The simplest approach to the problem is to construct a standard VAE, but to add a discretization
31 layer at the end of the encoder, $\mathbf{z}_e(\mathbf{x}) \in \{0, \dots, S-1\}^K$, where S is the number of states, and K is
32 the number of discrete latents. For example, we can binarize the latent vector (using $S=2$) using

33



Figure 22.21: Autoencoder for MNIST using 256 binary latents. Top row: input images. Middle row: reconstruction. Bottom row: latent code, reshaped to a 16×16 image. Generated by [quanzified_autoencoder_mnist.ipynb](#).

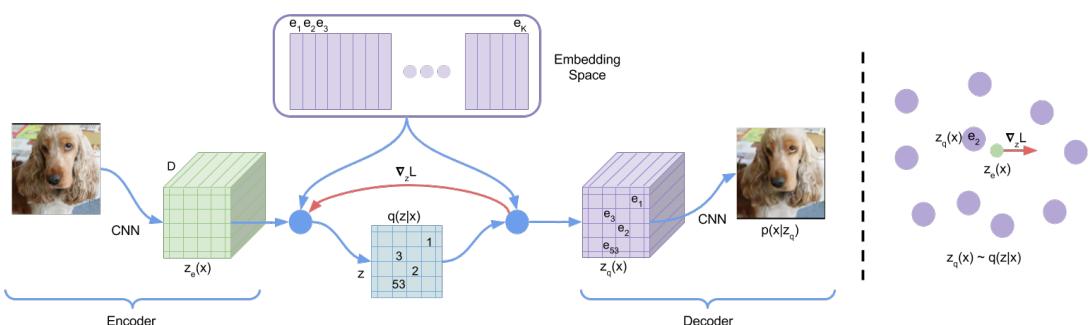


Figure 22.22: VQ-VAE architecture. From Figure 1 of [OVK17]. Used with kind permission of Aaron van den Oord.

a ReLU function, so the code becomes $\mathbf{z} = \text{ReLU}(\mathbf{z}_e(\mathbf{x})) \in \{0, 1\}^K$. This can be useful for data compression (see e.g., [BLS17]).

Suppose we assume the prior over the latent codes is uniform. Since the encoder is deterministic, the KL divergence reduces to a constant, equal to $\log K$. This avoids the problem with posterior collapse (Section 22.4). Unfortunately, the discreteness of the encoder prohibits the direct use of gradient based optimization. The solution proposed in [OVK17] is to use the straight-through estimator, which we discuss in Section 6.6.8. We show a simple example of this approach in Figure 22.21, where we use a Gaussian likelihood, so the loss function has the form

$$\mathcal{L} = \|\mathbf{x} - D(\mathbf{z})\|_2^2 \quad (22.127)$$

where $E(\mathbf{x}) \in \{0, 1\}^K$ is the encoder, and $D(\mathbf{z}) \in \mathbb{R}^{28 \times 28}$ is the decoder.

1 **22.6.2 VQ-VAE model**

3 We can get a more expressive model by using a 3d tensor of discrete latents, $\mathbf{z} \in \mathbb{R}^{H \times W \times K}$, where
4 K is the number of discrete values per latent variable. Rather than just binarizing the continuous
5 vector $\mathbf{z}_e(\mathbf{x})_{ij}$, we compare it to a **codebook** of embedding vectors, $\{\mathbf{e}_k : k = 1 : K, \mathbf{e}_k \in \mathbb{R}^L\}$, and
6 then set \mathbf{z}_{ij} to the index of the nearest codebook entry:
7

$$\underline{8} \quad q(\mathbf{z}_{ij} = k | \mathbf{x}) = \begin{cases} \underline{9} \quad 1 & \text{if } k = \operatorname{argmin}_{k'} \|\mathbf{z}_e(\mathbf{x})_{i,j,:} - \mathbf{e}_{k'}\|_2 \\ \underline{10} \quad 0 & \text{otherwise} \end{cases} \quad (22.128)$$

11 When reconstructing the input we replace each discrete code index by the corresponding real-valued
12 codebook vector:

$$\underline{13} \quad (\mathbf{z}_q)_{ij} = \mathbf{e}_k \text{ where } \mathbf{z}_{ij} = k \quad (22.129)$$

17 These values are then passed to the decoder, $p(\mathbf{x} | \mathbf{z}_q)$, as usual. See Figure 22.22 for an illustration of
18 the overall architecture. Note that although \mathbf{z}_q is generated from a discrete combination of codebook
19 vectors, the use of a distributed code makes the model very expressive. For example, if we use a
20 grid of 32×32 , with $K = 512$, then we can generate $512^{32 \times 32} = 2^{9216}$ distinct images, which is
21 astronomically large.

22 To fit this model, we can minimize the negative log likelihood (reconstruction error) using the
23 straight-through estimator, as before. This amounts to passing the gradients from the decoder input
24 $\mathbf{z}_q(\mathbf{x})$ to the encoder output $\mathbf{z}_e(\mathbf{x})$, bypassing Equation (22.128), as shown by the red arrow in
25 Figure 22.22. Unfortunately this means that the codebook entries will not get any learning signal.
26 To solve this, the authors proposed to add an extra term to the loss, known as the **codebook loss**,
27 that encourages the codebook entries \mathbf{e} to match the output of the encoder. We treat the encoder
28 $\mathbf{z}_e(\mathbf{x})$ as a fixed target, by adding a **stop gradient** operator to it; this ensures \mathbf{z}_e is treated normally
29 in the forwards pass, but has zero gradient in the backwards pass. The modified loss (dropping the
30 spatial indices i, j) becomes

$$\underline{31} \quad \mathcal{L} = -\log p(\mathbf{x} | \mathbf{z}_q(\mathbf{x})) + \|\operatorname{sg}(\mathbf{z}_e(\mathbf{x})) - \mathbf{e}\|_2^2 \quad (22.130)$$

34 where \mathbf{e} refers to the codebook vector assigned to $\mathbf{z}_e(\mathbf{x})$.

35 An alternative way to update the codebook vectors is to use moving averages. To see how this
36 works, first consider the batch setting. Let $\{\mathbf{z}_{i,1}, \dots, \mathbf{z}_{i,n_i}\}$ be the set of n_i outputs from the encoder
37 that are closest to the dictionary item \mathbf{e}_i . We can update \mathbf{e}_i to minimize the MSE

$$\underline{39} \quad \sum_{j=1}^{n_i} \|\mathbf{z}_{i,j} - \mathbf{e}_i\|_2^2 \quad (22.131)$$

42 which has the closed form update

$$\underline{44} \quad \mathbf{e}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{z}_{i,j} \quad (22.132)$$

This is like the M step of the EM algorithm when fitting the mean vectors of a GMM. In the minibatch setting, we replace the above operations with an exponentially moving average, as follows:

$$\underline{N}_i^t = \gamma \underline{N}_i^{t-1} + (1 - \gamma) \underline{n}_i^t \quad (22.133)$$

$$\underline{\mathbf{m}}_i^t = \gamma \underline{\mathbf{m}}_i^{t-1} + (1 - \gamma) \sum_j \underline{\mathbf{z}}_{i,j}^t \quad (22.134)$$

$$\underline{\mathbf{e}}_i^t = \frac{\underline{\mathbf{m}}_i^t}{\underline{N}_i^t} \quad (22.135)$$

The authors found $\gamma = 0.9$ to work well.

The above procedure will learn to update the codebook vectors so it match the output of the encoder. However, it is also important to ensure the encoder does not “change its mind” too often about what codebook value to use. To prevent this, the authors propose to add a third term to the loss, known as the **commitment loss**, that encourages the encoder output to be close to the codebook values. Thus we get the final loss:

$$\underline{\mathcal{L}} = -\log p(\mathbf{x}|\mathbf{z}_q(\mathbf{x})) + \|\text{sg}(\mathbf{z}_e(\mathbf{x})) - \mathbf{e}\|_2^2 + \beta \|\mathbf{z}_e(\mathbf{x}) - \text{sg}(\mathbf{e})\|_2^2 \quad (22.136)$$

The authors found $\beta = 0.25$ to work well, although of course the value depends on the scale of the reconstruction loss (NLL) term. (A probabilistic interpretation of this loss can be found in [Hen+18].) Overall, the decoder optimizes the first term only, the encoder optimizes the first and last term, and the embeddings optimize the middle term.

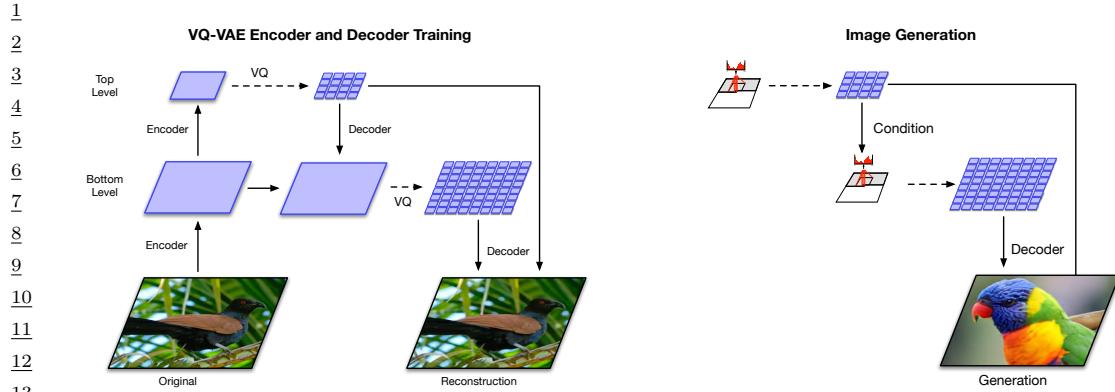
22.6.3 Learning the prior

After training the VQ-VAE model, it is possible to learn a better prior, to match the aggregated posterior. To do this, we just apply the encoder to a set of data, $\{\mathbf{x}_n\}$, thus converting them to discrete sequences, $\{\mathbf{z}_n\}$. We can then learn a joint distribution $p(\mathbf{z})$ using any kind of sequence model. In the original VQ-VAE paper [OVK17], they used the causal convolutional PixelCNN model (Section 23.3.2). More recent work has used transformer decoders (Section 23.4). Samples from this prior can then be decoded using the decoder part of the VQ-VAE model. We give some examples of this in the sections below.

22.6.4 Hierarchical extension (VQ-VAE-2)

In [ROV19], they extend the original VQ-VAE model by using a hierarchical latent code. The model is illustrated in Figure 22.23. They applied this to images of size $256 \times 256 \times 3$. The first latent layer maps this to a quantized representation of size 64×64 , and the second latent layer maps this to a quantized representation of size 32×32 . This hierarchical scheme allows the top level to focus on high level semantics of the image, leaving fine visual details, such as texture, to the lower level. (See Section 22.5 for more discussion of hierarchical VAEs.)

After fitting the VQ-VAE, they learn a prior over the top level code using a PixelCNN model augmented with self-attention (Section 16.2.7), to capture long-range dependencies. (This hybrid model is known as **PixelSNAIL** [Che+17c].) For the lower level prior, they just use standard PixelCNN, since attention would be too expensive. Samples from the model can then be decoded using the VQ-VAE decoder, as shown in Figure 22.23.



14 Figure 22.23: Hierarchical extension of VQ-VAE. (a) Encoder and decoder architecture.
15 (b) Combining a
16 Pixel-CNN prior with the decoder. From Figure 2 of [ROV19]. Used with kind permission of Aaron van den
17 Oord.



31 Figure 22.24: Some images generated by a class conditional VQ-VAE-2 model trained on Imagenet. Row 1:
32 Label = "Pekinese". Row 1: Label = "Papillon". Row 2: Label = "Drake" Row 3: Label = "spotted salamander".
33 From Figure 4 of [ROV19]. Used with kind permission of Aaron van den Oord.

36 Some samples from a class-conditional version of this model, after training on ImageNet at 256×256
37 resolution, are shown in Figure 22.24. They show good visual quality and diversity.

39 22.6.5 Discrete VAE

41 In VQ-VAE, we use a one-hot encoding for the latents, $q(z = k|\mathbf{x}) = 1$ iff $k = \operatorname{argmin}_k \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_k\|_2$,
42 and then set $\mathbf{z}_q = \mathbf{e}_k$. This does not capture any uncertainty in the latent code, and requires the use
43 of the straight-through estimator for training.

44 Various other approaches to fitting VAEs with discrete latent codes have been investigated. In the
45 DALL-E paper (Section 23.4.3), they use a fairly simple method, based on using the Gumbel-Softmax
46 relaxation for the discrete variables (see Section 6.6.6). In brief, let $q(z = k|\mathbf{x})$ be the probability
47

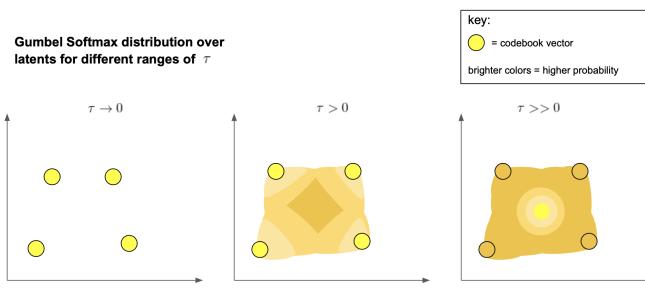


Figure 22.25: Illustration of the Gumbel Softmax trick applied to $K = 4$ codebook vectors in $L = 2$ dimensions. From <https://ml.berkeley.edu/blog/posts/dalle2/>. Used with kind permission of Charlie Snell.

that the input \mathbf{x} is assigned to codebook entry i . We can exactly sample $w_k \sim q(z = k|\mathbf{x})$ from this by computing $w_k = \text{argmax}_i g_k + \log q(z = k|\mathbf{x})$, where each g_k is from a Gumbel distribution. We can now “relax” this by using a softmax with temperature $\tau > 0$ and computing

$$w_k = \frac{\exp(\frac{g_k + \log q(z=k|\mathbf{x})}{\tau})}{\sum_{j=1}^K \exp(\frac{g_j + \log q(z=j|\mathbf{x})}{\tau})} \quad (22.137)$$

We now set the latent code to be a weighted sum of the codebook vectors:

$$\mathbf{z}_q = \sum_{k=1}^K w_k \mathbf{e}_k \quad (22.138)$$

In the limit that $\tau \rightarrow 0$, the distribution over weights \mathbf{w} converges to a one-hot distribution, in which case \mathbf{z} becomes equal to one of the codebook entries. But for finite τ , we “fill in” the space between the vectors, as illustrated in Figure 22.25.

This allows us to express the ELBO in the usual differentiable way:

$$\mathcal{L} = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x}|\mathbf{z})] + \beta D_{\text{KL}}(q(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) \quad (22.139)$$

where $\beta > 0$ controls the amount of regularization. (Unlike VQ-VAE, the KL term is not a constant, because the encoder is stochastic.) Furthermore, since the Gumbel noise variables are sampled from a distribution that is independent of the encoder parameters, we can use the reparameterization trick (Section 6.6.4) to optimize this.

The overall architecture is illustrated in Figure 22.26. (The “avocado chair” image is discussed in more detail on the section on DALL-E in Section 23.4.3.)

22.6.6 VQ-GAN

One drawback of VQ-VAE is that it uses mean squared error in its reconstruction loss, which can result in blurry samples. In the **VQ-GAN** paper [ERO21], they replace this with a (patch-wise) GAN loss (see Chapter 27), together with a perceptual loss; this results in much higher visual fidelity. In addition, they use a transform (see Section 16.3.4) to model the prior on the latent codes. See

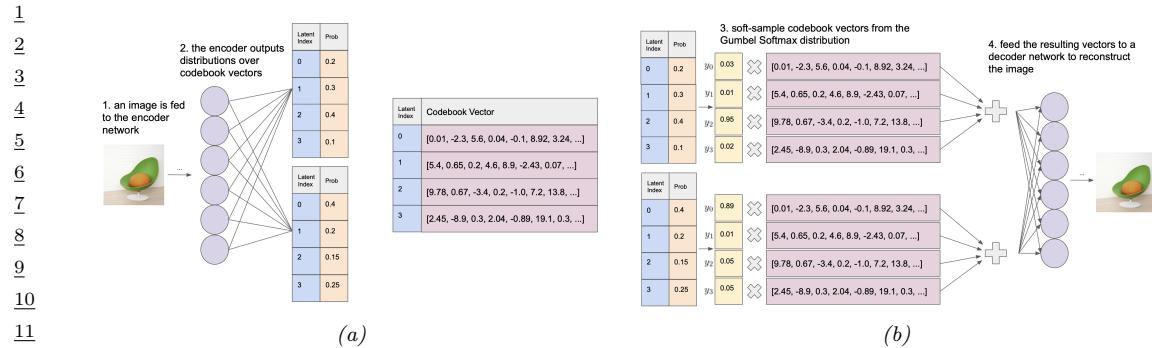


Figure 22.26: Illustration of the dVAE model (a) Encoder. (b) Decoder. From <https://ml.berkeley.edu/blog/posts/dalle2/>. Used with kind permission of Charlie Snell.

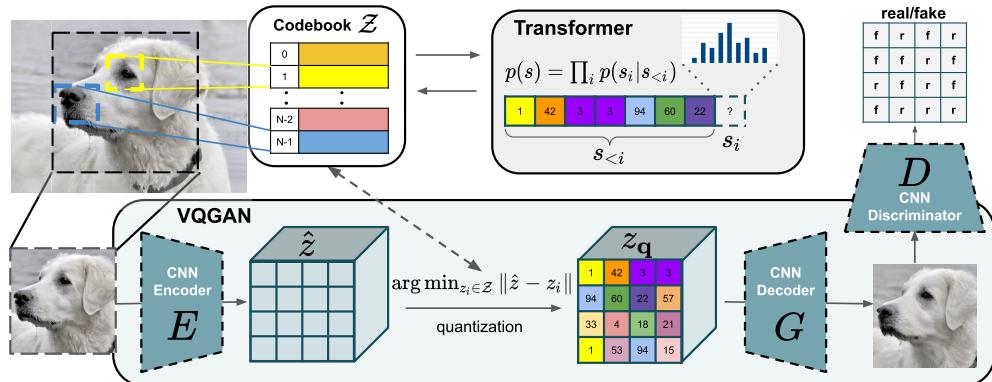


Figure 22.27: Illustration of the VQ-GAN. From Figure 2 of [ERO21]. Used with kind permission of Patrick Esser.

Figure 22.27 for a visualization of the overall model. In [Yu+21], they replace the CNN encoder and decoder of the VQ-GAN model with transformers, yielding improved results; they call this **VIM** (Vector-quantized Image Modeling).

22.7 Wake-sleep algorithm

So far in this chapter we have focused on fitting latent variable models by maximizing the ELBO. This has two main drawbacks. First, it does not work well when we have discrete latent variables, because in such cases we cannot use the reparameterization trick; thus we have to use higher variance estimators, such as REINFORCE (see Section 10.3.1). Second, even in the case where we can use the reparameterization trick, the lower bound may not be very tight. We can improve the tightness by using the IWAE multi-sample bound (Section 10.5.1), but paradoxically this may not result in learning a better model, for reasons discussed in Section 10.5.1.1.

In this section, we discuss a different way to jointly train generative and inference models, which

avoids some of the problems with ELBO maximization. The method is known as the **wake-sleep algorithm** [Hin+95; BB15b; Le+19; FT19]. because it alternates between two steps: in the wake phase, we optimize the generative model parameters θ to maximize the marginal likelihood of the observed data (we approximate $\log p_\theta(\mathbf{x})$ by drawing importance samples from the inference network); and in the sleep phase, we optimize the inference model parameters ϕ to learn to invert the generative model by training the inference network on labeled (\mathbf{x}, \mathbf{z}) pairs, where \mathbf{x} are samples generated by the current model parameters. This can be viewed as a form of **adaptive importance sampling**, which iteratively improves its proposal, while simultaneously optimizing the model. We give further details below.

22.7.1 Wake phase

In the **wake phase**, we minimize the KL divergence from the empirical distribution to the model's distribution:

$$\mathcal{L}(\theta) = D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\theta}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\log p_{\theta}(\mathbf{x})] + \text{const} \quad (22.140)$$

where $p_{\theta}(\mathbf{x}) = \int p_{\theta}(\mathbf{z}) p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z}$. This is equivalent to maximizing the likelihood of the observed data:

$$LL(\theta) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\log p_{\theta}(\mathbf{x})] \quad (22.141)$$

Since the log marginal likelihood $\log p_{\theta}(\mathbf{x})$ cannot be computed exactly, we will approximate it. In the original wake sleep paper, they proposed to use the ELBO lower bound. In the **reweighted wake sleep** (RWS) algorithm of [BB15b; Le+19], they propose to use the IWAE bound from Section 10.5.1 instead. In particular, if we draw S samples from the inference network, $\mathbf{z}_s \sim q_{\phi}(\mathbf{z}|\mathbf{x})$, we get the following estimator:

$$LL(\theta|\phi, \mathbf{x}) = \log \left(\frac{1}{S} \sum_{s=1}^S w_s \right) \quad (22.142)$$

where $w_s = \frac{p_{\theta}(\mathbf{x}, \mathbf{z}_s)}{q_{\phi}(\mathbf{z}_s | \mathbf{x})}$.

We now discuss how to compute the gradient of this objective. Using the log-derivative trick, we have that

$$\nabla_{\theta} \log w_s = \frac{1}{w_s} \nabla_{\theta} w_s = \nabla_{\theta} \log p_{\theta}(\mathbf{x}, \mathbf{z}_s) \quad (22.143)$$

Hence

$$\nabla_{\theta} LL(\theta|\phi, \mathbf{x}) = \frac{1}{\frac{1}{S} \sum_{s=1}^S w_s} \left(\frac{1}{S} \sum_{s=1}^S \nabla_{\theta} w_s \right) \quad (22.144)$$

$$= \frac{1}{\sum_{s=1}^S w_s} \left(\sum_{s=1}^S \frac{p_{\theta}(\mathbf{z}_s, \mathbf{x})}{q_{\phi}(\mathbf{z}_s | \mathbf{x})} \nabla_{\theta} \log p_{\theta}(\mathbf{z}_s, \mathbf{x}) \right) \quad (22.145)$$

$$= \sum_{s=1}^S \bar{w}_s \nabla_{\theta} \log p_{\theta}(\mathbf{z}_s, \mathbf{x}) \quad (22.146)$$

where $\bar{w}_s = w_s / (\sum_{s'=1}^S w_{s'})$.

1 **22.7.2 Sleep phase**

3 In the **sleep phase**, we try to minimize the KL divergence between the true posterior (under the
4 current model) and the inference network’s approximation to that posterior:

5

$$\mathcal{L}(\phi) = \mathbb{E}_{p_{\theta}(\mathbf{x})} [D_{\text{KL}}(p_{\theta}(\mathbf{z}|\mathbf{x}) \| q_{\phi}(\mathbf{z}|\mathbf{x}))] = \mathbb{E}_{p_{\theta}(\mathbf{z}, \mathbf{x})} [-\log q_{\phi}(\mathbf{z}|\mathbf{x})] + \text{const} \quad (22.147)$$

6 Equivalently, we can maximize the following loglikelihood objective:

7

$$\mathcal{L}(\phi|\theta) = \mathbb{E}_{(\mathbf{z}, \mathbf{x}) \sim p_{\theta}(\mathbf{z}, \mathbf{x})} [\log q_{\phi}(\mathbf{z}|\mathbf{x})] \quad (22.148)$$

8 where $p_{\theta}(\mathbf{z}, \mathbf{x}) = p_{\theta}(\mathbf{z})p_{\theta}(\mathbf{x}|\mathbf{z})$. We see that the sleep phase amounts to maximum likelihood training
9 of the inference network based on samples from the generative model. These “fantasy samples”,
10 created while the network “dreams”, can be easily generated using ancestral sampling (Section 4.2.4).
11 If we use S such samples, the objective becomes

12

$$\mathcal{L}(\phi|\theta) = \frac{1}{S} \sum_{s=1}^S \log q_{\phi}(\mathbf{z}'_s | \mathbf{x}'_s) \quad (22.149)$$

13 where $(\mathbf{z}'_s, \mathbf{x}'_s) \sim p_{\theta}(\mathbf{z}, \mathbf{x})$. The gradient of this is given by

14

$$\nabla_{\phi} \mathcal{L}(\phi|\theta) = \frac{1}{S} \sum_{s=1}^S \nabla_{\phi} \log q_{\phi}(\mathbf{z}'_s | \mathbf{x}'_s) \quad (22.150)$$

15 We do not require $q_{\phi}(\mathbf{z}'|\mathbf{x})$ to be reparameterizable, since the samples are drawn from a distribution
16 that is independent of ϕ . This means it is easy to apply this method to models with discrete latent
17 variables.

18 Note that the technique of training an inference network to invert a known generative model
19 using supervised learning on generated samples is known as **inference compilation** [LBW17] or
20 **amortized inference** [GG14].

21 **22.7.3 Daydream phase**

22 The disadvantage of the sleep phase is that the inference network, $q_{\phi}(\mathbf{z}|\mathbf{x})$, is trying to follow a
23 moving target, $p_{\theta}(\mathbf{z}|\mathbf{x})$. Furthermore, it is only being trained on synthetic data from the model,
24 not on real data. The reweighted wake sleep algorithm of [BB15b] proposed to learn the inference
25 network by using real data from the empirical distribution, in addition to fantasy data. They call
26 the case where you use real data the “**wake-phase q update**”, but we will call it the “**daydream**
27 **phase**”, since, unlike sleeping, the system uses real data \mathbf{x} to update the inference model, instead of
28 fantasies.¹³ [Le+19] went further, and proposed to only use the wake and daydream phases, and to
29 skip the sleep phase entirely.

30 In more detail, the new objective which we want to minimize becomes

31

$$\mathcal{L}(\phi|\theta) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(p_{\theta}(\mathbf{z}|\mathbf{x}) \| q_{\phi}(\mathbf{z}|\mathbf{x}))] \quad (22.151)$$

32 13. We thank Rif Sauros for suggesting this term.

We can compute a single sample approximation to the negative of the above expression as follows:

$$LL(\phi|\theta, \mathbf{x}) = \mathbb{E}_{p_\theta(\mathbf{z}|\mathbf{x})} [\log q_\phi(\mathbf{z}|\mathbf{x})] \quad (22.152)$$

where $\mathbf{x} \sim p_{\mathcal{D}}$. We can approximate this expectation using importance sampling, with q_ϕ as the proposal. This results in the following estimator of the gradient for each datapoint:

$$\nabla_\phi LL(\phi|\theta, \mathbf{x}) = \int p_\theta(\mathbf{z}|\mathbf{x}) \nabla_\phi \log q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \approx \sum_{s=1}^S \bar{w}_s \nabla_\phi \log q_\phi(\mathbf{z}_s|\mathbf{x}) \quad (22.153)$$

where $\mathbf{z}_s \sim q_\phi(\mathbf{z}_s|\mathbf{x})$ and \bar{w}_s are the normalized weights.

We see that Equation (22.153) is very similar to Equation (22.150). The key difference is that in the daydream phase, we sample from $(\mathbf{x}, \mathbf{z}_s) \sim p_{\mathcal{D}}(\mathbf{x})q_\phi(\mathbf{z}|\mathbf{x})$, where \mathbf{x} is a real data point, whereas in the sleep phase, we sample from $(\mathbf{x}'_s, \mathbf{z}'_s) \sim p_\theta(\mathbf{z}, \mathbf{x})$, where \mathbf{x}'_s is generated data point.

22.7.4 Summary of algorithm

Algorithm 27: One SGD update using wake-sleep algorithm.

- 1 Sample \mathbf{x}_n from dataset ;
 - 2 Draw S samples from inference network: $\mathbf{z}_s \sim q(\mathbf{z}|\mathbf{x}_n)$;
 - 3 Compute unnormalized weights: $w_s = \frac{p(\mathbf{x}_n, \mathbf{z}_s)}{q(\mathbf{z}_s|\mathbf{x}_n)}$;
 - 4 Compute normalized weights: $\bar{w}_s = \frac{w_s}{\sum_{s'=1}^S w_{s'}}$;
 - 5 Optional: Compute estimate of log likelihood: $\log p(\mathbf{x}_n) = \log(\frac{1}{S} \sum_{s=1}^S w_s)$;
 - 6 Wake phase: Update θ using $\sum_{s=1}^S \bar{w}_s \nabla_\theta \log p_\theta(\mathbf{z}_s, \mathbf{x}_n)$;
 - 7 Daydream phase: Update ϕ using $\sum_{s=1}^S \bar{w}_s \nabla_\phi \log q_\phi(\mathbf{z}_s|\mathbf{x}_n)$;
 - 8 Optional sleep phase: Draw S samples from model, $(\mathbf{x}'_s, \mathbf{z}'_s) \sim p_\theta(\mathbf{x}, \mathbf{z})$ and update ϕ using $\frac{1}{S} \sum_{s=1}^S \nabla_\phi \log q_\phi(\mathbf{z}'_s|\mathbf{x}'_s)$
-

We summarize the RWS algorithm in Algorithm 27. The disadvantage of the RWS algorithm is that it does not optimize a single well-defined objective, so it is not clear if the method will converge, in contrast to ELBO maximization. On the other hand, the method is fairly simple, since it consists of two alternating weighted maximum likelihood problems. It can also be shown to “sandwich” a lower and upper bound of the log marginal likelihood. We can think of this in terms of the two joint distributions $p_\theta(\mathbf{x}, \mathbf{z})$ and $q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z}) = p_{\mathcal{D}}(\mathbf{x})q_\phi(\mathbf{z}|\mathbf{x})$:

$$\text{wake phase } \min_{\theta} D_{\text{KL}}(q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z}) \| p_\theta(\mathbf{x}, \mathbf{z})) \quad (22.154)$$

$$\text{daydream phase } \min_{\phi} D_{\text{KL}}(p_\theta(\mathbf{x}, \mathbf{z}) \| q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z})) \quad (22.155)$$

23 Auto-regressive models

23.1 Introduction

By the chain rule of probability, we can write any joint distribution over T variables as follows:

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2, \mathbf{x}_1)p(\mathbf{x}_4|\mathbf{x}_3, \mathbf{x}_2, \mathbf{x}_1)\dots = \prod_{t=1}^T p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) \quad (23.1)$$

where $\mathbf{x}_t \in \mathcal{X}$ is the t 'th observation, and we define $p(\mathbf{x}_1|\mathbf{x}_{1:0}) = p(x_1)$ as the initial state distribution. This is called an **auto-regressive model**. This corresponds to a fully connected DAG, in which each node depends on all its predecessors in the ordering, as shown in Figure 23.1. The models can also be conditioned on arbitrary inputs or context \mathbf{c} , in order to define $p(\mathbf{x}|\mathbf{c})$, although we omit this for notational brevity.

We could of course also factorize the joint distribution “backwards” in time, using

$$p(\mathbf{x}_{1:T}) = \prod_{t=T}^1 p(\mathbf{x}_t|\mathbf{x}_{t+1:T}) \quad (23.2)$$

However, this “anti-causal” direction is often harder to learn (see e.g., [PJS17]).

Although the decomposition in Equation (23.1) is general, each term in this expression (i.e., each conditional distribution $p(\mathbf{x}_t|\mathbf{x}_{1:t-1})$) becomes more and more complex, since it depends on an increasing number of arguments, which makes the terms slow to compute, and makes estimating their parameters more data hungry (see Section 2.8.3.2).

One approach to solving this intractability is to make the (first-order) **Markov assumption**, which gives rise to a **Markov model** $p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = p(\mathbf{x}_t|\mathbf{x}_{t-1})$, which we discuss in Section 2.8. (This is also called an auto-regressive model of order 1.) Unfortunately, the Markov assumption is very limiting. One way to relax it, and to make \mathbf{x}_t depend on all the past $\mathbf{x}_{1:t-1}$ without explicitly regressing on them, is to assume the past can be compressed into a **hidden state** \mathbf{z}_t . If \mathbf{z}_t is a deterministic function of the past observations $\mathbf{x}_{1:t-1}$, the resulting model is known as a **recurrent neural network**, discussed in Section 16.3.3. If \mathbf{z}_t is a stochastic function of the past hidden state, \mathbf{z}_{t-1} , the resulting model is known as a **hidden Markov model**, which we discuss in Section 30.1.

Another approach is to stay with the general AR model of Equation (23.1), but to use a restricted functional form, such as some kind of neural network, for the conditionals $p(\mathbf{x}_t|\mathbf{x}_{1:t-1})$. Thus rather than making conditional independence assumptions, or explicitly compressing the past into a sufficient

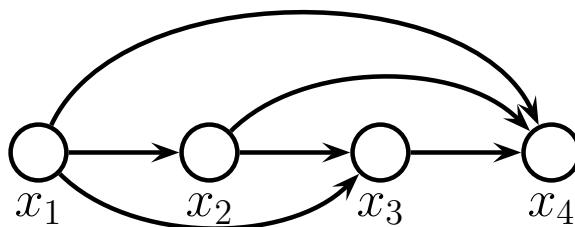


Figure 23.1: A fully-connected auto-regressive model.

statistic, we implicitly learn a compact mapping from the past to the future. In the sections below, we discuss different functional forms for these conditional distributions.

The main advantage of such AR models is that it is easy to compute, and optimize, the exact likelihood of each sequence (data vector). The main disadvantage is that generating samples is inherently sequential, which can be slow. In addition, the method does not learn a compact latent representation of the data.

19

20 23.2 Neural autoregressive density estimators (NADE)

21

22 A simple way to represent each conditional probability distribution $p(x_t | \mathbf{x}_{1:t-1})$ is to use a generalized
23 linear model, such as logistic regression, as proposed in [Fre98]. We can make the model be more
24 powerful by using a neural network. The resulting model is called the **neural auto-regressive**
25 **density estimator** or **NADE** model [LM11].

26 If we let $p(x_t | \mathbf{x}_{1:t-1})$ be a conditional mixture of Gaussians, we get a model known as **RNADE**
27 (“Real-valued Neural Autoregressive Density Estimator”) of [UML13]. More precisely, this has the
28 form

$$30 \quad p(x_t | \mathbf{x}_{1:t-1}) = \sum_{k=1}^K \pi_{t,k} \mathcal{N}(x_t | \mu_{t,k}, \sigma_{t,k}^2) \quad (23.3)$$

31

32 where the parameters are generated by a network, $(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t, \boldsymbol{\pi}_t) = f_t(\mathbf{x}_{1:t-1}; \boldsymbol{\theta}_t)$.

33 Rather than using separate neural networks, f_1, \dots, f_T , it is more efficient to create a single
34 network with T inputs and T outputs. This can be done using masking, resulting in a model called
35 the **MADE** (“Masked Autoencoder for Density Estimation”) model [Ger+15].

36 One disadvantage of NADE-type models is that they assume the variables have a natural linear
37 ordering. This makes sense for temporal or sequential data, but not for more general data types,
38 such as images or graphs. An orderless extension to NADE was proposed in [UML14; Uri+16].

41

42 23.3 Causal CNNs

43

44 One approach to representing the distribution $p(x_t | \mathbf{x}_{1:t-1})$ is to try to identify patterns in the past
45 history that might be predictive of the value of x_t . If we assume these patterns can occur in any
46 location, it makes sense to use a **convolutional neural network** to detect them. However, we need

47

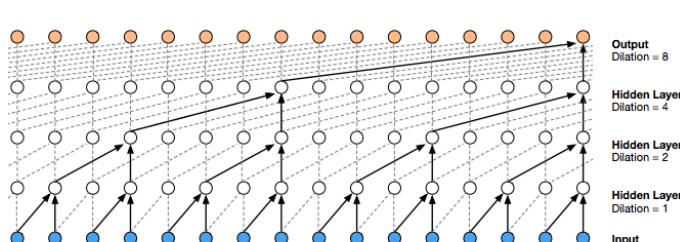


Figure 23.2: Illustration of the wavenet model using dilated (atrous) convolutions, with dilation factors of 1, 2, 4 and 8. From Figure 3 of [oor+16]. Used with kind permission of Aaron van den Oord.

to make sure we only apply the convolutional mask to past inputs, not future ones. This can be done using **masked convolution**, also called **causal convolution**. We discuss this in more detail below.

23.3.1 1d causal CNN (Convolutional Markov models)

Consider the following **convolutional Markov model** for 1d discrete sequences:

$$p(\mathbf{x}_{1:T}) = \prod_{t=1}^T p(x_t | \mathbf{x}_{1:t-1}; \boldsymbol{\theta}) = \prod_{t=1}^T \text{Cat}(x_t | \boldsymbol{\sigma}(\varphi(\sum_{\tau=1}^{t-k} \mathbf{w}^\top \mathbf{x}_{\tau:\tau+k}))) \quad (23.4)$$

where \mathbf{w} is the convolutional filter of size k , and we have assumed a single nonlinearity φ and categorical output, for notational simplicity. This is like regular 1d convolution except we “mask out” future inputs, so that x_t only depends on the past values. We can of course use deeper models, and we can condition on input features \mathbf{c} .

In order to capture long-range dependencies, we can use **dilated convolution** (see [Mur22, Sec 14.4.1]). This model has been successfully used to create a state of the art **text to speech** (TTS) synthesis system known as **wavenet** [oor+16]. See Figure 23.2 for an illustration.

The wavenet model is a conditional model, $p(\mathbf{x}|\mathbf{c})$, where \mathbf{c} is a set of linguistic features derived from an input sequence of words, and \mathbf{x} is raw audio. The **tacotron** system [Wan+17c] is a fully end-to-end approach, where the input is words rather than linguistic features.

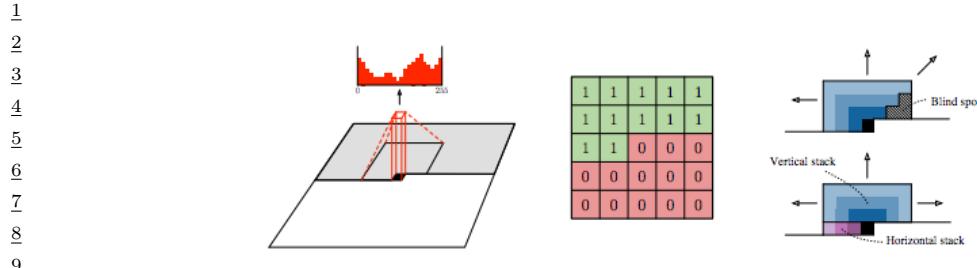
Although wavenet produces high quality speech, it is too slow for use in production systems. However, it can be “distilled” into a parallel generative model [Oor+18], as we discuss in Section 24.2.4.3.

23.3.2 2d causal CNN (PixelCNN)

We can extend causal convolutions to 2d, to get an autoregressive model of the form

$$p(\mathbf{x}|\boldsymbol{\theta}) = \prod_{r=1}^R \prod_{c=1}^C p(x_{r,c} | f_{\boldsymbol{\theta}}(\mathbf{x}_{1:r-1,1:C}, \mathbf{x}_{r,1:c-1})) \quad (23.5)$$

where R is the number of rows, C is the number of columns, and we condition on all previously generated pixels in a **raster scan** order, as illustrated in Figure 23.3. This is called the **pixelCNN**



1 PROMPT:
 2
 3 In a shocking finding, scientist discovered a herd of unicorns living in a remote,
 4 previously unexplored valley, in the Andes Mountains. Even more surprising to the
 5 researchers was the fact that the unicorns spoke perfect English.
 6
 7 RESPONSE:
 8 The scientist named the population, after their distinctive horn, Ovid's Unicorn.
 9 These four-horned, silver-white unicorns were previously unknown to science.
 10 Now, after almost two centuries, the mystery of what sparked this odd phenomenon
 11 is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz,
 12 and several companions, were exploring the Andes Mountains when they found a small valley,
 13 with no other animals or humans....
 14
 15 *Figure 23.4: Sample text generated by GPT-2 in response to an input prompt. From <https://openai.com/blog/better-language-models/>.*
 16
 17
 18 Transformers are the basis of many popular (conditional) generative models for sequences. We
 19 give some examples below.
 20

23.4.1 Text generation (GPT)

23 In [Rad+18], OpenAI proposed a model called **GPT**, which is short for “Generative Pre-training
 24 Transformer”. This is decoder-only transformer model that uses causal (masked) attention. In
 25 [Rad+19], they propose **GPT-2**, which is a larger version of GPT (1.5 billion parameters, or
 26 6.5GB, for the XL version), trained on a large web corpus (8 million pages, or 40GB). They
 27 also simplify the training objective, and just train it using maximum likelihood. The fluency of
 28 text generated by GPT-2 is quite remarkable; see Figure 23.4 for an example. See also <https://demo.allennlp.org/next-token-lm>, which lets you interact with the (medium sized) model, and
 29 generates the K most likely sequences (computed using beam search) given some input context.
 30

31 More recently, OpenAI released **GPT-3** [Bro+20b], which is an even larger version of GPT-2
 32 (175 billion parameters), trained on even more data (300 billion words), but based on the same
 33 principles. (Training was estimated to take 355 GPU years and cost \$4.6M.) Due to the large size
 34 of the data and model, GPT-3 shows even more remarkable abilities to generate novel text. In
 35 particular, the output can be (partially) controlled by just changing the conditioning prompt. This
 36 enables the model to perform tasks that it has never been trained on, just by giving it some examples
 37 in the prompt. This is called “**in-context learning**”, and is an example of (gradient-free) “few-shot
 38 learning” (see Section 20.5.2). See Figure 23.5 for an example, and <https://gpt3demo.com/apps/openai-gpt-3-playground> for an interactive demo.
 39

23.4.2 Music generation

43 It is possible to modify transformer decoders so that they generate music instead of natural language,
 44 as shown by the **music transformer** paper [Hua+18a]. The key “trick” is to note that the **midi**
 45 **format** for music can be represented as a sequence of parameterized tokens, as shown in Figure 23.6.
 46 To cope with the long sequence length, a relative attention mechanism was devised. See Figure 23.7
 47

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.

To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:
One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduckles.

A "yalubalu" is a type of vegetable that looks like a big pumpkin. An example of a sentence that uses the word yalubalu is:
I was on a trip to Africa and I tried this yalubalu vegetable that was grown in a garden there. It was delicious.

A "Burringo" is a car with very fast acceleration. An example of a sentence that uses the word Burringo is:
In our garage we have a Burringo that my father drives to work every day.

A "Gigamuru" is a type of Japanese musical instrument. An example of a sentence that uses the word Gigamuru is:
I have a Gigamuru that my uncle gave me as a gift. I love to play it at home.

To "screeg" something is to swing a sword at it. An example of a sentence that uses the word screeg is:
We screeghed at each other for several minutes and then we went outside and ate ice cream.

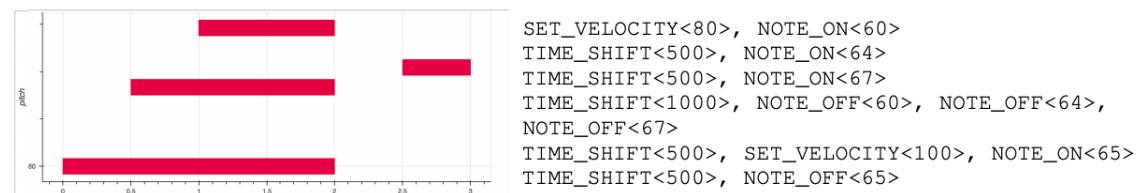


Figure 23.6: A snippet of a piano performance visualized as a pianoroll (left) and encoded as performance events (right, serialized from left to right and then down the rows). There are 128 discrete values for note on/off, 32 values for velocity, and 100 for time shift, so the input is a sequence of one-hot vectors of length 388. From Figure 7 of [Hua+18a]. Used with kind permission of Anna Huang.

for a visualization. To best appreciate the quality of the generated output, please see the interactive demo at <https://magenta.tensorflow.org/music-transformer>.

23.4.3 Text-to-image generation (DALL-E)

The **DALL-E** model¹ from OpenAI [Ram+21] can generate images of remarkable quality and diversity given text prompts, as shown in Figure 23.8. The methodology is conceptually quite

¹ 1. The name is derived from the artist Salvador Dalí and Pixar's movie "WALL-E"