

13 Sequential Monte Carlo

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 29), 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. For another good introduction to SMC, see [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:

$$\pi_T(\mathbf{z}_{1:T}) \propto p(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{t=1}^T p(\mathbf{z}_t|\mathbf{z}_{t-1})p(\mathbf{y}_t|\mathbf{z}_t) \quad (13.2)$$

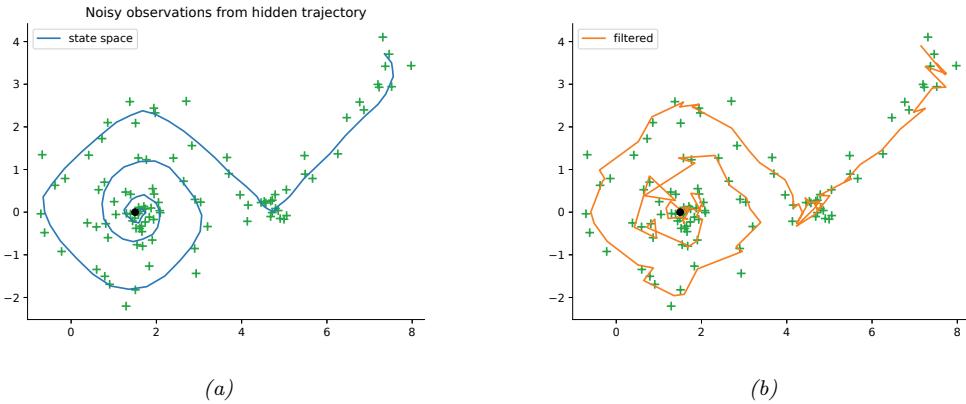


Figure 13.1: Illustration of particle filtering (using the dynamical prior as the proposal) applied to a 2d nonlinear dynamical system. (a) True underlying state and observed data. (b) PF estimate of the posterior mean. Generated by `bootstrap_filter_spiral.ipynb`.

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 latent variables. However, we often only care about the most recent marginal of this distribution, in which case we just need to compute $\tilde{\gamma}_t(\mathbf{z}_t)$, which avoids having to store the full history.

For example, consider the following 2d nonlinear tracking problem (the same one as in Section 8.3.2.3):

$$\begin{aligned} p(\mathbf{z}_t|\mathbf{z}_{t-1}) &= \mathcal{N}(\mathbf{z}_t|f(\mathbf{z}_{t-1}), q\mathbf{I}) \\ p(\mathbf{y}_t|\mathbf{z}_t) &= \mathcal{N}(\mathbf{y}_t|\mathbf{z}_t, r\mathbf{I}) \\ \mathbf{f}(\mathbf{z}) &= (z_1 + \Delta \sin(z_2), z_2 + \Delta \cos(z_1)) \end{aligned} \quad (13.4)$$

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 Supplementary Section 13.1.3 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, $\mathbf{z}_{1:t-1}$, and optionally the past observations, $\mathbf{y}_{1:t-1}$. In this case, the unnormalized target

13.2. Particle filtering

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 (this is example 1.2.1 from [NLS19]):

$$\begin{aligned} p(z_t|\mathbf{z}_{1:t-1}) &= \mathcal{N}(z_t|\phi z_{t-1}, q) \\ p(y_t|\mathbf{z}_{1:t}) &= \mathcal{N}(y_t| \sum_{s=1}^t \beta^{t-s} z_s, r) \end{aligned} \quad (13.6)$$

If we set $\beta = 0$, we get $p(y_t|\mathbf{z}_{1:t}) = \mathcal{N}(y_t|z_t, r)$ (where we define $0^0 = 1$), so the model becomes 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 treat the model as an SSM and apply particle filtering. 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 Particle filtering

In this section, we cover the basics of SMC for state space models, culminating in a method known as the **particle filter**.

13.2.1 Importance sampling

We start by reviewing the self-normalized importance sampling method (**SNIS**), which is the foundation of the particle filter. (See also Section 11.5.)

Suppose we are interested in estimating the expectation of some function φ_t with respect to a target distribution π_t , which we denote by

$$\pi_t(\varphi) \triangleq \mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] = \int \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{Z_t} \varphi_t(\mathbf{z}_{1:t}) d\mathbf{z}_{1:t} \quad (13.7)$$

where $Z_t = \int \tilde{\gamma}_t(\mathbf{z}_{1:t}) d\mathbf{z}_{1:t}$. Suppose we use SNIS with proposal $q_t(\mathbf{z}_{1:t})$. We then get the following approximation:

$$\pi_t(\varphi) \approx \frac{1}{\hat{Z}_t} \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t(\mathbf{z}_{1:t}^i) \varphi_t(\mathbf{z}_{1:t}^i) \quad (13.8)$$

where $\mathbf{z}_{1:t}^i \stackrel{\text{iid}}{\sim} q_t$ are independent samples from the proposal, \tilde{w}_t^i are the **unnormalized weights** defined by

$$\tilde{w}_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{q_t(\mathbf{z}_{1:t}^i)} \quad (13.9)$$

and \hat{Z}_t is the approximate normalization constant defined by

$$\hat{Z}_t \triangleq \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i \quad (13.10)$$

To simplify notation, let us define the **normalized weights** by

$$W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j} \quad (13.11)$$

Then we can write

$$\mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] \approx \sum_{i=1}^{N_s} W_t^i \varphi_t(\mathbf{z}_{1:t}^i) \quad (13.12)$$

Alternatively, instead of computing the expectation of a specific target function, we can just approximate the target distribution itself, using a sum of weighted samples:

$$\pi_t(\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i) \triangleq \hat{\pi}_t(\mathbf{z}_{1:t}) \quad (13.13)$$

The problem with importance sampling when applied in the context of sequential models is that the dimensionality of the state space is very large, and increases with t . This makes it very hard to define a good proposal that covers the high probability regions, resulting in most samples getting negligible weight. In the sections below, we discuss solutions to this problem.

13.2.2 Sequential importance sampling

In this section, we discuss **sequential importance sampling** or **SIS**, in which the proposal has the following autoregressive structure:

$$q_t(\mathbf{z}_{1:t}) = q_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1}) \quad (13.14)$$

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 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 (sequence of states). The unnormalized weights can be computed recursively as follows:

$$\tilde{w}_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{q_t(\mathbf{z}_{1:t})} = \frac{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})q_{t-1}(\mathbf{z}_{1:t-1})} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{(13.15)}$$

$$= \frac{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{q_{t-1}(\mathbf{z}_{1:t-1})} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.16)$$

$$= \tilde{w}_{t-1}(\mathbf{z}_{1:t-1}) \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.17)$$

The ratio factors are sometimes called the **incremental importance weights**:

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.18)$$

See Algorithm 13.1 for pseudocode for the resulting SIS algorithm. (In practice we compute the weights in log-space, and convert back using the log-sum-exp trick.)

Note that, in the special case of state space models, the weight computation can be further simplified. In particular, suppose we have

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) = p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1}) \quad (13.19)$$

$$= p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) \quad (13.20)$$

Then the incremental weight is given by

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} = \frac{p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})}{q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.21)$$

Unfortunately SIS suffers from a problem known as **weight degeneracy** or **particle impoverishment**, in which most of the weights become very small (near 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 amongst the set of candidates, so when we normalize the weights, the best one will get weight 1 and the rest will get weight 0. But this is a waste of most of the particles. We discuss a solution to this in Section 13.2.3.

Algorithm 13.1: Sequential importance sampling (SIS)

```

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   for  $i = 1 : N_s$  do
4     Sample  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
5     Compute incremental weight  $\alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
6     Compute unnormalized weight  $\tilde{w}_t^i = \tilde{w}_{t-1}^i \alpha_t^i$ 
7     Compute normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
8   Compute MC posterior  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

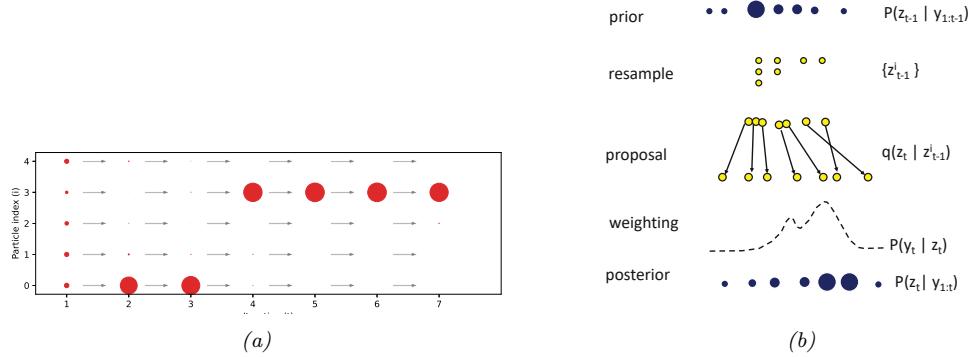


Figure 13.2: (a) Illustration of weight degeneracy for SIS applied to the model in Equation (13.6), with parameters $(\phi, q, \beta, r) = (0.9, 10.0, 0.5, 1.0)$. We use $T = 6$ steps and $N_s = 5$ samples. We see that as t increases, almost all the probability mass concentrates on particle 3. Generated by [sis_vs_smcl.ipynb](#). Adapted from Figure 2 of [NLS19]. (b) Illustration of the bootstrap particle filtering algorithm.

13.2.3 Sequential importance sampling with resampling

In this section, we describe **sequential importance sampling with resampling (SISR)**. The basic idea is this: instead of “growing” all of the old particle sequences by one step, we first select the N_s “fittest” particles, by sampling from the old posterior, and then we let these survivors grow by one step.

In more detail, at step t , we sample from

$$q_t^{\text{SISR}}(\mathbf{z}_{1:t}) = \hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.22)$$

where $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1})$ is the previous weighted posterior approximation. By contrast, in SIS, we sample from

$$q_t^{\text{SIS}}(\mathbf{z}_{1:t}) = q_{t-1}^{\text{SIS}}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.23)$$

Algorithm 13.2: Sequential importance sampling with resampling (SISR)

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 Compute ancestors $\mathbf{a}_{t-1}^{1:N_s} = \text{resample}(\tilde{w}_{t-1}^{1:N_s})$

4 Select $\mathbf{z}_{t-1}^{1:N_s} = \text{permute}(\mathbf{a}_{t-1}^{1:N_s}, \mathbf{z}_{t-1}^{1:N_s})$

5 Reset unnormalized weights $\tilde{w}_{t-1}^{1:N_s} = 1/N_s$

6 **for** $i = 1 : N_s$ **do**

7 Sample $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$

8 Compute unnormalized weight $\tilde{w}_t^i = \alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$

9 Compute normalized weights $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$ for $i = 1 : N_s$

10 Compute MC posterior $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$

We can sample from Equation (13.22) in two steps. First we **resample** N_s samples from $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1})$ to get a *uniformly weighted* set of new samples $\mathbf{z}_{1:t-1}^i$. (See Section 13.2.4 for details on how to do this.) Then we extend each sample using $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$, and concatenate \mathbf{z}_t^i to $\mathbf{z}_{1:t-1}^i$,

After making a proposal, we compute the unnormalized weights. We use the standard SNIS method, except we “pretend” that the proposal is given by $\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)$ 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.) We then compute the unnormalized weights, which are the same as the incremental weights, since the resampling step sets $\tilde{w}_{t-1}^i = 1$. We then normalize these weights and compute the new approximation to the target posterior $\hat{\pi}_t(\mathbf{z}_{1:t})$. See Algorithm 13.2 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:

$$q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.24)$$

In this case, the corresponding incremental weight in Equation (13.21) simplifies to

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{p(\mathbf{y}_t | \mathbf{z}_{1:t}) p(\mathbf{z}_t | \mathbf{z}_{1:t-1})}{q(\mathbf{z}_t | \mathbf{z}_{1:t-1})} = \frac{p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{z}_{t-1})}{p(\mathbf{z}_t | \mathbf{z}_{t-1})} = p(\mathbf{y}_t | \mathbf{z}_{1:t}) \quad (13.25)$$

This special case is called the **bootstrap filter** [Gor93] or the **survival of the fittest** algorithm [KKR95]. (In the computer vision literature, this 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.

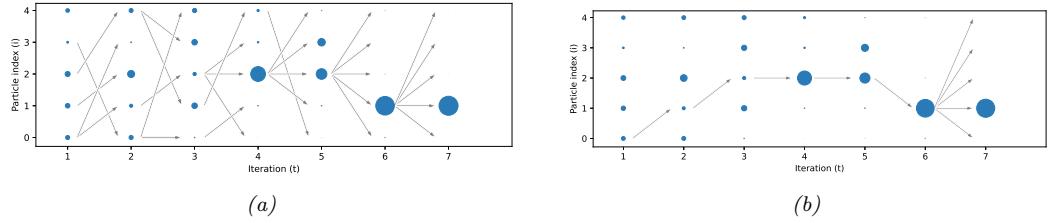


Figure 13.3: (a) Illustration of diversity of samples in SMC applied to the model in Equation (13.6). (b) Illustration of the path degeneracy problem. Generated by `sis_vs_smc.ipynb`. Adapted from Figure 3 of [NLS19].

The bootstrap filter is 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 equations (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 13.3.

13.2.3.2 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” (have non-negligible weight) over 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.

13.2.3.3 Estimating the normalizing constant

We can use particle filtering to approximate the normalization constant $Z_T = p(\mathbf{y}_{1:T}) = \prod_{t=1}^T p(\mathbf{y}_t | \mathbf{y}_{1:t-1})$ as follows:

$$\hat{Z}_T = \prod_{t=1}^T \hat{Z}_t \tag{13.26}$$

where, from Equation (13.10), we have

$$\hat{Z}_t = \frac{1}{N_s} \sum_{i=1}^{N_s} \hat{w}_t^i = \hat{Z}_{t-1} \left(\widehat{Z_t / Z_{t-1}} \right) \tag{13.27}$$

13.2. Particle filtering

where

$$\widehat{Z_t/Z_{t-1}} = \frac{\sum_{i=1}^{N_s} \tilde{w}_t^i}{\sum_{i=1}^{N_s} \tilde{w}_{t-1}^i} \quad (13.28)$$

This estimate of the marginal likelihood is very useful for tasks such as parameter estimation.

13.2.4 Resampling methods

Importance sampling gives a weighted set of particles, $\{(W_t^i, \mathbf{z}_t^i) : i = 1 : N\}$, which we can use to approximate posterior expectations using

$$\mathbb{E}[f(\mathbf{z}_t) | \mathbf{y}_{1:t}] \approx \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.29)$$

Suppose we sample a single index $A \in \{1, \dots, N\}$ with probabilities (W_t^1, \dots, W_t^N) . Then the expected value evaluated at this index is

$$\mathbb{E}[f(\mathbf{z}_t^A) | \mathbf{y}_{1:t}] = \sum_{i=1}^N p(A=i) f(\mathbf{z}_t^i) = \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.30)$$

If we sample N indices independently and compute their average, we get

$$\mathbb{E}[f(\mathbf{z}_t) | \mathbf{y}_{1:t}, A_{1:N}] \approx \frac{1}{N} \sum_{i=1}^N f(\mathbf{z}_t^{A_i}) \quad (13.31)$$

which is a standard unweighted Monte Carlo estimate, with weights $W_t^i = 1/N$. Averaging over the indices gives

$$\mathbb{E}_{A_{1:N}} \left[\frac{1}{N} \sum_{i=1}^N f(\mathbf{z}_t^{A_i}) \right] = \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.32)$$

Thus using the output from the resampling procedure — which drops particles with low weight, and duplicates particles with high weight — will give the same result in expectation as the original weighted estimate. However, to reduce the variance of the method, we need to pick the resampling method carefully, as we discuss below.

13.2.4.1 Inverse cdf

Most of the common resampling methods work as follows. First we form the cumulative distribution from the weights $W^{1:N}$, as illustrated by the staircase in Figure 13.4. (We drop the t index for brevity.) Then, given a set of N uniform random variables, $U^i \sim \text{Unif}(0, 1)$, we check to see which bin (interval) U^i lands in; if it falls in bin a , we return index a , i.e., sample i gets mapped to index a if

$$\sum_{j=1}^{a-1} W^j \leq U^i < \sum_{j=1}^a W^j \quad (13.33)$$

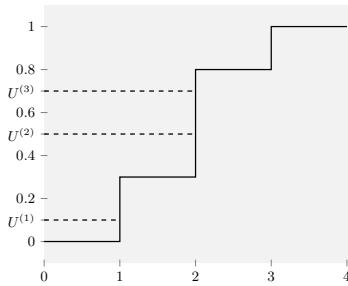


Figure 13.4: Illustration of how to sample from the empirical cdf $P(x) = \sum_{n=1}^N W^n \mathbb{I}(x \geq n)$ shown in black. The height of step n is W_n . If U^m picks step n , then we set the ancestor of m to be n , i.e., $A^m = n$. In this example, $A^{1:3} = (1, 2, 2)$. Adapted from Figure 9.3 of [CP20b].

It would seem that each index would take $O(N)$ time to compute, for a total time of $O(N^2)$, but if the U^i are ordered from smallest to largest, we can implement it in $O(N)$ time. We denote this function $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$. See Listing 13.1 for some JAX code.¹

Listing 13.1: Sampling from an ordered inverse CDF

```
def icdf(weights, u):
    n = weights.shape[0]
    cumsum = jnp.cumsum(weights)
    idx = jnp.searchsorted(cumsum, u)
    return jnp.clip(idx, 0, n - 1)
```

13.2.4.2 Multinomial resampling

In **multinomial resampling**, we set $\mathbf{U}^{1:N}$ to be an ordered set of N samples from the uniform distribution. We then compute the ancestor indices using $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.

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.

13.2.4.3 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

$$U^i \sim \text{Unif}((i - 1)/N_s, i/N_s) \tag{13.34}$$

and compute $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.²

1. Modified from <https://github.com/blackjax-devs/blackjax/blob/main/blackjax/smcr/resampling.py>.

2. To compute the $\mathbf{U}^{1:N}$, we can use $v = jr.uniform(rngkey, (n,))$ and $u = (jnp.arange(n) + v) / n$.

13.2.4.4 Systematic resampling

We can further reduce the variance by forcing all the samples to be deterministically generated from a shared random source, $u \sim \text{Unif}(0, 1)$, by computing

$$U^i = \frac{i - 1}{N_s} + \frac{u}{N_s} \quad (13.35)$$

We then compute $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.³

13.2.4.5 Comparison

It can be proved that all of the above methods are unbiased. Empirically it seems that systematic resampling is lower variance than other methods [HSG06], although stratified resampling, and the more complex method of [GCW19], have better theoretical properties. Multinomial resampling is not recommended, since it has provably higher variance than the other methods.

13.2.5 Adaptive resampling

The resampling step can result in loss of diversity, since each ancestor may generate multiple children, and some may generate no children, since the ancestor indices A_t^n are sampled independently; this is the path degeneracy problem mentioned above. On the other hand, if we never resample, we end up with SIS, which suffers from weight degeneracy (particles with negligible weight). A compromise is to use **adaptive resampling**, in which we resample whenever the **effective sample size** or ESS drops below some minimum, such as $N/2$. A common way to define the ESS is as follows:⁴

$$\text{ESS}(W^{1:N}) = \frac{1}{\sum_{n=1}^N (W^n)^2} \quad (13.36)$$

Alternatively we can compute the ESS using the unnormalized weights:

$$\text{ESS}(\tilde{w}^{1:N}) = \frac{\left(\sum_{n=1}^N \tilde{w}^n\right)^2}{\sum_{n=1}^N (\tilde{w}^n)^2} \quad (13.37)$$

Note that if we have k weights with $\tilde{w}^n = 1$ and $N - k$ weights with $\tilde{w}^n = 0$, then the ESS is k ; thus ESS is between 1 and N .

The pseudocode for SISR with adaptive resampling is given in Algorithm 13.3. (We use the notation of [Law+22, App. B], in which we first sample new extensions of the sequences, and then optionally resample the sequences at the end of each step.)

13.3 Proposal distributions

The efficiency of PF is crucially dependent on the quality of the proposal distribution. We discuss some options below.

3. To compute the $\mathbf{U}^{1:N}$, we can use $v = \text{jnp.uniform(rngkey, ())}$ and $u = (\text{jnp.arange}(n) + v) / n$.

4. Note that the ESS used in SMC is different than the ESS used in MCMC (Section 12.6.3); the latter takes into account auto-correlation of the MCMC samples.

Algorithm 13.3: SISR with adaptive resampling (generic SMC)

```

1 Initialization:  $\tilde{w}_0^{1:N_s} = 1$ ,  $\hat{Z}_0 = 1$ 
2 for  $t = 1 : T$  do
3   for  $i = 1 : N_s$  do
4     Sample particle  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
5     Compute incremental weight  $\alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i)q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
6     Compute unnormalized weight  $\tilde{w}_t^i = \tilde{w}_{t-1}^i \alpha_t^i$ 
7   Estimate normalization constant:  $\widehat{Z_t / Z_{t-1}} = \frac{\sum_{i=1}^{N_s} \tilde{w}_t^i}{\sum_{i=1}^{N_s} \tilde{w}_{t-1}^i}$ ,  $\hat{Z}_t = \hat{Z}_{t-1} (\widehat{Z_t / Z_{t-1}})$ 
8   if  $ESS(\tilde{w}_{t-1}^{1:N}) < ESS_{min}$  then
9     Compute ancestors  $\mathbf{a}_t^{1:N_s} = \text{resample}(\tilde{w}_t^{1:N_s})$ 
10    Select  $\mathbf{z}_t^{1:N_s} = \text{permute}(\mathbf{a}_t^{1:N_s}, \mathbf{z}_t^{1:N_s})$ 
11    Reset unnormalized weights  $\tilde{w}_t^{1:N_s} = 1/N_s$ 
12   Compute normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
13   Compute MC posterior  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

13.3.1 Locally optimal proposal

We define the (one-step) **locally optimal proposal distribution** $q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1})$ to be the one that minimizes

$$D_{\text{KL}}(\pi_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_{1:t})) \quad (13.38)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [\log \{\pi_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})\} - \log \pi_t(\mathbf{z}_{1:t})] \quad (13.39)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [\log q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) - \log \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})] + \text{const} \quad (13.40)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [D_{\text{KL}}(q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}))] + \text{const} \quad (13.41)$$

The KL is minimized by choosing

$$q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_t(\mathbf{z}_{1:t-1})} \quad (13.42)$$

where $\tilde{\gamma}_t(\mathbf{z}_{1:t-1}) = \int \tilde{\gamma}_t(\mathbf{z}_{1:t}) d\mathbf{z}_t$ is the probability of the past sequence under the current target distribution.

Note that the subscript t specifies the t 'th distribution, so in the context of SSMs, we have $\pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{1:t-1}, \mathbf{y}_{1:t})$. Thus we see that when proposing \mathbf{z}_t , we should condition on all the data, including the most recent observation, \mathbf{y}_t ; this is called a **guided particle filter**, and will be better than the bootstrap filter, which proposes from the prior.

In general, it is intractable to compute the locally optimal proposal, so we consider various approximations below.

13.3.2 Proposals based on the extended and unscented Kalman filter

One way to approximate the locally optimal proposal distribution is based on the extended Kalman filter (Section 8.3.2) or the unscented Kalman filter (Section 13.3.2), which gives rise to the **extended particle filter** [DGA00] and **unscented particle filter** [Mer+00] respectively. To explain these methods, we follow the presentation of [NLS19, p36]. As usual, we assume the dynamical system can be written as $\mathbf{z}_t = \mathbf{f}(\mathbf{z}_{t-1}) + \mathbf{q}_t$ and $\mathbf{y}_t = \mathbf{h}(\mathbf{z}_t) + \mathbf{r}_t$, where \mathbf{q}_t is the system noise and \mathbf{r}_t is the observation noise. The EKF and UKF approximations assume that the joint distribution over neighboring time steps, given the i 'th history, is Gaussian:

$$p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{z}_{1:t-1}^i) \approx \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_t \\ \mathbf{y}_t \end{pmatrix} | \hat{\boldsymbol{\mu}}^i, \hat{\boldsymbol{\Sigma}}^i\right) \quad (13.43)$$

where

$$\hat{\boldsymbol{\mu}}^i = \begin{pmatrix} \hat{\boldsymbol{\mu}}_z^i \\ \hat{\boldsymbol{\mu}}_y^i \end{pmatrix}, \hat{\boldsymbol{\Sigma}}^i = \begin{pmatrix} \hat{\Sigma}_{zz}^i & \hat{\Sigma}_{zy}^i \\ \hat{\Sigma}_{yz}^i & \hat{\Sigma}_{yy}^i \end{pmatrix} \quad (13.44)$$

(See Section 8.5.1 for details.)

The EKF and UKF compute $\hat{\boldsymbol{\mu}}^i$ and $\hat{\boldsymbol{\Sigma}}^i$ differently. In the EKF, we linearize \mathbf{f} and \mathbf{h} , and assume the noise terms are Gaussian. We then compute $p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{z}_{1:t-1}^i)$ exactly for this linearized model (see Section 8.3.1). In the UKF, we propagate sigma points through \mathbf{f} and \mathbf{h} , and approximate the resulting means and covariances using the unscented transform, which can be more accurate (see Section 8.4). Once we have computed $\hat{\boldsymbol{\mu}}^i$ and $\hat{\boldsymbol{\Sigma}}^i$, we can use standard rules for Gaussian conditioning to compute the approximate proposal as follows:

$$q(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i, \mathbf{y}_t) \approx \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_t^i, \boldsymbol{\Sigma}_t^i) \quad (13.45)$$

$$\boldsymbol{\mu}_t^i = \hat{\boldsymbol{\mu}}_z^i + \hat{\boldsymbol{\Sigma}}_{zy}^i (\hat{\boldsymbol{\Sigma}}_{yy}^i)^{-1} (\mathbf{y}_t - \hat{\boldsymbol{\mu}}_y^i) \quad (13.46)$$

$$\boldsymbol{\Sigma}_t^i = \hat{\boldsymbol{\Sigma}}_{zz}^i - \hat{\boldsymbol{\Sigma}}_{zy}^i (\hat{\boldsymbol{\Sigma}}_{yy}^i)^{-1} \hat{\boldsymbol{\Sigma}}_{yz}^i \quad (13.47)$$

Note that the linearization (or sigma point) approximation needs to be performed for each particle separately.

13.3.3 Proposals based on the Laplace approximation

To handle non-Gaussian likelihoods in an SSM, we can use the Laplace approximation (Section 7.4.3), as suggested in [DGA00]. In particular, consider an SSM with linear-Gaussian latent dynamics and a 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., using Newton-Raphson), and then approximate the likelihood using

$$p(\mathbf{y}_t | \mathbf{z}_t) \approx \mathcal{N}(\mathbf{z}_t | \mathbf{z}_t^*, -\mathbf{H}_t^*) \quad (13.48)$$

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 update step of the Kalman filter, using the same equations as in Section 13.3.2. This combination is called the **Laplace Gaussian filter** [Koy+10]. We give an example in Section 13.3.3.1.

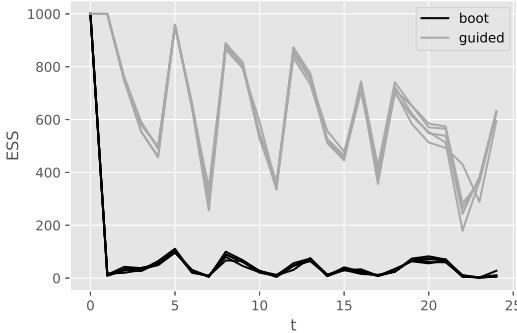


Figure 13.5: Effective sample size at each step for the bootstrap particle filter and a guided particle filter for a Gaussian SSM with Poisson likelihood. Adapted from Figure 10.4 of [CP20b]. Generated by [pf_guided_neural_decoding.ipynb](#).

13.3.3.1 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]:

$$\begin{pmatrix} z_t(i) \\ z_t(i+3) \end{pmatrix} | \mathbf{z}_{t-1} \sim \mathcal{N}_2 \left(\begin{pmatrix} 1 & \Delta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} z_{t-1}(i) \\ z_{t-1}(i+3) \end{pmatrix}, \sigma^2 \mathbf{Q} \right), \quad i = 1 : 3 \quad (13.49)$$

where the covariance of the noise is given by the following, assuming a discretization step of Δ :

$$\mathbf{Q} = \begin{pmatrix} \Delta^3/3 & \Delta^2/2 \\ \Delta^2/2 & \Delta \end{pmatrix} \quad (13.50)$$

We assume the k 'th observation at time t is the number of spikes for neuron k in this sensing interval:

$$p(y_t(k) | \mathbf{z}_t) = \text{Poi}(\lambda_k(\mathbf{z}_t)) \quad (13.51)$$

$$\log \lambda_k(\mathbf{z}_t) = \alpha_k + \beta_k^\top \mathbf{z}_t \quad (13.52)$$

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 its value for furthering basic science, this can be useful for applications such as helping disabled people control their arms using “mind control”.)

To illustrate this, we sample a synthetic dataset from the model, to simulate a “monkey” moving its arm for $T = 25$ time steps; this generates $K = 50$ neuronal counts per time step. We then apply particle filtering to this dataset (using the true model), 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 13.5, we see that the effective sample size of the guided filter is much higher than for the bootstrap filter.

13.3.4 Proposals based on SMC (nested SMC)

It is possible to use SMC as a subroutine to compute a proposal distribution for SMC: at each step t , for each particle i , we run an SMC algorithm where the target distribution is the optimal proposal, $p(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i, \mathbf{y}_{1:t})$. 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, although the inner SMC algorithm can be run in parallel for each outer sample [NLS15; NLS19].

13.4 Rao-Blackwellized particle filtering (RBPF)

In some models, we can partition the hidden variables into two kinds, \mathbf{m}_t and \mathbf{z}_t , such that we can analytically integrate out \mathbf{z}_t provided we know the values of $\mathbf{m}_{1:t}$. This means we only have to sample $\mathbf{m}_{1:t}$, and can represent $p(\mathbf{z}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t})$ 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 8.2.2).

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-Blackwellized particle filtering** or **RBPF** for short. (See Section 11.6.2 for more details on Rao-Blackwellization.) In Section 13.4.1 we give an example of RBPF for inference in a switching linear dynamical systems. In Section 13.4.3 we illustrate RBPF for inference in the SLAM model for a mobile robot.

13.4.1 Mixture of Kalman filters

In this section, we consider the application of RBPF to a switching linear dynamical system (Section 29.9). This model has both continuous and discrete latent variables. This can be used to track a system that switches between discrete modes or operating regimes, represented by the discrete variable m_t .

For notational simplicity, we ignore the control inputs \mathbf{u}_t . Thus the model is given by

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, m_t = k) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_k \mathbf{z}_{t-1}, \mathbf{Q}_k) \quad (13.53)$$

$$p(\mathbf{y}_t | \mathbf{z}_t, m_t = k) = \mathcal{N}(\mathbf{y}_t | \mathbf{H}_k \mathbf{z}_t, \mathbf{R}_k) \quad (13.54)$$

$$p(m_t = k | m_{t-1} = j) = A_{jk} \quad (13.55)$$

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 .

Exact inference is intractable, but if we sample the discrete variables, we can infer the continuous variables conditioned on the discretes exactly, making this a good candidate for RBPF. In particular, if we sample trajectories $\mathbf{m}_{1:t}^n$, 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

$$p(\mathbf{z}_t, \mathbf{m}_t | \mathbf{y}_{1:t}) \approx \sum_{n=1}^N W_t^n \delta(\mathbf{m}_t - \mathbf{m}_t^n) \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_t^n, \boldsymbol{\Sigma}_t^n) \quad (13.56)$$

To derive the filtering algorithm, note that the full posterior at time t can be written as follows:

$$p(\mathbf{m}_{1:t}, \mathbf{z}_{1:t} | \mathbf{y}_{1:t}) = p(\mathbf{z}_{1:t} | \mathbf{m}_{1:t}, \mathbf{y}_{1:t}) p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) \quad (13.57)$$

The second term is given by the following:

$$p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) \propto p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t-1}) \quad (13.58)$$

$$= p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_t | \mathbf{m}_{1:t-1}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.59)$$

$$= p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_t | \mathbf{m}_{t-1}) p(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.60)$$

Note that, unlike the case of standard particle filtering, we cannot write $p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) = p(\mathbf{y}_t | \mathbf{m}_t)$, since \mathbf{m}_t does not d-separate the past observations from \mathbf{y}_t , as is evident from Figure 29.25a.

Suppose we use the following recursive proposal distribution:

$$q(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) = q(\mathbf{m}_t | \mathbf{m}_{1:t-1}, \mathbf{y}_{1:t}) q(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t}) \quad (13.61)$$

Then we get the unnormalized importance weights

$$\tilde{w}_t^n \propto \frac{p(\mathbf{y}_t | m_t^n, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) p(m_t^n | m_{t-1}^n)}{q(m_t^n | \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t})} \tilde{w}_{t-1}^n \quad (13.62)$$

As a special case, suppose we propose from the prior, $q(m_t | \mathbf{m}_{t-1}^n, \mathbf{y}_{1:t}) = p(m_t | m_{t-1}^n)$. If we sample discrete state k , the weight update becomes

$$\tilde{w}_t^n \propto \tilde{w}_{t-1}^n p(\mathbf{y}_t | m_t^n = k, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \tilde{w}_{t-1}^n L_{tk}^n \quad (13.63)$$

where

$$L_{tk}^n = p(\mathbf{y}_t | m_t = k, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \int p(\mathbf{y}_t | m_t = k, \mathbf{z}_t) p(\mathbf{z}_t | m_t = k, \mathbf{y}_{1:t-1}, \mathbf{m}_{1:t-1}^n) d\mathbf{z}_t \quad (13.64)$$

The quantity L_{tk}^n is the predictive density for the new observation \mathbf{y}_t conditioned on $m_t = k$ and the history of previous latents, $\mathbf{m}_{1:t-1}^n$. In the case of SLDS models, this can be computed using the normalization constant of the Kalman filter, Equation (8.35). The resulting algorithm is shown in Algorithm 13.4. The step marked “KFupdate” refers to the Kalman filter update equations in Section 8.2.2, and is applied to each particle separately.

Algorithm 13.4: One step of RBPF for SLDS using prior as proposal

```

1 for n = 1 : N do
2   k ~ p(m_t | m_{t-1}^n)
3   m_t^n := k
4   ( $\mu_t^n, \Sigma_t^n, L_{tk}^n$ ) = KFupdate( $\mu_{t-1}^n, \Sigma_{t-1}^n, \mathbf{y}_t, \theta_k$ )
5    $\tilde{w}_t^n = \tilde{w}_{t-1}^n L_{tk}^n$ 
6 Compute ESS = ESS( $\tilde{w}_t^{1:N_s}$ )
7 if ESS < ESS_min then
8    $\mathbf{a}_t^{1:N} = \text{Resample}(\tilde{w}_t^{1:N})$ 
9   ( $\mathbf{m}_t^{1:N_s}, \mu_t^{1:N_s}, \Sigma_t^{1:N_s}$ ) = permute( $\mathbf{a}_t, \mathbf{m}_t^{1:N_s}, \mu_t^{1:N_s}, \Sigma_t^{1:N_s}$ )
10   $\tilde{w}_t^n = 1/N_s$ 

```

13.4.1.1 Improvements

An improved version of the algorithm can be developed based on the fact that we are sampling a discrete state space. At each step, we propagate each of the N old particles through all K possible transition models. We then compute the weight for all NK new particles, and sample from this to get the final set of N particles. This latter step can be done using the **optimal resampling** method of [FC03], which will stochastically select the particles with the largest weight, while also ensuring the result is an unbiased approximation. In addition, this approach ensures that we do not have duplicate particles, which is wasteful and unnecessary when the state space is discrete.

13.4.2 Example: tracking a maneuvering object

In this section we give an example of RBPF for an SLDS from [DGK01]. Our goal is to track an object that has the following motion model:

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, m_t = k) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}\mathbf{z}_{t-1} + \mathbf{b}_k, \mathbf{Q}) \quad (13.65)$$

where $\mathbf{z}_t = (x_{1t}, \dot{x}_{1t}, x_{2t}, \dot{x}_{2t})$ contains the 2d position and velocity. We define the observation matrix by $\mathbf{H} = \mathbf{I}$ and the observation covariance by $\mathbf{R} = 10 \text{ diag}(2, 1, 2, 1)$. We define the dynamics matrix by

$$\mathbf{F} = \begin{pmatrix} 1 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (13.66)$$

where $\Delta = 0.1$. We set the noise covariance to $\mathbf{Q} = 0.2\mathbf{I}$ and the input bias vectors for each state to $\mathbf{b}_1 = (0, 0, 0, 0)$, $\mathbf{b}_2 = (-1.225, -0.35, 1.225, 0.35)$ and $\mathbf{b}_3 = (1.225, 0.35, -1.225, -0.35)$. Thus the system will turn in different directions depending on the discrete state. The discrete state transition matrix is given by

$$\mathbf{A} = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix} \quad (13.67)$$

Figure 13.6a shows some observations, and the true state of the system, from a sample run, for 100 steps. The colors denote the discrete state, and the location of the symbol denotes the (x, y) location. The small dots represent noisy observations. Figure 13.6b shows the estimate of the state computed using RBPF with the optimal proposal with 1000 particles. In Figure 13.6c, we show the analogous estimate using the bootstrap filter, which does much worse.

In Figure 13.7a and Figure 13.7b, we show the posterior marginals of the (x, y) locations over time. In Figure 13.7c we show the true discrete state, and in Figure 13.7d we show the posterior marginal over discrete states. The overall state classification error rate is 29%, but it seems that occasionally misclassifying isolated time steps does not significantly hurt estimation of the continuous states, as we can see from Figure 13.6b.

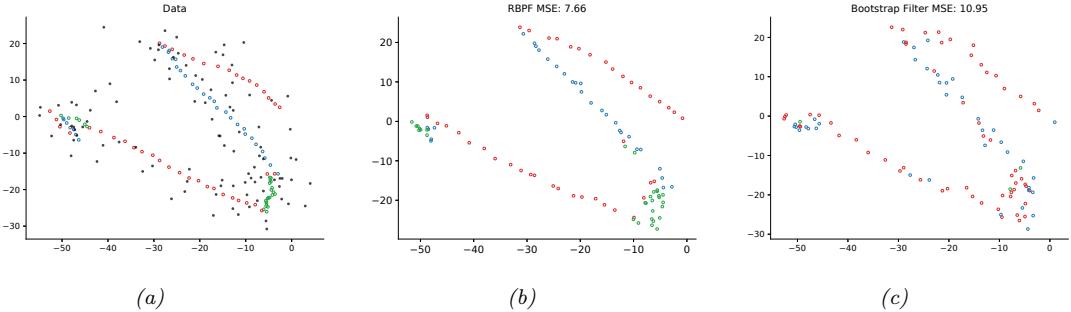


Figure 13.6: Illustration of state estimation for a switching linear model. (a) Black dots are observations, hollow circles are the true location, colors represent the discrete state. (b) Estimate from RBPF. Generated by `rpbpf_maneuver.ipynb`. (c) Estimate from bootstrap filter. Generated by `bootstrap_filter_maneuver.ipynb`.

13.4.3 Example: FastSLAM

Consider a robot moving around an environment, such as a maze or indoor office environment. It needs to learn a map of the environment, and keep track of its location (pose) within that map. This problem is known as **simultaneous localization and mapping**, or **SLAM** for short. SLAM is widely used in mobile robotics (see e.g., [SC86; CN01; TBF06] for details). It is also useful in augmented reality, where the task is to recursively estimate the 3d pose of a handheld camera with respect to a set of 2d visual landmarks (this is known as **visual SLAM**, [TUI17; SMT18; Cza+20; DH22]).

Let us assume we can represent the map as the 2d locations of a set of K landmarks, denote them by $\mathbf{l}^1, \dots, \mathbf{l}^K$ (each is a vector in \mathbb{R}^2). (We can use data association to figure out which landmark generated each observation, as discussed in Section 29.9.3.2.) Let \mathbf{r}_t represent the unknown location of the robot at time t . Let $\mathbf{z}_t = (\mathbf{r}_t, \mathbf{l}_t^{1:K})$ be the combined state space. We can then perform online inference so that the robot can update its estimate of its own location, and the landmark locations.

The state transition model is defined as

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{u}_t) = p(\mathbf{r}_t | \mathbf{r}_{t-1}, \mathbf{l}_{t-1}^{1:K}, \mathbf{u}_t) \prod_{k=1}^K p(\mathbf{l}_t^k | \mathbf{l}_{t-1}^k) \quad (13.68)$$

where $p(\mathbf{r}_t | \mathbf{r}_{t-1}, \mathbf{l}_{t-1}^{1:K}, \mathbf{u}_t)$ specifies how the robot moves given the control signal \mathbf{u}_t and the location of the obstacles $\mathbf{l}_{t-1}^{1:K}$. (Note that in this section, we assume that a human is joysticking the robot through the environment, so $\mathbf{u}_{1:t}$ is given as input, i.e., we do not address the decision-theoretic issue of choosing where to move.)

If the obstacles (landmarks) are static, we can define $p(\mathbf{l}_t^k | \mathbf{l}_{t-1}^k) = \delta(\mathbf{l}_t^k - \mathbf{l}_{t-1}^k)$, which is equivalent to treating the map as an unknown parameter that is shared globally across all time steps. More generally, we can let the landmark locations evolve over time [Mur00].

The observations \mathbf{y}_t measure the distance from \mathbf{r}_t to the set of closest landmarks. Figure 13.8 shows the corresponding graphical model for the case where $K = 2$, and where on the first step it sees landmarks 1 and 2, then just landmark 2, then just landmark 1, etc.

If all the CPDs are linear-Gaussian, then we can use a Kalman filter to maintain our belief state

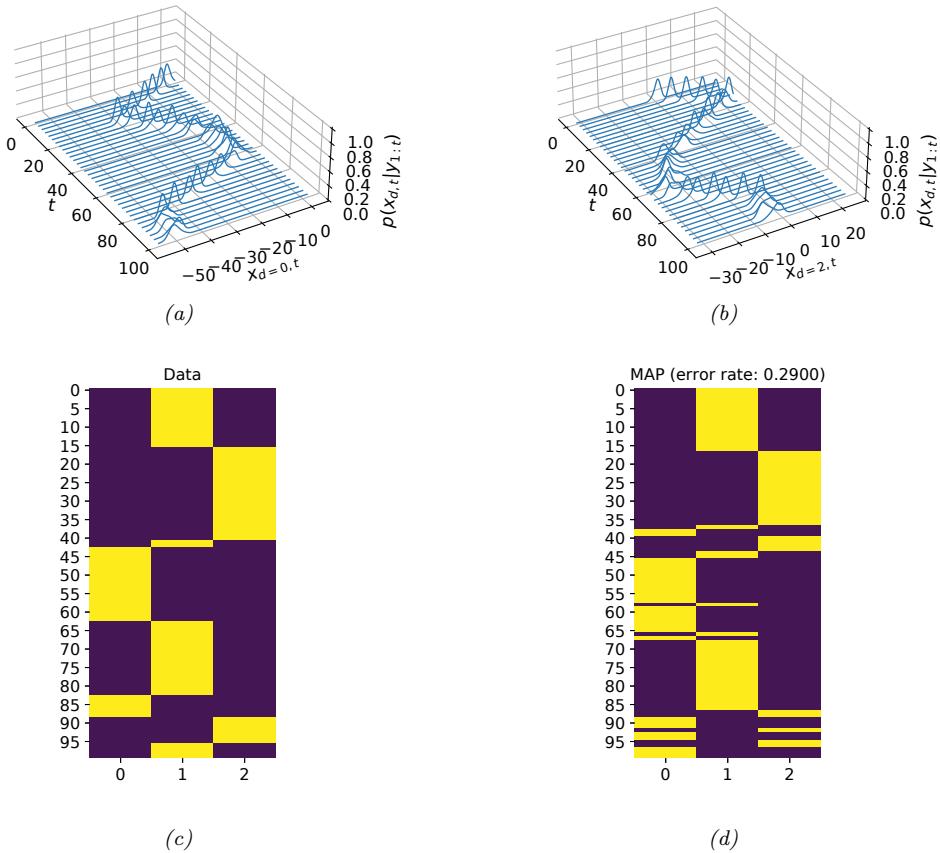


Figure 13.7: Visualizing the posterior from the RBPF algorithm. Top row: Posterior marginals of the location of the object over time, derived from the mixture of Gaussian representation for (a) x location (dimension 0), (b) y location (dimension 2). Bottom row: visualization of the true (c) and predicted (d) discrete states. Generated by [rbpf_maneuver.ipynb](#).

about the location of the robot and the location of the landmarks, $p(\mathbf{z}_t | \mathbf{y}_{1:t}, \mathbf{u}_{1:t})$. In the more general case of a nonlinear model, we can use the EKF (Section 8.3.2) or UKF (Section 8.4.2).

Over time, the uncertainty in the robot's location will increase, due to wheel slippage, etc., but when the robot returns to a familiar location, its uncertainty will decrease again. This is called **closing the loop**, and is illustrated in Figure 13.9(a), where we see the uncertainty ellipses, representing $\text{Cov}[\mathbf{z}_t | \mathbf{y}_{1:t}, \mathbf{u}_{1:t}]$, grow and then shrink.

In addition to visualizing the uncertainty of the robot's location, we can visualize the uncertainty about the map. To do this, consider the posterior precision matrix, $\Lambda_t = \Sigma_t^{-1}$. Zeros in the precision

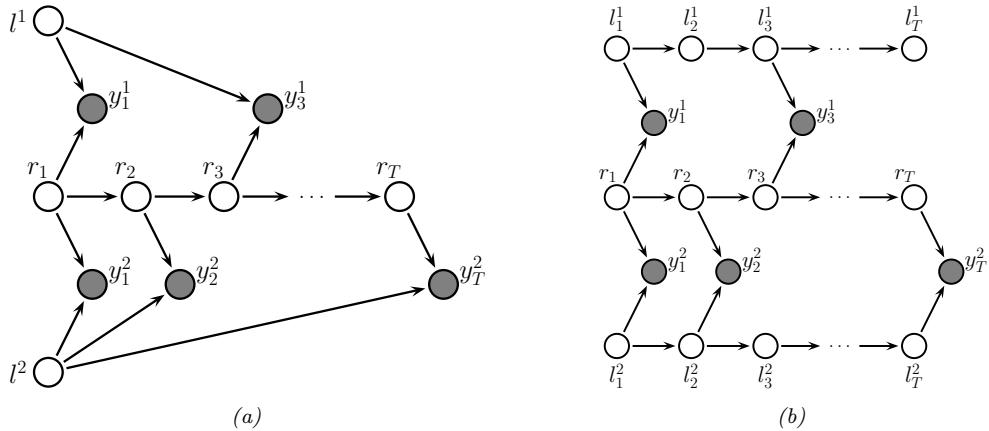


Figure 13.8: Graphical model representing the SLAM problem. \mathbf{l}_t^k is the location of landmark k at time t , \mathbf{r}_t is the location of the robot at time t , and \mathbf{y}_t is the observation vector. In the model on the left, the landmarks are static (so they act like global shared parameters), on the right, their location can change over time. The robot's observations are based on the distance to the nearest landmarks from the current state, denoted $f(\mathbf{r}_t, \mathbf{l}_t^k)$. The number of observations per time step is variable, depending on how many landmarks are within the range of the sensor. Adapted from Figure 15.A.3 of [KF09a].

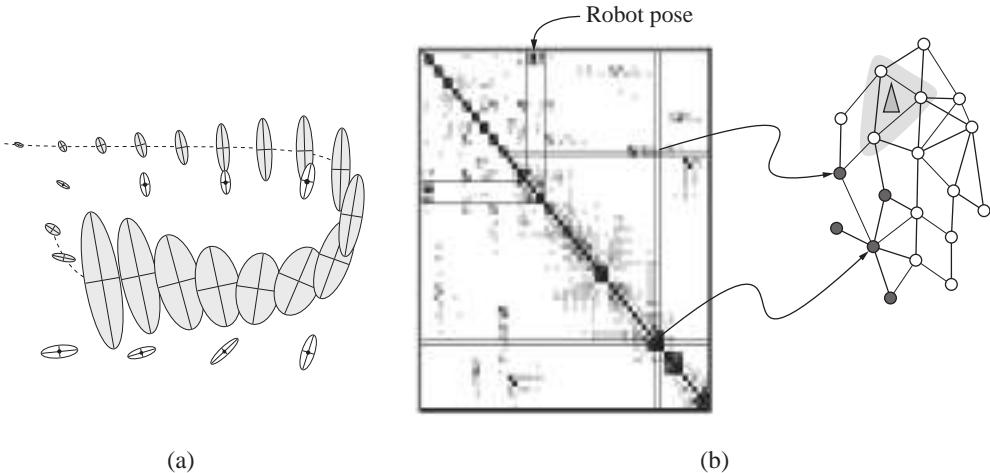


Figure 13.9: Illustration of the SLAM problem. (a) A robot starts at the top left and moves clockwise in a circle back to where it started. We see how the posterior uncertainty about the robot's location increases and then decreases as it returns to a familiar location, closing the loop. If we performed smoothing, this new information would propagate backwards in time to disambiguate the entire trajectory. (b) We show the precision matrix, representing sparse correlations between the landmarks, and between the landmarks and the robot's position (pose). The conditional independencies encoded by the sparse precision matrix can be visualized as a Gaussian graphical model, as shown on the right. From Figure 15.A.3 of [KF09a]. Used with kind permission of Daphne Koller.

matrix correspond to absent edges in the corresponding undirected Gaussian graphical model (GGM, see Section 4.3.5). Initially all the beliefs about landmark locations are uncorrelated (by assumption), so the GGM is a disconnected graph, and Λ_t is diagonal. However, as the robot moves about, it will induce correlation between nearby landmarks. Intuitively this is because the robot is estimating its position based on distance to the landmarks, but the landmarks' locations are being estimated based on the robot's position, so they all become interdependent. This can be seen more clearly from the graphical model in Figure 13.8: it is clear that \mathbf{l}^1 and \mathbf{l}^2 are not d-separated by $\mathbf{y}_{1:t}$, because there is a path between them via the unknown sequence of $\mathbf{r}_{1:t}$ nodes. Consequently, the precision matrix becomes denser over time. As a consequence of the precision matrix becoming denser, each inference step takes $O(K^3)$ time. This prevents the method from being applied to large maps.

One way to speed this up is based on the following observation: conditional on knowing the robot's path, $\mathbf{r}_{1:t}$, the landmark locations are independent, i.e., $p(\mathbf{l}_t | \mathbf{r}_{1:t}, \mathbf{y}_{1:t}) = \prod_{k=1}^K p(\mathbf{l}_t^k | \mathbf{r}_{1:t}, \mathbf{y}_{1:t})$. This can be seen by looking at the DGM in Figure 13.8. We can therefore sample the trajectory using some proposal, and apply (2d) Kalman filtering to each landmark independently. This is an example of RBPF, and reduces the inference cost to $O(NK)$, where N is the number of particles and K is the number of landmarks.

The overall cost of this technique is $O(NK)$ per step. Fortunately, the number of particles N 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**.

13.5 Extensions of the particle filter

There are many extensions to the basic particle filtering algorithm, such as the following:

- We can increase particle diversity by applying one or more steps of MCMC sampling (Section 12.2) at each PF step using $\pi_t(\mathbf{z}_t)$ as the target distribution. This is called the **resample-move** algorithm [DJ11]. It is also possible to use SMC instead of MCMC to diversify the samples [GM17].
- We can extend PF to the case of offline inference; this is called **particle smoothing** (see e.g., [Kla+06]).
- We can extend PF to inference in general graphical models (not just chains) by combining PF with loopy belief propagation (Section 9.4); this is called **non-parametric BP** or **particle BP** (see e.g., [Sud+03; Isa03; Sud+10; Pac+14]).
- We can extend PF to perform inference in static models (e.g., for parameter inference), as we discuss in Section 13.6.

13.6 SMC samplers

In this section, we discuss **SMC samplers** (sequential Monte Carlo samplers), which are a way to apply particle filters to sample from a generic target distribution, $\pi(\mathbf{z}) = \tilde{\gamma}(\mathbf{z})/Z$, rather than

requiring the model to be an SSM. Thus SMC is an alternative to MCMC.

The advantages of SMC samplers over 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 all but the final state, we recover samples from the target distribution, $\pi(\mathbf{z}) = \sum_{\mathbf{z}_{1:T-1}} \bar{\pi}_T(\mathbf{z}_{1:T})$, 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 **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 from our current estimate when we apply particle filtering.
- 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 sequence of variables by working backwards in time from the final target value to the first time step. In particular, we create the following joint distribution:

$$\bar{\pi}_t(\mathbf{z}_{1:t}) = \pi_t(\mathbf{z}_t) \prod_{s=1}^{t-1} L_s(\mathbf{z}_s|\mathbf{z}_{s+1}) \quad (13.69)$$

This satisfies $\sum_{\mathbf{z}_{1:t-1}} \bar{\pi}_t(\mathbf{z}_{1:t}) = \pi_t(\mathbf{z}_t)$, so if we apply particle filtering to this for $t = 1 : T$, then samples from the “end” of such sequences will be from the target distribution π_t .

With the above ingredients, we can compute the incremental weight at step t using

$$\alpha_t = \frac{\bar{\pi}_t(\mathbf{z}_{1:t})}{\bar{\pi}_{t-1}(\mathbf{z}_{1:t-1})M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \propto \frac{\tilde{\gamma}_t(\mathbf{z}_t)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \frac{L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t)}{M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.70)$$

This can be plugged into the generic SMC algorithm, Algorithm 13.3.

We still have to specify the forwards and backwards kernels. We will assume the forwards kernel M_t is an MCMC kernel that leaves π_t invariant. We can then define the backwards kernel to be the **time reversal** of the forwards kernel. More precisely, suppose we define L_{t-1} so it satisfies

$$\pi_t(\mathbf{z}_t)L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t) = \pi_t(\mathbf{z}_{t-1})M_t(\mathbf{z}_t|\mathbf{z}_{t-1}) \quad (13.71)$$

In this case, the incremental weight simplifies as follows:

$$\alpha_t = \frac{Z_t \pi_t(\mathbf{z}_t) L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t)}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.72)$$

$$= \frac{Z_t \pi_t(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.73)$$

$$= \frac{\tilde{\gamma}_t(\mathbf{z}_{t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \quad (13.74)$$

We can use any kind of MCMC kernel for M_t . For example, if the parameters are real valued and unconstrained, we can use a Markov kernel that corresponds to K steps of a random walk Metropolis-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 covariance of the weighted samples from the previous step, $(W_{t-1}^{1:N}, z_{t-1}^{1:N})$, and $\delta = 2.38D^{-3/2}$ (which is the optimal scaling parameter for RWMH). In high dimensional problems, we can use gradient based Markov kernels, such as HMC [BCJ20] and NUTS [Dev+21]. For binary state spaces, we can use the method of [SC13].

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

$$\tilde{\gamma}_t(\mathbf{z}) = \tilde{\gamma}_0(\mathbf{z})^{1-\lambda_t} \tilde{\gamma}(\mathbf{z})^{\lambda_t} \quad (13.75)$$

where $0 = \lambda_0 < \lambda_1 < \dots < \lambda_T = 1$ are **inverse temperature** parameters, and $\tilde{\gamma}_0$ 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 denote the latent variable \mathbf{z} by $\boldsymbol{\theta}$, we define $\tilde{\gamma}_0(\boldsymbol{\theta}) \propto \pi_0(\boldsymbol{\theta})$ as the prior, and $\tilde{\gamma}(\boldsymbol{\theta}) = \pi_0(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$ as the posterior. We can then define the intermediate distributions to be

$$\tilde{\gamma}_t(\boldsymbol{\theta}) = \pi_0(\boldsymbol{\theta})^{1-\lambda_t} \pi_0(\boldsymbol{\theta})^{\lambda_t} p(\mathcal{D}|\boldsymbol{\theta})^{\lambda_t} = \pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.76)$$

where $\mathcal{E}(\boldsymbol{\theta}) = -\log p(\mathcal{D}, \boldsymbol{\theta})$ is the energy (potential) function. The incremental weights are given by

$$\alpha_t(\boldsymbol{\theta}) = \frac{\pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})]}{\pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_{t-1} \mathcal{E}(\boldsymbol{\theta})]} = \exp[-\delta_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.77)$$

where $\lambda_t = \lambda_{t-1} + \delta_t$.

For this method to work well, it is important to choose the λ_t so that the successive distributions are “equidistant”; this is called **adaptive tempering**. 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

$$\delta_t = \underset{\delta \in [0, 1 - \lambda_{t-1}]}{\operatorname{argmin}} (\operatorname{ESSLW}(\{-\delta \mathcal{E}(\boldsymbol{\theta}_t^n)\}) - \operatorname{ESS}_{\min}) \quad (13.78)$$

where $\operatorname{ESSLW}(\{l_n\}) = \operatorname{ESS}(\{e^{l_n}\})$ computes the ESS (Equation (13.37)) 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 13.5 for the overall algorithm.

13.6.2.1 Example: sampling from a 1d bimodal distribution

Consider the simple distribution

$$p(\boldsymbol{\theta}) \propto \mathcal{N}(\boldsymbol{\theta} | \mathbf{0}, \mathbf{I}) \exp(-\mathcal{E}(\boldsymbol{\theta})) \quad (13.79)$$

Algorithm 13.5: 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 (13.78)
10   $\lambda_t = \lambda_{t-1} + \delta_t$ 
11   $\tilde{w}_t^n = \exp[-\delta\mathcal{E}(\theta_t^n)]$ 
12   $W_t^n = \tilde{w}_t^n / (\sum_{m=1}^N \tilde{w}_t^m)$ 

```

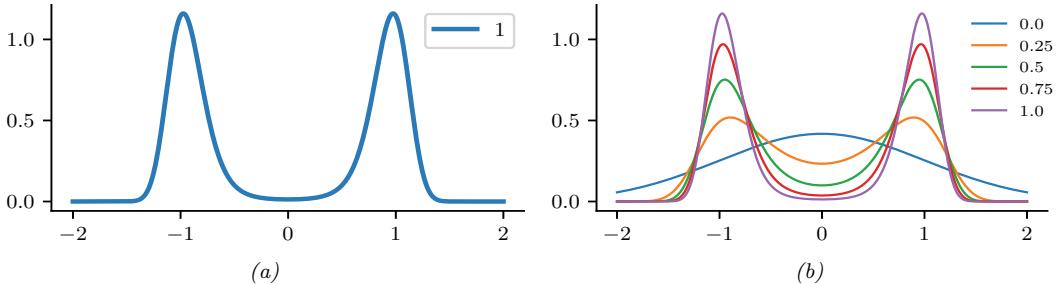


Figure 13.10: (a) Illustration of a bimodal target distribution. (b) Tempered versions of the target at different inverse temperatures, from $\lambda_T = 1$ down to $\lambda_1 = 0$. Generated by [smc_tempered_1d_bimodal.ipynb](#).

where $\mathcal{E}(\boldsymbol{\theta}) = c(\|\boldsymbol{\theta}\|^2 - 1)^2$. We plot this in 1d in Figure 13.10a 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.

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 13.11a and Figure 13.11b 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 13.10b, 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 SMC can track the change until it ends up at the target distribution with $\lambda_T = 1$, as shown in Figure 13.11c.

These SMC results were obtained using the adaptive tempering scheme described above. In Figure 13.11d we see that initially the temperature is small, and then it increases exponentially. The algorithm takes 8 steps until $\lambda_T \geq 1$.

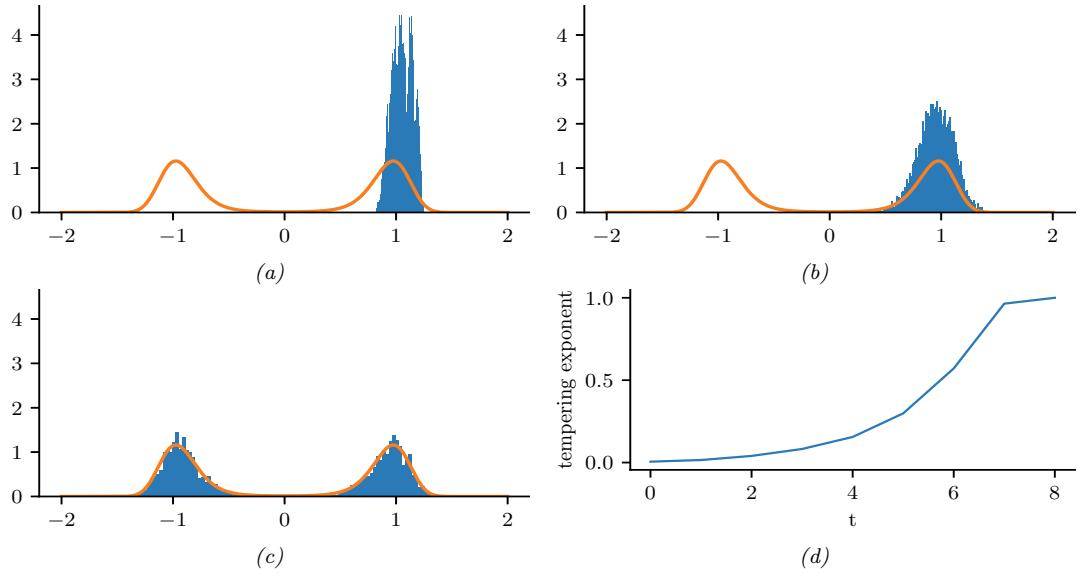


Figure 13.11: Sampling from the bimodal distribution in Figure 13.10a. (a) HMC. (b) NUTS. (c) Tempered SMC with HMC kernel (single step). (d) Adaptive inverse temperature schedule. Generated by `smc_tempered_1d_bimodal.ipynb`.

13.6.3 Data tempering

If we have a set of iid observations, we can define the t 'th target to be

$$\tilde{\gamma}_t(\boldsymbol{\theta}) = p(\boldsymbol{\theta})p(\mathbf{y}_{1:t}|\boldsymbol{\theta}) \quad (13.80)$$

We can now apply SMC to this model. From Equation (13.74), the incremental weight becomes

$$\alpha_t(\boldsymbol{\theta}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} = \frac{p(\boldsymbol{\theta})p(\mathbf{y}_{1:t}|\boldsymbol{\theta})}{p(\boldsymbol{\theta})p(\mathbf{y}_{1:t-1}|\boldsymbol{\theta})} = p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \boldsymbol{\theta}) \quad (13.81)$$

This can be plugged into the generic SMC algorithm in Algorithm 13.3.

Unfortunately, to sample from the MCMC kernel will typically take $O(t)$ time, since the MH accept/reject step requires computing $p(\boldsymbol{\theta}') \prod_{i=1}^t p(\mathbf{y}_{1:i}|\boldsymbol{\theta}')$ for any proposed $\boldsymbol{\theta}'$. Hence the total cost is $O(T^2)$ if there are T observations. To reduce this, we can only sample parameters at times t when the ESS drops below a certain level; in the remaining steps, we just grow the sequence deterministically by repeating the previously sampled value. This technique was proposed in [Cho02], who called it the **iterated batch importance sampling** or **IBIS** algorithm.

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 $\boldsymbol{\theta} = (\mu, \sigma)$. The prior is $p(\boldsymbol{\theta}) = \mathcal{N}(\mu|0, 1)\text{Ga}(\sigma|a =$

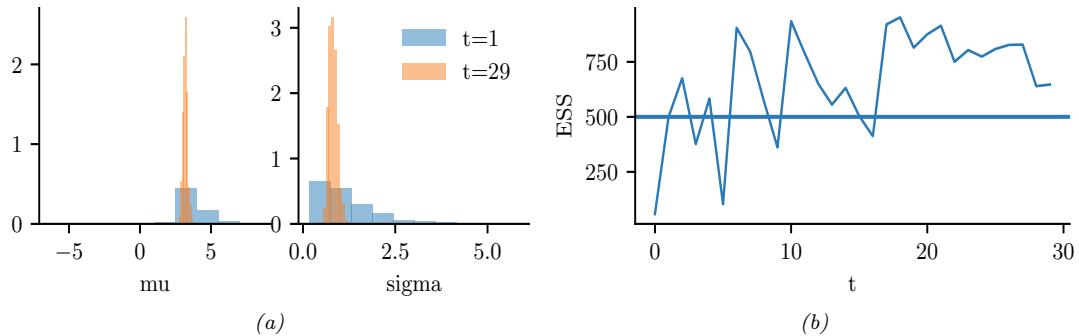


Figure 13.12: Illustration of IBIS applied to 30 samples from $N(\mu = 3.14, \sigma = 1)$. (a) Posterior approximation after $t = 1$ and $t = 29$ observations. (b) Effective sample size over time. The sudden jumps up occur whenever resampling is triggered, which happens when the ESS drops below 500. Generated by [smc_ibis_1d.ipynb](#).

$1, b = 1$). We use IBIS with an adaptive RWMH kernel. We use $N = 20$ particles, each updated for $K = 50$ MCMC steps, so we collect 1000 samples per time step.

Figure 13.12a 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 13.12b 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 3 times.

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. If λ^* is in the tail of the score distribution, this corresponds to sampling a **rare event**, which can be hard.

One approach is to use SMC to sample from a sequence of distributions with gradually increasing thresholds:

$$\pi_t(\boldsymbol{\theta}) = \frac{1}{Z_t} \mathbb{I}(S(\boldsymbol{\theta}) \geq \lambda_t) \pi_0(\boldsymbol{\theta}) \quad (13.82)$$

with $\lambda_0 < \dots < \lambda_T = \lambda^*$. We can then use likelihood tempering, where the “likelihood” is the function

$$G_t(\boldsymbol{\theta}_t) = \mathbb{I}(S(\boldsymbol{\theta}_t) \geq \lambda_t) \quad (13.83)$$

We can use SMC to generate samples from the final distribution π_T . We may also be interested in estimating

$$Z_T = p(S(\boldsymbol{\theta}) \geq \lambda_T) \quad (13.84)$$

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 score, and set λ_t to the α 'th highest 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** (Section 6.7.5). 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, $\mathbf{y} \sim p(\cdot | \boldsymbol{\theta})$, but where we cannot evaluate $p(\mathbf{y} | \boldsymbol{\theta})$ pointwise. 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 26.1).

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:

$$\pi_t(\boldsymbol{\theta}, \mathbf{y}) = \frac{1}{Z_t} \pi_0(\boldsymbol{\theta}) p(\mathbf{y} | \boldsymbol{\theta}) \mathbb{I}(d(\mathbf{y}, \mathbf{y}^*) < \epsilon_t) \quad (13.85)$$

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 (specifically the estimate \hat{Z}_t in Equation (13.10)) as a subroutine to approximate these likelihoods. This is called **SMC²**. For details, see [CP20b, Ch. 18].

13.6.7 Variational filtering SMC

One way to improve SMC is to learn a proposal distribution (e.g., using a neural network) such that the approximate posterior, $\hat{\pi}_T(\mathbf{z}_{1:T}; \boldsymbol{\phi}, \boldsymbol{\theta})$, is close to the target posterior, $\pi_T(\mathbf{z}_{1:T}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the model parameters, and $\boldsymbol{\phi}$ are the proposal parameters (which may depend on $\boldsymbol{\theta}$). One can show

[Nae+18] that the KL divergence between these distributions can be bounded as follows:

$$0 \leq D_{\text{KL}}(\mathbb{E}[\hat{\pi}_T(\mathbf{z}_{1:T})] \parallel \pi_T(\mathbf{z}_{1:T})) \leq -\mathbb{E}\left[\log \frac{\hat{Z}_T}{Z_T}\right] \quad (13.86)$$

where

$$Z_T(\boldsymbol{\theta}) = p_{\boldsymbol{\theta}}(\mathbf{y}_{1:T}) = \int p_{\boldsymbol{\theta}}(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) d\mathbf{z}_{1:T} \quad (13.87)$$

Hence

$$\mathbb{E}\left[\log \hat{Z}_T(\boldsymbol{\theta}, \phi)\right] \leq \mathbb{E}[\log Z_T(\boldsymbol{\theta})] = \log Z_T(\boldsymbol{\theta}) \quad (13.88)$$

Thus we can use SMC sampling to compute an unbiased approximation to $\mathbb{E}[\log \hat{Z}_T(\boldsymbol{\theta}, \phi)]$, which is a lower bound on the evidence (log marginal likelihood).

We can now maximize this lower bound wrt $\boldsymbol{\theta}$ and ϕ using SGD, as a way to learn both proposals and the model. Unfortunately, computing the gradient of the bound is tricky, since the resampling step is non-differentiable. However, in practice one can ignore the dependence of the resampling operator on the parameters, or one can use differentiable approximations (see e.g., [Ros+22]). This overall approach was independently proposed in several papers: the **FIVO** (filtering variational objective) paper [Mad+17], the **variational SMC** paper [Nae+18] and the **auto-encoding SMC** paper [Le+18].

13.6.8 Variational smoothing SMC

The methods in Section 13.6.7 use SMC in which the target distributions are defined to be the filtered distributions, $\pi_t(\mathbf{z}_{1:t}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t} | \mathbf{y}_{1:t})$; this is called **filtering SMC**. Unfortunately, this can work poorly when fitting models to offline sequence data, since at time t , all future observations are ignored in the objective, no matter how good the proposal. This can create situations where future observations are unlikely given the current set of sampled trajectories, which can result in particle impoverishment and high variance in the estimate of the lower bound.

Recently, a new method called **SIXO** (smoothing inference with twisted objectives) was proposed in [Law+22] that uses the smoothing distributions as targets, $\pi_t(\mathbf{z}_{1:t}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t} | \mathbf{y}_{1:T})$, to create a much lower variance variational lower bound. Of course it is impossible to directly compute this posterior, but we can approximate it using **twisted particle filters** [WL14a; AL+16]. In this approach, we approximate the (unnormalized) posterior using

$$p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:T}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_{1:t}, \mathbf{y}_{1:t}) \quad (13.89)$$

$$= p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t) \quad (13.90)$$

$$\approx p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) r_{\psi}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \quad (13.91)$$

where $r_{\psi}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \approx p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t)$ is the **twisting function**, which acts as a “**lookahead function**”.

One way to approximate the twisting function is to note that

$$p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t) = \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t | \mathbf{y}_{t+1:T}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)} \propto \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t | \mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)} \quad (13.92)$$

where we drop terms that are independent of \mathbf{z}_t since such terms will cancel out when we normalize the sampling weights. We can approximate the density ratio using the binary classifier method of Section 2.7.5. To do this, we define one distribution to be $p_1 = p_{\boldsymbol{\theta}}(\mathbf{z}_t, \mathbf{y}_{t+1:T})$ and the other to be $p_2 = p_{\boldsymbol{\theta}}(\mathbf{z}_t)p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T})$, so that $p_1/p_2 = \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t|\mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)}$. We can easily draw a sample $(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) \sim p_{\boldsymbol{\theta}}$ using ancestral sampling, from which we can compute $(\mathbf{z}_t, \mathbf{y}_{t+1:T}) \sim p_1$ by marginalization. We can also sample a fresh sequence from $(\tilde{\mathbf{z}}_{1:T}, \tilde{\mathbf{y}}_{1:T}) \sim p_{\boldsymbol{\theta}}$ from which we can compute $(\tilde{\mathbf{z}}_t, \tilde{\mathbf{y}}_{t+1:T}) \sim p_2$ by marginalization. We then use $(\mathbf{z}_t, \mathbf{y}_{t+1:T})$ as a positive example and $(\tilde{\mathbf{z}}_t, \tilde{\mathbf{y}}_{t+1:T})$ as a negative example when training the binary classifier, $r_{\boldsymbol{\psi}}(\mathbf{y}_{t+1:T}, \mathbf{z}_t)$.

Once we have updated the twisting parameters $\boldsymbol{\psi}$, we can rerun SMC to get a tighter lower bound on the log marginal likelihood, which we can then optimize wrt the model parameters $\boldsymbol{\theta}$ and proposal parameters $\boldsymbol{\phi}$. Thus the overall method is a stochastic variational EM-like method for optimziing the bound

$$\mathcal{L}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \triangleq \mathbb{E} \left[\log \hat{Z}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \right] \quad (13.93)$$

$$\leq \log \mathbb{E} \left[\hat{Z}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \right] = \log p_{\boldsymbol{\theta}}(\mathbf{y}_{1:T}) \quad (13.94)$$

In [Law+22] they prove the following: suppose the true model p^* is an SSM in which the optimal proposal function for the model satisfies $p^*(\mathbf{z}_t|\mathbf{z}_{1:t-1}, \mathbf{y}_{1:T}) \in \mathcal{Q}$, and the optimal lookahead function for the model satisfies $p^*(\mathbf{y}_{t+1:T}|\mathbf{z}_t) \in \mathcal{R}$. Furthermore, assume the SIXO objective has a unique maximizer. Then, at the optimum, we have that the learned proposal $q_{\boldsymbol{\phi}^*}(\mathbf{z}_t|\mathbf{z}_{1:t-1}, \mathbf{y}_{1:T}) \in \mathcal{Q}$ is equal to the optimal proposal, the learned twisting function $r_{\boldsymbol{\psi}^*}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \in \mathcal{R}$ is equal to the optimal lookahead, and the lower bound is tight (i.e., $\mathcal{L}_{\text{SIXO}}(\boldsymbol{\theta}^*, \boldsymbol{\phi}^*, \boldsymbol{\psi}^*) = p^*(\mathbf{y}_{1:T})$) for any number of samples $N_s \geq 1$ and for any kind of SSM p^* . (This is in contrast to the FIVO bound, whiere the bound does not usually become tight.)

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 20). 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

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{P}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] + \frac{1}{\lambda} D_{\text{KL}}(q \parallel \pi_0) \quad (14.8)$$

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:

$$\hat{q} = \operatorname{argmin}_{q \in \mathcal{Q}(\Theta)} \mathbb{E}_{q(\boldsymbol{\theta})} [-\log p(\mathcal{D} | \boldsymbol{\theta})] + D_{\text{KL}}(q \parallel \pi_0) \quad (14.12)$$

This is known as **variational inference**; see Chapter 10 for details.

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].

14.2 Evaluating predictive models

In this section we discuss how to evaluate the quality of a trained discriminative model.

14.2.1 Proper scoring rules

It is common to measure performance of a predictive model using a **proper scoring rule** [GR07], which is defined as follows. Let $S(p_{\theta}, (y, \mathbf{x}))$ be the score for predictive distribution $p_{\theta}(y|\mathbf{x})$ when given an event $y|\mathbf{x} \sim p^*(y|\mathbf{x})$, where p^* is the true conditional distribution. (If we want to evaluate a Bayesian model, where we marginalize out θ rather than condition on it, we just replace $p_{\theta}(y|\mathbf{x})$ with $p(y|\mathbf{x}) = \int p_{\theta}(y|\mathbf{x})p(\theta|\mathcal{D})d\theta$.) The expected score is defined by

$$S(p_{\theta}, p^*) = \int p^*(\mathbf{x})p^*(y|\mathbf{x})S(p_{\theta}, (y, \mathbf{x}))dyd\mathbf{x} \quad (14.14)$$

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 maximizing such a proper scoring rule will force the model to match the true probabilities.

The log-likelihood, $S(p_{\theta}, (y, \mathbf{x})) = \log p_{\theta}(y|\mathbf{x})$, is a proper scoring rule. This follows from Gibbs inequality:

$$S(p_{\theta}, p^*) = \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p_{\theta}(y|\mathbf{x})] \leq \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p^*(y|\mathbf{x})] \quad (14.15)$$

Therefore minimizing the NLL (aka log loss) should result in well-calibrated probabilities. However, in practice, log-loss can over-emphasize tail probabilities [QC+06].

A common alternative is to use the **Brier score** [Bri50], which is defined as follows:

$$S(p_{\theta}, (y, \mathbf{x})) \triangleq \frac{-1}{C} \sum_{c=1}^C (p_{\theta}(y=c|\mathbf{x}) - \mathbb{I}(y=c))^2 \quad (14.16)$$

This is proportional to the squared error of the predictive distribution $\mathbf{p} = p(1:C|\mathbf{x})$ compared to the one-hot label distribution \mathbf{y} . (We add a negative sign to the original definition so that larger values (less negative) are better, to be consistent with the conventions above.) Since it is based on squared error, the Brier score is less sensitive to extremely rare or extremely common classes. The Brier score is also a proper scoring rule.

14.2.2 Calibration

A model whose predicted probabilities match the empirical frequencies is said to be **calibrated** [Daw82; NMC05; Guo+17]. For example, if a classifier predicts $p(y=c|\mathbf{x}) = 0.9$, then we expect this to be the true label about 90% of the time. A well-calibrated model is useful to avoid making the wrong decision when the outcome is too uncertain. In the sections below, we discuss some ways to measure and improve calibration.

14.2.2.1 Expected calibration error

To assess calibration, we divide the predicted probabilities into a finite set of bins or buckets, and then assess the discrepancy between the empirical probability and the predicted probability by counting.

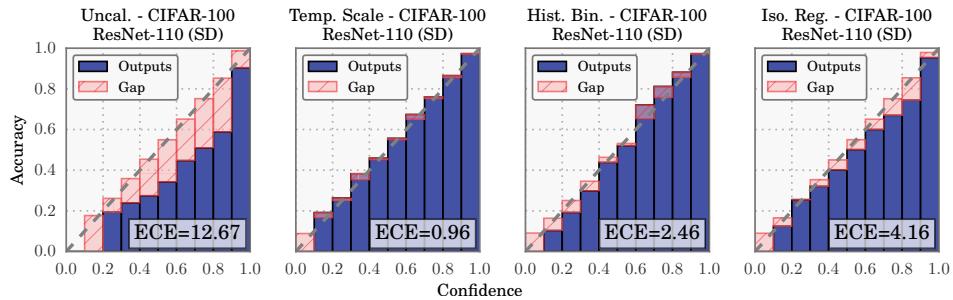


Figure 14.1: Reliability diagrams for the ResNet CNN image classifier [He+16b] applied to CIFAR-100 dataset. ECE is the expected calibration error, and measures the size of the red gap. Methods from left to right: original probabilities; after temperature scaling; after histogram binning; after isotonic regression. From Figure 4 of [Guo+17]. Used with kind permission of Chuan Guo.

More precisely, suppose we have B bins. Let \mathcal{B}_b be the set of indices of samples whose prediction 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 $f(\mathbf{x})_c = p(y = c|\mathbf{x})$, $\hat{y}_n = \operatorname{argmax}_{c \in \{1, \dots, C\}} f(\mathbf{x}_n)_c$, 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]:

$$\text{MCE} = \sum_{c=1}^C w_c \mathbb{E} [(p(Y = c|f(\mathbf{x})_c) - f(\mathbf{x})_c)^2] \quad (14.20)$$

$$= \sum_{c=1}^C w_c \sum_{b=1}^B \frac{|\mathcal{B}_{b,c}|}{B} (\text{acc}(\mathcal{B}_{b,c}) - \text{conf}(\mathcal{B}_{b,c}))^2 \quad (14.21)$$

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 $w_c = 1/C$ if all classes are equally important.) In [Nix+19], they call this metric **static calibration error**; they show that certain methods that have good ECE may have poor MCE. Other multi-class calibration metrics are discussed in [WLZ19].

14.2.2.2 Improving calibration

In principle, training a classifier so it optimizes a proper scoring rule (such as NLL) should automatically result in a well-calibrated classifier. In practice, however, unbalanced datasets can result in poorly calibrated predictions. Below we discuss various ways for improving the calibration of probabilistic classifiers, following [Guo+17].

14.2.2.3 Platt scaling

Let z be the log-odds, or logit, and $p = \sigma(z)$, produced by a probabilistic binary classifier. We wish to convert this to a more calibrated value q . The simplest way to do this is known as **Platt scaling**, and was proposed in [Pla00]. The idea is to compute $q = \sigma(az + b)$, where a and b are estimated via maximum likelihood on a validation set.

In the multiclass case, we can extend Platt scaling by using matrix scaling: $\mathbf{q} = \text{softmax}(\mathbf{W}\mathbf{z} + \mathbf{b})$, where we estimate \mathbf{W} and \mathbf{b} via maximum likelihood on a validation set. Since \mathbf{W} has $K \times K$ parameters, where K is the number of classes, this method can easily overfit, so in practice we restrict \mathbf{W} to be diagonal.

14.2.2.4 Nonparametric (histogram) methods

Platt scaling makes a strong assumption about how the shape of the calibration curve. A more flexible, nonparametric, method is to partition the predicted probabilities into bins, p_m , and to estimate an empirical probability q_m for each such bin; we then replace p_m with q_m ; this is known as **histogram binning** [ZE01a]. We can regularize this method by requiring that $q = f(p)$ be a piecewise constant, monotonically non-decreasing function; this is known as **isotonic regression** [ZE01a]. An alternative approach, known as the **scaling-binning calibrator**, is to apply a scaling method (such as Platt scaling), and then to apply histogram binning to that. This has the advantage of using the average of the scaled probabilities in each bin instead of the average of the observed binary labels (see Figure 14.2). In [KLM19], they prove that this results in better calibration, due to the lower variance of the estimator.

In the multiclass case, \mathbf{z} is the vector of logits, and $\mathbf{p} = \text{softmax}(\mathbf{z})$ is the vector of probabilities. We wish to convert this to a better calibrated version, \mathbf{q} . [ZE01b] propose to extend histogram binning and isotonic regression to this case by applying the above binary method to each of the K one-vs-rest problems, where K is the number of classes. However, this requires K separate calibration models, and results in an unnormalized probability distribution.

14.2.2.5 Temperature scaling

In [Guo+17], they noticed empirically that the diagonal version of Platt scaling, when applied to a variety of DNNs, often ended learning a vector of the form $\mathbf{w} = (c, c, \dots, c)$, for some constant c . This suggests a simpler form of scaling, which they call **temperature scaling**: $\mathbf{q} = \text{softmax}(\mathbf{z}/T)$,

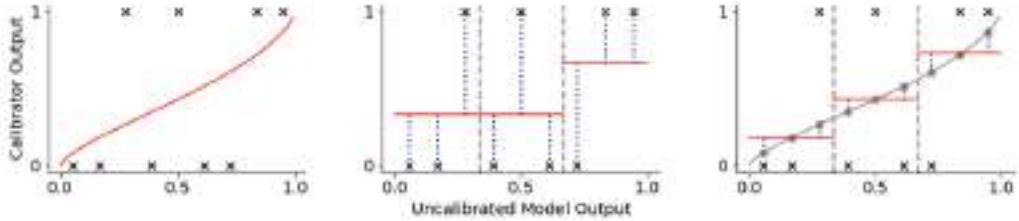


Figure 14.2: Visualization of 3 different approaches to calibrating a binary probabilistic classifier. Black crosses are the observed binary labels, red lines are the calibrated outputs. (a) Platt scaling. (b) Histogram binning with 3 bins. The output in each bin is the average of the binary labels in 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.

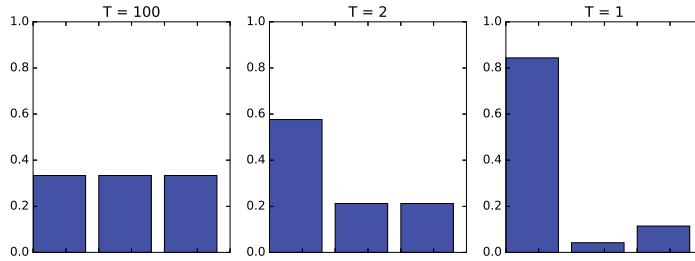


Figure 14.3: Softmax distribution $\text{softmax}(\mathbf{a}/T)$, where $\mathbf{a} = (3, 0, 1)$, at temperatures of $T = 100$, $T = 2$ and $T = 1$. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is “spiky”, with most of its mass on the largest element. Generated by softmax_plot.ipynb.

where $T > 0$ 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].

14.2.2.7 Bayesian methods

Bayesian approaches to fitting classifiers often result in more calibrated predictions, since they represent uncertainty in the parameters. See Section 17.3.8 for an example. However, [Ova+19] shows that well-calibrated models (even Bayesian ones) often become mis-calibrated when applied to inputs that come from a different distribution (see Section 19.2 for details).

14.2.3 Beyond evaluating marginal probabilities

Calibration (Section 14.2.2) focuses on assessing properties of the marginal predictive distribution $p(y|\mathbf{x})$. But this can sometimes be insufficient to distinguish between a good and bad model, especially in the context of online learning and sequential decision making, as pointed out in [Lu+22; Osb+21; WSG21; KKG22]. For example, consider two learning agents who observe a sequence of coin tosses. Let the outcome at time t be $Y_t \sim \text{Ber}(\theta)$, where θ is the unknown parameter. Agent 1 believes $\theta = 2/3$, whereas agent 2 believes either $\theta = 0$ or $\theta = 1$, but is not sure which, and puts probabilities 1/3 and 2/3 on these events. Thus both agents, despite having different models, make identical predictions for the next outcome: $p(Y_1^i = 0) = 1/3$ for agents $i = 1, 2$. However, the predictions of the two agents about a *sequence* of τ future outcomes is very different: In particular, agent 1 predicts each individual coin toss is a random Bernoulli event, where the probability is due to irreducible noise or **aleatoric uncertainty**:

$$p(Y_1^1 = 0, \dots, Y_\tau^1 = 0) = \frac{1}{3^\tau} \quad (14.22)$$

By contrast, agent 2 predicts that the sequence will either be all heads or all tails, where the probability is induced by **epistemic uncertainty** about the true parameters:

$$p(Y_1^2 = y_1, \dots, Y_\tau^2 = y_\tau) = \begin{cases} 1/3 & \text{if } y_1 = \dots = y_\tau = 0 \\ 2/3 & \text{if } y_1 = \dots = y_\tau = 1 \\ 0 & \text{otherwise} \end{cases} \quad (14.23)$$

The difference in beliefs between these agents will impact their behavior. For example, in a casino, agent 1 incurs little risk on repeatedly betting on heads in the long run, but for agent 2, this would be a very unwise strategy, and some initial information gathering (exploration) would be worthwhile.

Based on the above, we see that it is useful to evaluate *joint* predictive distributions when assessing predictive models. In [Lu+22; Osb+21] they propose to evaluate the posterior predictive distributions 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, $\mathcal{D}_T = \{(X_t, Y_{t+1}) : t = 0, 1, \dots, T-1\}$. The Bayes optimal predictive distribution is

$$P_T^B = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.24)$$

This is usually intractable to compute. Instead the agent will use an approximate distribution, known as a **belief state**, which we denote by

$$Q_T = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.25)$$

The natural performance metric is the KL between these distributions. Since this depend on the inputs \mathbf{x} and $\mathcal{D}_T = (X_{0:T-1}, Y_{1:T})$, we will averaged 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

$$p_{jn} = P(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{E}^j) = \prod_{t=T}^{T+\tau-1} P(Y_{n,t+1}^j|X_{n,t}^j, \mathcal{E}^j) \quad (14.33)$$

$$q_{jn} = Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{D}_T^j) = \int Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \boldsymbol{\theta})Q(\boldsymbol{\theta}|\mathcal{D}_T^j)d\boldsymbol{\theta} \quad (14.34)$$

$$\approx \frac{1}{M} \sum_{m=1}^M \prod_{t=T}^{T+\tau-1} Q(Y_{n,t+1}^j|X_{n,t}^j, \boldsymbol{\theta}_m^j) \quad (14.35)$$

where $\theta_m^j \sim Q(\theta|\mathcal{D}_T^j)$ is a sample from the agent's posterior over the environment.

The above assumes that $P(Y|X)$ is known; this will be the case if we use a synthetic data generator, as in the “neural testbed” in [Osb+21]. If we just have an J empirical distributions for $P^j(X, Y)$, we can replace the KL with the cross entropy, which only differs by an additive constant:

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{E}) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.36)$$

$$= \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{E})} [\log P(\mathbf{y}|\mathbf{x}, \mathcal{E})]}_{\text{const}} - \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T | \mathcal{E}) P(\mathcal{E})} [\log Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)]}_{d_{\mathcal{E},Q}^{CE}} \quad (14.37)$$

where the latter term is just the empirical negative log likelihood (NLL) of the agent on samples from the environment. Hence if we rank agents in terms of their NLL or cross entropy $d_{\mathcal{E},Q}^{CE}$ we will get the same results as ranking them by $d_{\mathcal{E},Q}^{KL}$, which will in turn give the same results as ranking them by $d_{B,Q}^{KL}$.

In practice we can approximate the cross entropy as follows:

$$\hat{d}_{\mathcal{E},Q}^{CE} = -\frac{1}{JN} \sum_{j=1}^J \sum_{n=1}^N \log Q(\mathbf{y}_n^j | \mathbf{x}_n^j, \mathcal{D}_T^j) \quad (14.38)$$

where $\mathcal{D}_T^j \sim P^j$, and $(\mathbf{x}_n^j, \mathbf{y}_n^j) \sim P^j$.

An alternative to estimating the KL or NLL is to evaluate the joint predictive accuracy by using it in a downstream task. In [Osb+21], they show that good predictive accuracy (for $\tau > 1$) correlates with good performance on a bandit problem (see Section 34.4). In [WSG21] they show that good predictive accuracy (for $\tau > 1$) results in good performance on a transductive active learning task.

14.2.3.1 Proof of claim

We now prove Equation (14.29), based on [Lu+21a]. First note that

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E}) P(\mathbf{y}|\mathbf{x}, \mathcal{E})} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.39)$$

$$= \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] + \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.40)$$

For the first term in Equation (14.40) we have

$$\mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] = \sum P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.41)$$

$$= \sum P(\mathbf{x}, \mathcal{D}_T) \sum P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.42)$$

$$= \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T)} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] = d_{B,Q}^{KL} \quad (14.43)$$

We now show that the second term in Equation (14.40) reduces to the mutual information. We exploit the fact that

$$P(\mathbf{y}|\mathbf{x}, \mathcal{E}) = P(\mathbf{y}|\mathcal{D}_T, \mathbf{x}, \mathcal{E}) = \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})}{P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})} \quad (14.44)$$

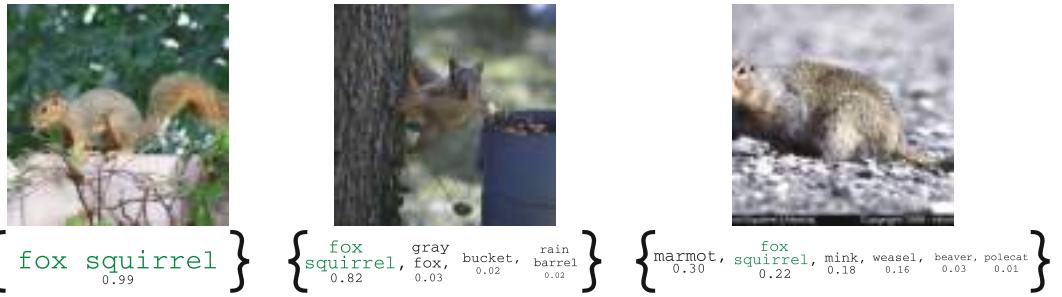


Figure 14.4: Prediction set examples on Imagenet. We show three progressively more difficult examples of the class fox squirrel and the prediction sets generated by conformal prediction. (Compare to Figure 17.9.) From Figure 1 of [AB21]. Used with kind permission of Anastasios Angelopoulos.

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; Man22b]. 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; Bha+23]. The exchangeability assumption also rules out distribution shift, although extensions to this case have also been developed [Tib+19].

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 [Man22a].

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 $\mathcal{S} = \{s_i = s(\mathbf{x}_i, y_i) : i = 1 : n\}$.³ 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 $P^*(\mathbf{x}_1, y_1, \dots, \mathbf{x}_{n+1}, y_{n+1})$. 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.

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 such degenerate cases, we seek prediction sets that are as small as possible (although we allow for the set to be larger for 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.⁴ It is also important to note that the coverage guarantees are frequentist in nature, and refer to average behavior, rather than representing per-instance uncertainty, as in the Bayesian approach.

3. Using a calibration set is called **split conformal prediction**. If we don’t have enough data to adopt this splitting approach, we can use **full conformal prediction** [VGS05], which requires fitting the model n times using a leave-one-out type procedure.

4. It is also possible to learn conformal scores in an end-to-end way, jointly with the predictive model, as discussed in [Stu+22].

14.3. Conformal prediction

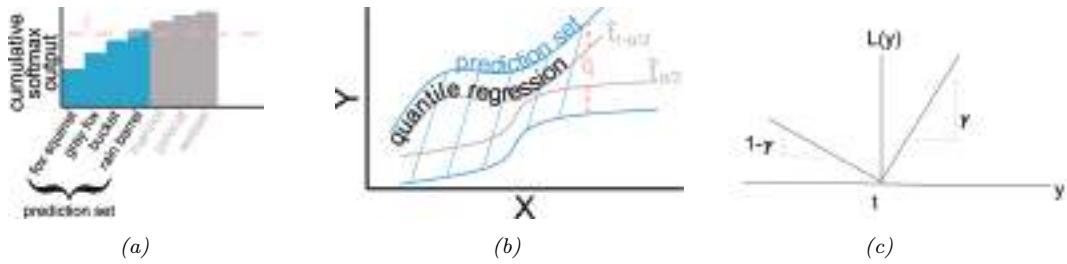


Figure 14.5: (a) Illustration of adaptive prediction set. From Figure 5 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (b) Illustrate of conformalized quantile regression. From Figure 6 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (c) Illustration of pinball loss function.

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 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 cdf $P_Y(Y) = p(Y|\mathbf{x})$. 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

$$\ell_\gamma(y, \hat{t}) = (\gamma - \mathbb{I}(y < \hat{t}))(y - \hat{t}) = (y - \hat{t})\gamma\mathbb{I}(y > \hat{t}) + (\hat{t} - y)(1 - \gamma)\mathbb{I}(y < \hat{t}) \quad (14.53)$$

where y is the true output and \hat{t} is the predicted value at quantile γ . See Figure 14.5c for an illustration. The key property of this loss function is that its minimizer is the γ -quantile of the distribution P_Y , i.e.,

$$\underset{\hat{t}}{\operatorname{argmin}} \mathbb{E}_{p_Y(y)} [\ell_\gamma(y, \hat{t})] = P_Y^{-1}(\gamma)$$

However, the regression quantiles are usually only approximately a 90% interval because the model may be mismatched to the true distribution. Fortunately 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.

15 Generalized linear models

15.1 Introduction

A **generalized linear model** or **GLM** [MN89] is a conditional version of an exponential family distribution (Section 2.4). 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 2.4.3, 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:

$$\ell(\mu_n) = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.4)$$

GLMs are quite limited in their predictive power, due to the assumption of linearity (although we can always use basis function expansion on \mathbf{x}_n to improve the flexibility). However, the main use of GLMs in the statistics literature is not for prediction, but for hypothesis testing, as we explain in Section 3.10.3. This relies on the ability to compute the posterior, $p(\mathbf{w}|\mathcal{D})$, which we discuss in Section 15.1.4. We can use this to draw conclusions about whether any of the inputs (e.g., representing different groups) have a significant effect on the output.

15.1.1 Some popular GLMs

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

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2}(y_n - \eta_n)^2 - \frac{1}{2} \log(2\pi\sigma^2) \quad (15.6)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. We can write this in GLM form as follows:

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \frac{y_n \eta_n - \frac{\eta_n^2}{2}}{\sigma^2} - \frac{1}{2} \left(\frac{y_n^2}{\sigma^2} + \log(2\pi\sigma^2) \right) \quad (15.7)$$

We see that $A(\eta_n) = \eta_n^2/2$ and hence

$$\mathbb{E}[y_n] = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.8)$$

$$\mathbb{V}[y_n] = \sigma^2 \quad (15.9)$$

See Section 15.2 for details on linear regression.

15.1.1.2 Binomial regression

If the response variable is the number of successes in N_n trials, $y_n \in \{0, \dots, N_n\}$, we can use **binomial regression**, which is defined by

$$p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = \text{Bin}(y_n | \sigma(\mathbf{w}^\top \mathbf{x}_n), N_n) \quad (15.10)$$

We see that binary logistic regression is the special case when $N_n = 1$.

The log pdf is given by

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \log \mu_n + (N_n - y_n) \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.11)$$

$$= y_n \log\left(\frac{\mu_n}{1 - \mu_n}\right) + N_n \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.12)$$

where $\mu_n = \sigma(\eta_n)$. To rewrite this in GLM form, let us define

$$\eta_n \triangleq \log \left[\frac{\mu_n}{(1 - \mu_n)} \right] = \log \left[\frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}} \frac{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}}{e^{-\mathbf{w}^\top \mathbf{x}_n}} \right] = \log \frac{1}{e^{-\mathbf{w}^\top \mathbf{x}_n}} = \mathbf{w}^\top \mathbf{x}_n \quad (15.13)$$

Hence we can write binomial regression in GLM form as follows

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \eta_n - A(\eta_n) + h(y_n) \quad (15.14)$$

where $h(y_n) = \log \binom{N_n}{y_n}$ and

$$A(\eta_n) = -N_n \log(1 - \mu_n) = N_n \log(1 + e^{\eta_n}) \quad (15.15)$$

Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = \frac{N_n e^{\eta_n}}{1 + e^{\eta_n}} = \frac{N_n}{1 + e^{-\eta_n}} = N_n \mu_n \quad (15.16)$$

and

$$\mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = N_n \mu_n (1 - \mu_n) \quad (15.17)$$

See Section 15.3.9 for an example of binomial regression.

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

$$\mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = e^{\eta_n} = \mu_n \quad (15.23)$$

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 supply is zero)? 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 choose 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 $\lambda^0 e^{-\lambda} / 0! = e^{-\lambda}$.)

15.1.2 GLMs with noncanonical link functions

We have seen how the mean parameters of the output distribution are given by $\mu = \ell^{-1}(\eta)$, where the function ℓ is the link function. There are several choices for this function, as we now discuss.

The **canonical link function** ℓ satisfies the property that $\theta = \ell(\mu)$, where θ are the canonical (natural) parameters. Hence

$$\theta = \ell(\mu) = \ell(\ell^{-1}(\eta)) = \eta \quad (15.25)$$

This is what we have assumed so far. For example, for the Bernoulli distribution, the canonical parameter is the log-odds $\eta = \log(\mu/(1 - \mu))$, which is given by the logit transform

$$\eta = \ell(\mu) = \text{logit}(\mu) = \log\left(\frac{\mu}{1 - \mu}\right) \quad (15.26)$$

The inverse of this is the sigmoid or logistic function

$$\mu = \ell^{-1}(\eta) = \sigma(\eta) = 1/(1 + e^{-\eta}) \quad (15.27)$$

However, we are free to use other kinds of link function. For example, in Section 15.4 we use

$$\eta = \ell(\mu) = \Phi^{-1}(\mu) \quad (15.28)$$

$$\mu = \ell^{-1}(\eta) = \Phi(\eta) \quad (15.29)$$

This is known as the **probit link function**.

Another link function that is sometimes used for binary responses is the **complementary log-log** function

$$\eta = \ell(\mu) = \log(-\log(1 - \mu)) \quad (15.30)$$

This is used in applications where we either observe 0 events (denoted by $y = 0$) or one or more (denoted by $y = 1$), where events are assumed to be governed by a Poisson distribution with rate λ . Let E be the number of events. The Poisson assumption means $p(E = 0) = \exp(-\lambda)$ and hence

$$p(y = 0) = (1 - \mu) = p(E = 0) = \exp(-\lambda) \quad (15.31)$$

Thus $\lambda = -\log(1 - \mu)$. When λ is a function of covariates, we need to ensure it is positive, so we use $\lambda = e^\eta$, and hence

$$\eta = \log(\lambda) = \log(-\log(1 - \mu)) \quad (15.32)$$

15.1.3 Maximum likelihood estimation

GLMs can be fit using similar methods to those that we used to fit logistic regression. In particular, the negative log-likelihood has the following form (ignoring constant terms):

$$\text{NLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) = -\frac{1}{\sigma^2} \sum_{n=1}^N \ell_n \quad (15.33)$$

where

$$\ell_n \triangleq \eta_n y_n - A(\eta_n) \quad (15.34)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. For notational simplicity, we will assume $\sigma^2 = 1$.

We can compute the gradient for a single term as follows:

$$\mathbf{g}_n \triangleq \frac{\partial \ell_n}{\partial \mathbf{w}} = \frac{\partial \ell_n}{\partial \eta_n} \frac{\partial \eta_n}{\partial \mathbf{w}} = (y_n - A'(\eta_n)) \mathbf{x}_n = (y_n - \mu_n) \mathbf{x}_n \quad (15.35)$$

where $\mu_n = f(\mathbf{w}^\top \mathbf{x}_n)$, and f is the inverse link function that maps from canonical parameters to mean parameters. (For example, in the case of logistic regression, we have $\mu_n = \sigma(\mathbf{w}^\top \mathbf{x})$.)

The Hessian is given by

$$\mathbf{H} = \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}^\top} \text{NLL}(\mathbf{w}) = - \sum_{n=1}^N \frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} \quad (15.36)$$

where

$$\frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} = \frac{\partial \mathbf{g}_n}{\partial \mu_n} \frac{\partial \mu_n}{\partial \mathbf{w}^\top} = -\mathbf{x}_n f'(\mathbf{w}^\top \mathbf{x}_n) \mathbf{x}_n^\top \quad (15.37)$$

Hence

$$\mathbf{H} = \sum_{n=1}^N f'(\eta_n) \mathbf{x}_n \mathbf{x}_n^\top \quad (15.38)$$

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 general, we see that the Hessian is positive definite, since $f'(\eta_n) > 0$; hence the negative log likelihood is convex, so the MLE for a GLM is unique (assuming $f(\eta_n) > 0$ for all n).

For small datasets, we can use the **iteratively reweighted least squares** or **IRLS** algorithm, which is a form of Newton's method, to compute the MLE (see e.g., [Mur22, Sec 10.2.6]). For large datasets, we can use SGD. (In practice it is often useful to combine SGD with methods that automatically tune the step size, such as [Loi+21].)

15.1.4 Bayesian inference

Maximum likelihood estimation provides a point estimate of the parameters, but does not convey any notion of uncertainty, which is important for hypothesis testing, as we explain in Section 3.10.3,

as well as for avoiding overfitting. To compute the uncertainty, we will perform Bayesian inference of the parameters. To do this, we first need to specify a prior. Choosing a suitable prior depends on the form of link function. For example, a “flat” or “uninformative” prior on the offset term $\alpha \in \mathbb{R}$ will not translate to an uninformative prior on the probability scale if we pass α through a sigmoid, as we discuss in Section 15.3.4.

Once we have chosen the prior, we can compute the posterior using a variety of approximate inference methods. For small datasets, HMC (Section 12.5) is the easiest to use, since you just need to write down the log likelihood and log prior; we can then use autograd to compute derivatives which can be passed to the HMC engine (see e.g., [BG13] for details).

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/BlackJAX, **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. These libraries support a convenient **formula syntax**, initially created in the R language, for compactly specifying the form of the model, including possible interaction terms between the inputs.

For large datasets, HMC can be slow, since it is a full batch algorithm. In such settings, variational Bayes (see e.g., [HOW11; TN13]), expectation propagation (see e.g., [KW18]), or more specialized algorithms (e.g., [HAB17]) are the best choice.

15.2 Linear regression

Linear regression is the simplest case of a GLM, and refers to the following model:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + \mathbf{w}^\top \mathbf{x}, \sigma^2) \quad (15.39)$$

where $\boldsymbol{\theta} = (w_0, \mathbf{w}, \sigma^2)$ are all the parameters of the model. (In statistics, the parameters w_0 and \mathbf{w} are usually denoted by β_0 and $\boldsymbol{\beta}$.) We gave a detailed introduction to this model in the prequel to this book, [Mur22]. In this section, we briefly discuss maximum likelihood estimation, and then focus on a Bayesian analysis.

15.2.1 Ordinary least squares

From Equation (15.39), we can derive the negative log likelihood of the data as follows:

$$\text{NLL}(\mathbf{w}, \sigma^2) = -\sum_{n=1}^N \log \left[\left(\frac{1}{2\pi\sigma^2} \right)^{\frac{1}{2}} \exp \left(-\frac{1}{2\sigma^2} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 \right) \right] \quad (15.40)$$

$$= \frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \hat{y}_n)^2 + \frac{N}{2} \log(2\pi\sigma^2) \quad (15.41)$$

where we have defined the predicted response $\hat{y}_n \triangleq \mathbf{w}^\top \mathbf{x}_n$. In [Mur22, Sec 11.2.2] we show that the MLE is given by

$$\hat{\mathbf{w}}_{\text{mle}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.42)$$

This is called the **ordinary least squares (OLS)** solution.

The MLE for the observation noise is given by

$$\hat{\sigma}_{\text{mle}}^2 = \underset{\sigma^2}{\operatorname{argmin}} \text{NLL}(\hat{\mathbf{w}}, \sigma^2) = \frac{1}{N} \sum_{n=1}^N (y_n - \mathbf{x}_n^\top \hat{\mathbf{w}})^2 \quad (15.43)$$

This is just the mean squared error of the residuals, which is an intuitive result.

15.2.2 Conjugate priors

In this section, we derive the posterior for the parameters using a conjugate prior. We first consider the case where just \mathbf{w} is unknown (so the observation noise variance parameter σ^2 is fixed), and then we consider the general case, where both σ^2 and \mathbf{w} are unknown.

15.2.2.1 Noise variance is known

The conjugate prior for linear regression has the following form:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \check{\mathbf{w}}, \check{\Sigma}) \quad (15.44)$$

We often use $\check{\mathbf{w}} = \mathbf{0}$ as the prior mean and $\check{\Sigma} = \tau^2 \mathbf{I}_D$ as the prior covariance. (We assume the bias term is included in the weight vector, but often use a much weaker prior for it, since we typically do not want to regularize the overall mean level of the output.)

To derive the posterior, let us first rewrite the likelihood in terms of an MVN as follows:

$$\ell(\mathbf{w}) = p(\mathcal{D} | \mathbf{w}, \sigma^2) = \prod_{n=1}^N p(y_n | \mathbf{w}^\top \mathbf{x}_n, \sigma^2) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \quad (15.45)$$

where \mathbf{I}_N is the $N \times N$ identity matrix. We can then use Bayes' rule for Gaussians (Equation (2.121)) to derive the posterior, which is as follows:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}, \sigma^2) \propto \mathcal{N}(\mathbf{w} | \check{\mathbf{w}}, \check{\Sigma}) \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) = \mathcal{N}(\mathbf{w} | \hat{\mathbf{w}}, \hat{\Sigma}) \quad (15.46)$$

$$\hat{\mathbf{w}} \triangleq \hat{\Sigma}^{-1} (\check{\Sigma}^{-1} \check{\mathbf{w}} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{y}) \quad (15.47)$$

$$\hat{\Sigma} \triangleq (\check{\Sigma}^{-1} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.48)$$

where $\hat{\mathbf{w}}$ is the posterior mean, and $\hat{\Sigma}$ is the posterior covariance.

Now suppose $\check{\mathbf{w}} = \mathbf{0}$ and $\check{\Sigma} = \tau^2 \mathbf{I}$. In this case, the posterior mean becomes

$$\hat{\mathbf{w}} = \frac{1}{\sigma^2} \hat{\Sigma} \mathbf{X}^\top \mathbf{y} = \left(\frac{\sigma^2}{\tau^2} \mathbf{I} + \mathbf{X}^\top \mathbf{X} \right)^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.49)$$

If we define $\lambda = \frac{\sigma^2}{\tau^2}$, we see this is equivalent to **ridge regression**, which optimizes

$$\mathcal{L}(\mathbf{w}) = \text{RSS}(\mathbf{w}) + \lambda \|\mathbf{w}\|^2 \quad (15.50)$$

where RSS is the residual sum of squares:

$$\text{RSS}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 = \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \frac{1}{2} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \quad (15.51)$$

15.2.2.2 Noise variance is unknown

In this section, we assume \mathbf{w} and σ^2 are both unknown. The likelihood is given by

$$\ell(\mathbf{w}, \sigma^2) = p(\mathcal{D} | \mathbf{w}, \sigma^2) \propto (\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right) \quad (15.52)$$

Since the regression weights now depend on σ^2 in the likelihood, the conjugate prior for \mathbf{w} has the form

$$p(\mathbf{w} | \sigma^2) = \mathcal{N}(\mathbf{w} | \check{\mathbf{w}}, \sigma^2 \check{\Sigma}) \quad (15.53)$$

For the noise variance σ^2 , the conjugate prior is based on the inverse gamma distribution, which has the form

$$\text{IG}(\sigma^2 | \check{a}, \check{b}) = \frac{\check{b}^{\check{a}}}{\Gamma(\check{a})} (\sigma^2)^{-(\check{a}+1)} \exp(-\frac{\check{b}}{\sigma^2}) \quad (15.54)$$

(See Section 2.2.3.4 for more details.) Putting these two together, we find that the joint conjugate prior is the **normal inverse gamma** distribution:

$$\text{NIG}(\mathbf{w}, \sigma^2 | \check{\mathbf{w}}, \check{\Sigma}, \check{a}, \check{b}) \triangleq \mathcal{N}(\mathbf{w} | \check{\mathbf{w}}, \sigma^2 \check{\Sigma}) \text{IG}(\sigma^2 | \check{a}, \check{b}) \quad (15.55)$$

$$\begin{aligned} &= \frac{\check{b}^{\check{a}}}{(2\pi)^{D/2} |\check{\Sigma}|^{1/2} \Gamma(\check{a})} (\sigma^2)^{-(\check{a}+(D/2)+1)} \\ &\times \exp\left[-\frac{(\mathbf{w} - \check{\mathbf{w}})^\top \check{\Sigma}^{-1} (\mathbf{w} - \check{\mathbf{w}}) + 2\check{b}}{2\sigma^2}\right] \end{aligned} \quad (15.56)$$

This results in the following posterior:

$$p(\mathbf{w}, \sigma^2 | \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.57)$$

$$\hat{\mathbf{w}} = \hat{\Sigma} (\check{\Sigma}^{-1} \check{\mathbf{w}} + \mathbf{X}^\top \mathbf{y}) \quad (15.58)$$

$$\hat{\Sigma} = (\check{\Sigma}^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.59)$$

$$\hat{a} = \check{a} + N/2 \quad (15.60)$$

$$\hat{b} = \check{b} + \frac{1}{2} \left(\check{\mathbf{w}}^\top \check{\Sigma}^{-1} \check{\mathbf{w}} + \mathbf{y}^\top \mathbf{y} - \hat{\mathbf{w}}^\top \hat{\Sigma}^{-1} \hat{\mathbf{w}} \right) \quad (15.61)$$

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 intuitive, since it just updates the counts. The expression for \hat{b} can be interpreted as follows: it is 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 the prior on \mathbf{w} .

The posterior marginals are as follows. For the variance, we have

$$p(\sigma^2 | \mathcal{D}) = \int p(\mathbf{w} | \sigma^2, \mathcal{D}) p(\sigma^2 | \mathcal{D}) d\mathbf{w} = \text{IG}(\sigma^2 | \hat{a}, \hat{b}) \quad (15.62)$$

For the regression weights, it can be shown that

$$p(\mathbf{w} | \mathcal{D}) = \int p(\mathbf{w} | \sigma^2, \mathcal{D}) p(\sigma^2 | \mathcal{D}) d\sigma^2 = \mathcal{T}(\mathbf{w} | \hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}}, \hat{\Sigma}, 2\hat{a}) \quad (15.63)$$

15.2.2.3 Posterior predictive distribution

In machine learning we usually care more about uncertainty (and accuracy) of our predictions, not our parameter estimates. Fortunately, one can derive the posterior predictive distribution in closed form. In particular, one can show that, given N' new test inputs $\tilde{\mathbf{X}}$, we have

$$p(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}, \mathcal{D}) = \int \int p(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}, \mathbf{w}, \sigma^2) p(\mathbf{w}, \sigma^2 | \mathcal{D}) d\mathbf{w} d\sigma^2 \quad (15.64)$$

$$= \int \int \mathcal{N}(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}\mathbf{w}, \sigma^2 \mathbf{I}_{N'}) \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) d\mathbf{w} d\sigma^2 \quad (15.65)$$

$$= \mathcal{T}(\tilde{\mathbf{y}}|\tilde{\mathbf{X}} \hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}} (\mathbf{I}_{N'} + \tilde{\mathbf{X}} \hat{\Sigma} \tilde{\mathbf{X}}^\top), 2 \hat{a}) \quad (15.66)$$

The posterior predictive mean is equivalent to “normal” linear regression, but where we plug in $\hat{\mathbf{w}} = \mathbb{E}[\mathbf{w}|\mathcal{D}]$ instead of the MLE. The posterior predictive variance has two components: $\hat{b}/\hat{a}\mathbf{I}_{N'}$ 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 depending on how close the test inputs are to the training data. The results are similar to using a Gaussian prior (with fixed $\hat{\sigma}^2$), except the predictive distribution is even wider, since we are taking into account uncertainty about σ^2 .

15.2.3 Uninformative priors

A common criticism of Bayesian inference is the need to use a prior. This is sometimes thought to “pollute” the inferences one makes from the data. We can minimize the effect of the prior by using an uninformative prior, as we discussed in Section 3.5. Below we discuss various uninformative priors for linear regression.

15.2.3.1 Jeffreys prior

From Section 3.5.3.1, we know that the Jeffreys prior for the location parameter has the form $p(\mathbf{w}) \propto 1$, and from Section 3.5.3.2, we know that the Jeffreys prior for the scale factor has the form $p(\sigma) \propto \sigma^{-1}$. We can emulate these priors using an improper NIG prior with $\tilde{\mathbf{w}} = \mathbf{0}$, $\tilde{\Sigma} = \infty \mathbf{I}$, $\tilde{a} = -D/2$ and $\tilde{b} = 0$. The corresponding posterior is given by

$$p(\mathbf{w}, \sigma^2 | \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.67)$$

$$\hat{\mathbf{w}} = \hat{\mathbf{w}}_{\text{mle}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.68)$$

$$\hat{\Sigma} = (\mathbf{X}^\top \mathbf{X})^{-1} \triangleq \mathbf{C} \quad (15.69)$$

$$\hat{a} = \frac{\nu}{2} \quad (15.70)$$

$$\hat{b} = \frac{s^2 \nu}{2} \quad (15.71)$$

$$s^2 \triangleq \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|^2}{\nu} \quad (15.72)$$

$$\nu = N - D \quad (15.73)$$

Hence the posterior distribution of the weights is given by

$$p(\mathbf{w}|\mathcal{D}) = \mathcal{T}(\mathbf{w}|\hat{\mathbf{w}}, s^2 \mathbf{C}, \nu) \quad (15.74)$$

where $\hat{\mathbf{w}}$ is the MLE. The marginals for each weight therefore have the form

$$p(w_d|\mathcal{D}) = \mathcal{T}(w_d|\hat{w}_d, s^2 C_{dd}, \nu) \quad (15.75)$$

15.2.3.2 Connection to frequentist statistics

Interestingly, the posterior when using Jeffreys prior is formally equivalent to the **frequentist sampling distribution** of the MLE, which has the form

$$p(\hat{w}_d|\mathcal{D}^*) = \mathcal{T}(\hat{w}_d|w_d, s^2 C_{dd}, \nu) \quad (15.76)$$

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]):

$$\frac{\hat{w}_d - w_d}{s\sqrt{C_{dd}}} \sim t_{N-D} \quad (15.77)$$

The sampling distribution is numerically the same as the posterior distribution in Equation (15.75) 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.

15.2.3.3 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.

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 $\check{\Sigma} = g(\mathbf{X}^\top \mathbf{X})^{-1}$, where $g > 0$ plays a role analogous to $1/\lambda$ in ridge regression. This is called Zellner's **g-prior** [Zel86].¹ We see that the prior covariance is proportional to $(\mathbf{X}^\top \mathbf{X})^{-1}$ 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].

¹ 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.

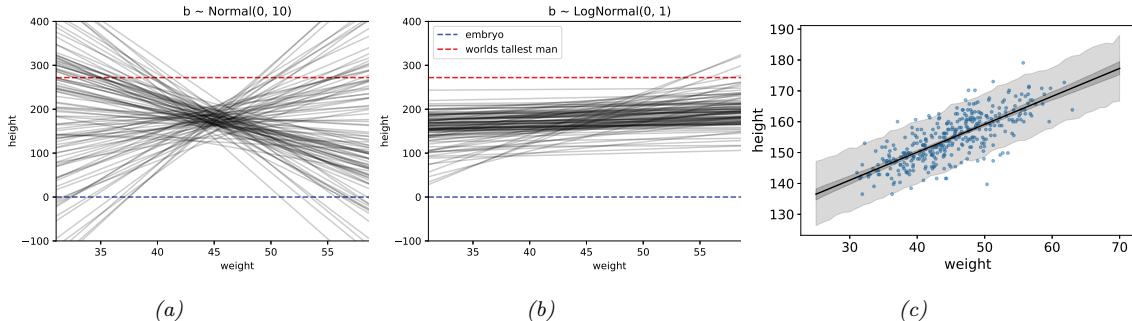


Figure 15.1: Linear regression for predicting height given weight, $y \sim \mathcal{N}(\alpha + \beta x, \sigma^2)$. (a) Prior predictive samples using a Gaussian prior for β . (b) Prior predictive samples using a log-Gaussian prior for β . (c) Posterior predictive samples using the log-Gaussian prior. 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.ipynb](#).

With this prior, the posterior becomes

$$p(\mathbf{w}, \sigma^2 | g, \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \mathbf{w}_N, \mathbf{V}_N, a_N, b_N) \quad (15.78)$$

$$\mathbf{V}_N = \frac{g}{g+1} (\mathbf{X}^T \mathbf{X})^{-1} \quad (15.79)$$

$$\mathbf{w}_N = \frac{g}{g+1} \hat{\mathbf{w}}_{mle} \quad (15.80)$$

$$a_N = N/2 \quad (15.81)$$

$$b_N = \frac{s^2}{2} + \frac{1}{2(g+1)} \hat{\mathbf{w}}_{mle}^T \mathbf{X}^T \mathbf{X} \hat{\mathbf{w}}_{mle} \quad (15.82)$$

Various approaches have been proposed for setting g , including cross validation, empirical Bayes [Min00a; GF00], hierarchical Bayes [Lia+08], etc.

15.2.4 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 use a dataset collected from Kalahari foragers by the anthropologist Nancy Howell (this example is from [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

$$y_i \sim \mathcal{N}(\mu_i, \sigma) \quad (15.83)$$

$$\mu_i = \alpha + \beta(x_i - \bar{x}) \quad (15.84)$$

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.

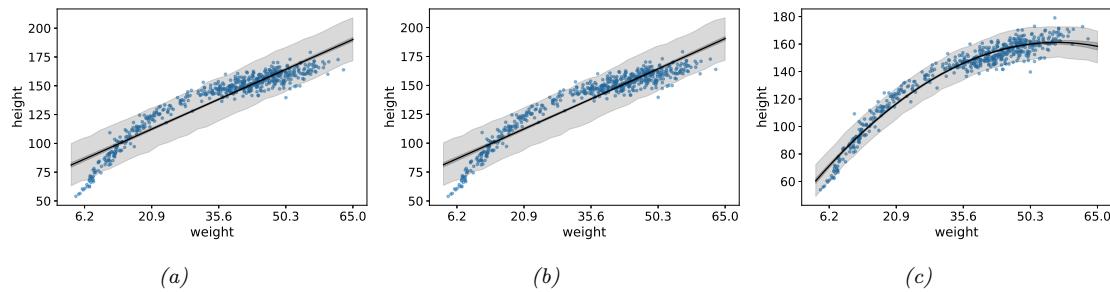


Figure 15.2: Linear regression for predicting height given weight for the full dataset (including children) using polynomial regression. (a) Posterior fit for linear model with log-Gaussian prior for β_1 . (b) Posterior fit for quadratic model with log-Gaussian prior for β_2 . (c) Posterior fit for quadratic model with Gaussian prior for β_2 . Adapted from Figure 4.11 of [McE20]. Generated by `linreg_height_weight.ipynb`.

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 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 :

$$\mu_i = \alpha + \beta_1 x_i + \beta_2 x_i^2 \quad (15.85)$$

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 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.

15.2.5 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 :

$$p(\mathbf{w}) = \prod_{d=1}^D (1 - \pi) \delta(w_d) + \pi \text{Unif}(w_d | -a, a) \quad (15.86)$$

where π is the prior probability that each coefficient is non-zero. The corresponding log prior on the coefficients is thus

$$\log p(\mathbf{w}) = \|\mathbf{w}\|_0 \log(1 - \pi) + (D - \|\mathbf{w}\|_0) \log \pi = -\lambda \|\mathbf{w}\|_0 + \text{const} \quad (15.87)$$

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 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,

$$p(\mathbf{w}) = \prod_d (1 - \pi) \delta(w_d) + \pi \mathcal{N}(w_d | 0, \sigma_w^2) \quad (15.88)$$

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”.

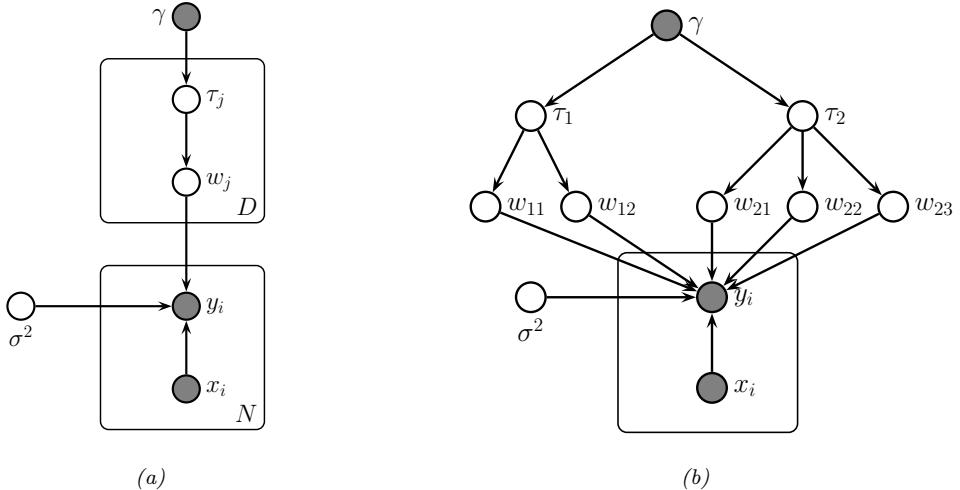


Figure 15.3: (a) Representing lasso using a Gaussian scale mixture prior. (b) Graphical model for group lasso with 2 groups, the first has size $G_1 = 2$, the second has size $G_2 = 3$.

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]).

15.2.6 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:

$$\text{PNLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) - \log p(\mathbf{w}|\lambda) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1 \quad (15.89)$$

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:

$$p(\mathbf{w}|\sigma^2) = \prod_j \frac{\lambda}{2\sqrt{\sigma^2}} e^{-\lambda|w_j|/\sqrt{\sigma^2}} \quad (15.90)$$

(Note that conditioning the prior on σ^2 is important to ensure that the full posterior is unimodal.)

To simplify inference, we will represent the Laplace prior as a Gaussian scale mixture, which we discussed in Section 28.2.3.2. In particular, one can show that the Laplace distribution is an infinite

weighted sum of Gaussians, where the precision comes from a gamma distribution:

$$\text{Laplace}(w|0, \lambda) = \int \mathcal{N}(w|0, \tau^2) \text{Ga}(\tau^2|1, \frac{\lambda^2}{2}) d\tau^2 \quad (15.91)$$

We can therefore represent the Bayesian lasso model as a hierarchical latent variable model, as shown in Figure 15.3a. The corresponding joint distribution has the following form:

$$p(\mathbf{y}, \mathbf{w}, \boldsymbol{\tau}, \sigma^2 | \mathbf{X}) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \left[\prod_j \mathcal{N}(w_j|0, \sigma^2 \tau_j^2) \text{Ga}(\tau_j^2|1, \lambda^2/2) \right] p(\sigma^2) \quad (15.92)$$

We can also create a GSM to match the **group lasso** prior, which sets multiple coefficients to zero at the same time:

$$\mathbf{w}_g | \sigma^2, \tau_g^2 \sim \mathcal{N}(\mathbf{0}, \sigma^2 \tau_g^2 \mathbf{I}_{d_g}) \quad (15.93)$$

$$\tau_g^2 \sim \text{Ga}(\frac{d_g + 1}{2}, \frac{\lambda^2}{2}) \quad (15.94)$$

where d_g is the size of group g . So we see that there is one variance term per group, each of which comes from a gamma prior, whose shape parameter depends on the group size, and whose rate parameter is controlled by γ .

Figure 15.3b gives an example, where we have 2 groups, one of size 2 and one of size 3. This picture makes it clearer why there should be a grouping effect. For example, suppose $w_{1,1}$ is small; then τ_1^2 will be estimated to be small, which will force $w_{1,2}$ to be small, due to shrinkage (cf. Section 3.6). 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 become large as well.

Given these hierarchical models, we can easily derive a Gibbs sampling algorithm (Section 12.3) to sample from the posterior (see e.g., [PC08]). Unfortunately, these posterior samples are not sparse, even though the MAP estimate is sparse. This is because the prior puts infinitesimal probability on the event that each coefficient is zero.

15.2.7 Horseshoe prior

The Laplace prior is not suitable for sparse Bayesian models, because posterior samples are not sparse. The spike and slab prior does not have this problem but is often too slow to use (although see [BRG20]). Fortunately, it is possible to devise continuous priors (without discrete latent variables) that are both sparse and computationally efficient. One popular prior of this type is the **horseshoe prior** [CPS10], so-named because of the shape of its density function.

In the horseshoe prior, instead of using a Laplace prior for each weight, we use the following Gaussian scale mixture:

$$w_j \sim \mathcal{N}(0, \lambda_j^2 \tau^2) \quad (15.95)$$

$$\lambda_j \sim \mathcal{C}_+(0, 1) \quad (15.96)$$

$$\tau^2 \sim \mathcal{C}_+(0, 1) \quad (15.97)$$

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, (1/\beta)\mathbf{I}_N) \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{A}^{-1}) d\mathbf{w} \quad (15.98)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top) \quad (15.99)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_{\boldsymbol{\alpha}}) \quad (15.100)$$

where $\mathbf{C}_{\boldsymbol{\alpha}} \triangleq \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top$. This is very similar to the marginal likelihood under the spike-and-slab prior (Section 15.2.5), which is given by

$$p(\mathbf{y}|\mathbf{X}, \mathbf{s}, \sigma_w^2, \sigma_y^2) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}_s \mathbf{w}_s, \sigma_y^2 \mathbf{I}) \mathcal{N}(\mathbf{w}_s|\mathbf{0}_s, \sigma_w^2 \mathbf{I}) d\mathbf{w}_s = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_s) \quad (15.101)$$

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

$$\ell(\boldsymbol{\alpha}, \beta) = -2 \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \log |\mathbf{C}_{\boldsymbol{\alpha}}| + \mathbf{y}^\top \mathbf{C}_{\boldsymbol{\alpha}}^{-1} \mathbf{y} \quad (15.102)$$

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$, some of which we discuss in Section 15.2.8.3.

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.

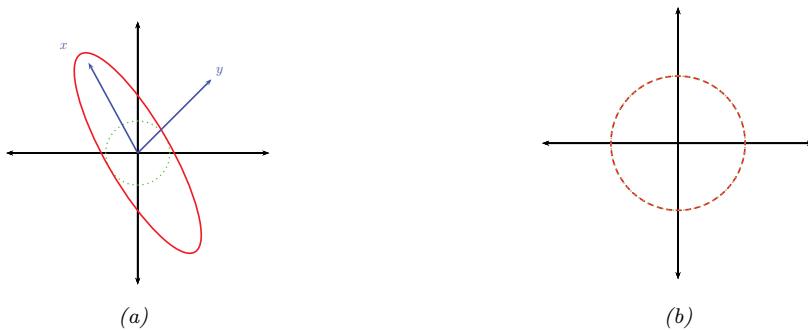


Figure 15.4: Illustration of why ARD results in sparsity. The vector of inputs \mathbf{x} does not point towards the vector of outputs \mathbf{y} , so the feature should be removed. (a) For finite α , the probability density is spread in directions away from \mathbf{y} . (b) When $\alpha = \infty$, the probability density at \mathbf{y} is maximized. Adapted from Figure 8 of [Tip01].

15.2.8.2 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 $p(\mathbf{w}|\mathcal{D}, \hat{\boldsymbol{\alpha}}, \hat{\beta}) = \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \hat{\boldsymbol{\Sigma}})$, where $\hat{\boldsymbol{\Sigma}}^{-1} = \hat{\beta}\mathbf{X}^T\mathbf{X} + \mathbf{A}$ and $\hat{\mathbf{w}} = \hat{\beta}\hat{\boldsymbol{\Sigma}}\mathbf{X}^T\mathbf{y}$. 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 10.3.3), with a factored posterior approximation of the form

$$q(\mathbf{w}, \boldsymbol{\alpha}) = q(\mathbf{w})q(\boldsymbol{\alpha}) \approx \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \prod_d \text{Ga}(\alpha_d | \hat{a}_d, \hat{b}_d) \quad (15.103)$$

ARD approximates $q(\boldsymbol{\alpha})$ by a point estimate. However, in VB, we integrate out $\boldsymbol{\alpha}$; the resulting

posterior marginal $q(\mathbf{w})$ on the weights is given by

$$p(\mathbf{w}|\mathcal{D}) = \int \mathcal{N}(\mathbf{w}|\mathbf{0}, \text{diag}(\boldsymbol{\alpha})^{-1}) \prod_d \text{Ga}(\alpha_d | \hat{a}_d, \hat{b}) d\boldsymbol{\alpha} \quad (15.104)$$

This is a Gaussian scale mixture, and can be shown to be the same as a multivariate Student distribution (see Section 28.2.3.1), with non-diagonal covariance. Note that the Student has a large spike at 0, which intuitively explains why the posterior mean (which, for a Student distribution, is equal to the posterior mode) is sparse.

Finally, we can also view ARD as a MAP estimation problem with a **non-factorial prior** [WN07]. Intuitively, the dependence between the w_j parameters arises, despite the use of a diagonal Gaussian prior, because the prior precision α_j is estimated based after marginalizing out all \mathbf{w} , and hence depends on all the features. Interestingly, [WRN10] prove that MAP estimation with non-factorial priors is strictly better than MAP estimation with any possible factorial prior in the following sense: the non-factorial objective always has fewer local minima than factorial objectives, while still satisfying the property that the global optimum of the non-factorial objective corresponds to the global optimum of the ℓ_0 objective — a property that ℓ_1 regularization, which has no local minima, does not enjoy.

15.2.8.3 Algorithms for ARD

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$. One approach is to use EM, in which we compute $p(\mathbf{w}|\mathcal{D}, \boldsymbol{\alpha})$ in the E step and then maximize $\boldsymbol{\alpha}$ in the M step. In variational Bayes, we infer both \mathbf{w} and $\boldsymbol{\alpha}$ (see [Dru08] for details). In [WN10], they present a method based on iteratively reweighted ℓ_1 estimation.

Recently, [HXW17] showed that the nested iterative computations performed these methods can be emulated by a recurrent neural network (Section 16.3.4). Furthermore, by training this model, it is possible to achieve much faster convergence than manually designed optimization algorithms.

15.2.8.4 Relevance vector machines

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 $\phi(\mathbf{x}) = [\mathcal{K}(\mathbf{x}, \mathbf{x}_1), \dots, \mathcal{K}(\mathbf{x}, \mathbf{x}_N)]$, where $\mathcal{K}()$ is a kernel function (Section 18.2) and $\mathbf{x}_1, \dots, \mathbf{x}_N$ are the N training points. This is called **kernel basis function expansion**, and transforms the input 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. However, we can use ARD to select a small subset of the exemplars. This technique is called the relevance vector machine (RVM) [Tip01; TF03].

15.2.9 Multivariate linear regression

This section is written by Xinglong Li.

In this section, we consider the **multivariate linear regression** model, which has the form

$$\mathbf{Y} = \mathbf{WX} + \mathbf{E} \quad (15.105)$$

where $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$ is the matrix of regression coefficient, $\mathbf{X} \in \mathbb{R}^{N_x \times N}$ is the matrix of input features (with each row being an input variable and each column being an observation), $\mathbf{Y} \in \mathbb{R}^{N_y \times N}$ is the

matrix of responses (with each row being an output variable and each *column* being an observation), and $\mathbf{E} = [\mathbf{e}_1, \dots, \mathbf{e}_N]$ is the matrix of residual errors, where $\mathbf{e}_i \stackrel{iid}{\sim} \mathcal{N}(\mathbf{0}, \Sigma)$. It can be seen from the definition that given Σ , \mathbf{W} and \mathbf{X} , columns of \mathbf{Y} are independently random variables following multivariate normal distributions. So the likelihood of the observation is

$$p(\mathbf{Y}|\mathbf{W}, \mathbf{X}, \Sigma) = \frac{1}{(2\pi)^{N_y \times N} |\Sigma|^{N/2}} \exp \left(\sum_{i=1}^N -\frac{1}{2} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i)^\top \Sigma^{-1} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i) \right) \quad (15.106)$$

$$= \frac{1}{(2\pi)^{N_y \times N} |\Sigma|^{N/2}} \exp \left(-\frac{1}{2} \text{tr} ((\mathbf{Y} - \mathbf{W}\mathbf{X})^\top \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X})) \right) \quad (15.107)$$

$$= \mathcal{MN}(\mathbf{Y}|\mathbf{W}\mathbf{X}, \Sigma, \mathbf{I}_{N \times N}), \quad (15.108)$$

The conjugate prior for this is the **matrix normal inverse Wishart** distribution,

$$\mathbf{W}, \Sigma \sim \text{MNIW}(\mathbf{M}_0, \mathbf{V}_0, \nu_0, \Psi_0) \quad (15.109)$$

where the MNIW is defined by

$$\mathbf{W}|\Sigma \sim \mathcal{MN}(\mathbf{M}_0, \Sigma_0, \mathbf{V}_0) \quad (15.110)$$

$$\Sigma \sim \text{IW}(\nu_0, \Psi_0), \quad (15.111)$$

where $\mathbf{V}_0 \in \mathbb{R}_{++}^{N_x \times N_x}$, $\Psi_0 \in \mathbb{R}_{++}^{N_y \times N_y}$ and $\nu_0 > N_x - 1$ is the degree of freedom of the inverse Wishart distribution.

The posterior distribution of $\{\mathbf{W}, \Sigma\}$ still follows a matrix normal inverse Wishart distribution. We follow the derivation in [Fox09, App.F]. The density of the joint distribution is

$$p(\mathbf{Y}, \mathbf{W}, \Sigma) \propto |\Sigma|^{-(\nu_0 + N_y + 1 + N_x + N)/2} \times \exp \left\{ -\frac{1}{2} \text{tr}(\Omega_0) \right\} \quad (15.112)$$

$$\Omega_0 \triangleq \Psi_0 \Sigma^{-1} + (\mathbf{Y} - \mathbf{W}\mathbf{X})^\top \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^\top \Sigma^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0 \quad (15.113)$$

We first aggregate items including \mathbf{W} in the exponent so that it takes the form of a matrix normal distribution. This is similar to the “completing the square” technique that we used in deriving the conjugate posterior for multivariate normal distributions in Section 3.4.4.3. Specifically,

$$\text{tr} [(\mathbf{Y} - \mathbf{W}\mathbf{X})^\top \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^\top \Sigma^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0] \quad (15.114)$$

$$= \text{tr} (\Sigma^{-1} [(\mathbf{Y} - \mathbf{W}\mathbf{X})(\mathbf{Y} - \mathbf{W}\mathbf{X})^\top + (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0 (\mathbf{W} - \mathbf{M}_0)^\top]) \quad (15.115)$$

$$= \text{tr} (\Sigma^{-1} [\mathbf{W}\mathbf{S}_{xx}\mathbf{W}^\top - 2\mathbf{S}_{yx}\mathbf{W}^\top + \mathbf{S}_{yy}]) \quad (15.116)$$

$$= \text{tr} (\Sigma^{-1} [(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})\mathbf{S}_{xx}(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})^\top + \mathbf{S}_{y|x}]). \quad (15.117)$$

where

$$\mathbf{S}_{xx} = \mathbf{XX}^\top + \mathbf{V}_0, \quad \mathbf{S}_{yx} = \mathbf{YX}^\top + \mathbf{M}_0 \mathbf{V}_0, \quad (15.118)$$

$$\mathbf{S}_{yy} = \mathbf{YY}^\top + \mathbf{M}_0 \mathbf{V}_0 \mathbf{M}_0^\top, \quad \mathbf{S}_{y|x} = \mathbf{S}_{yy} - \mathbf{S}_{yx} \mathbf{S}_{xx}^{-1} \mathbf{S}_{yx}^\top. \quad (15.119)$$

Therefore, it can be seen from Equation (15.117) that given Σ , \mathbf{W} follows a matrix normal distribution

$$\mathbf{W}|\Sigma, \mathbf{X}, \mathbf{Y} \sim \mathcal{MN}(\mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}, \Sigma, \mathbf{S}_{xx}). \quad (15.120)$$

Marginalizing out \mathbf{W} (which corresponds to removing the terms including \mathbf{W} in the exponent in Equation (15.113)), it can be shown that the posterior distribution of Σ is an inverse Wishart distribution. In fact, by replacing Equation (15.117) to the corresponding terms in Equation (15.113), it can be seen that the only terms left after integrating out \mathbf{W} are $\Sigma^{-1}\Psi$ and $\Sigma^{-1}\mathbf{S}_{y|x}$, which indicates that the scale matrix of the posterior inverse Wishart distribution is $\Psi_0 + \mathbf{S}_{y|x}$.

In conclusion, the joint posterior distribution of $\{\mathbf{W}, \Sigma\}$ given the observation is

$$\mathbf{W}, \Sigma | \mathbf{X}, \mathbf{Y} \sim \text{MNIW}(\mathbf{M}_1, \mathbf{V}_1, \nu_1, \Psi_1) \quad (15.121)$$

$$\mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \quad (15.122)$$

$$\mathbf{V}_1 = \mathbf{S}_{xx} \quad (15.123)$$

$$\nu_1 = N + \nu_0 \quad (15.124)$$

$$\Psi_1 = \Psi_0 + \mathbf{S}_{y|x} \quad (15.125)$$

The MAP estimate of \mathbf{W} and Σ are the mode of the posterior matrix normal inverse Wishart distribution. To derive this, notice that \mathbf{W} only appears in the matrix normal density function in the posterior, so the matrix \mathbf{W} maximizing the posterior density of $\{\mathbf{W}, \Sigma\}$ is the matrix $\hat{\mathbf{W}}$ that maximizes the matrix normal posterior of \mathbf{W} . So the MAP estimate of \mathbf{W} is $\hat{\mathbf{W}} = \mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}$, and this holds for any value of Σ . By plugging $\mathbf{W} = \hat{\mathbf{W}}$ into the joint posterior of $\{\mathbf{W}, \Sigma\}$, and taking derivatives over Σ , it can be seen that the matrix maximizing the density is $(\nu_1 + N_y + N_x + 1)^{-1}\Psi_1$. Since Ψ_1 is positive definite, it is the MAP estimate of Σ .

In conclusion, the MAP estimate of $\{\mathbf{W}, \Sigma\}$ are

$$\hat{\mathbf{W}} = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \quad (15.126)$$

$$\hat{\Sigma} = \frac{1}{\nu_1 + N_y + N_x + 1} (\Psi_0 + \mathbf{S}_{y|x}) \quad (15.127)$$

15.3 Logistic regression

Logistic regression is a very widely used discriminative classification model that maps input 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 **binary logistic regression**, and if $C > 2$, it is known as **multinomial logistic regression**, or alternatively, **multiclass logistic regression**.

15.3.1 Binary logistic regression

In the binary case, where $y \in \{0, 1\}$, the model has the following form

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Ber}(y|\sigma(\mathbf{w}^\top \mathbf{x} + b)) \quad (15.128)$$

where \mathbf{w} are the weights, b is the bias (offset), and σ is the **sigmoid** or **logistic** function, defined by

$$\sigma(a) \triangleq \frac{1}{1 + e^{-a}} \quad (15.129)$$

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 the output. Then we can write the log likelihood as the negative cross entropy:

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^N \mu_n^{y_n} (1 - \mu_n)^{1-y_n} = \sum_{n=1}^N y_n \log \mu_n + (1 - y_n) \log(1 - \mu_n) \quad (15.130)$$

We can expand this equation into a more explicit form (that is commonly seen in implementations) by performing some simple algebra. First note that

$$\mu_n = \frac{1}{1 + e^{-\eta_n}} = \frac{e^{\eta_n}}{1 + e^{\eta_n}}, \quad 1 - \mu_n = 1 - \frac{e^{\eta_n}}{1 + e^{\eta_n}} = \frac{1}{1 + e^{\eta_n}} \quad (15.131)$$

Hence

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{n=1}^N y_n [\log e^{\eta_n} - \log(1 + e^{\eta_n})] + (1 - y_n) [\log 1 - \log(1 + e^{\eta_n})] \quad (15.132)$$

$$= \sum_{n=1}^N y_n [\eta_n - \log(1 + e^{\eta_n})] + (1 - y_n) [-\log(1 + e^{\eta_n})] \quad (15.133)$$

$$= \sum_{n=1}^N y_n \eta_n - \sum_{n=1}^N \log(1 + e^{\eta_n}) \quad (15.134)$$

Note that the $\log(1 + e^a)$ function is often implemented using `np.log1p(np.exp(a))`.

15.3.2 Multinomial logistic regression

Multinomial logistic regression is a discriminative classification model of the following form:

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})) \quad (15.135)$$

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} is C -dimensional bias vector, and $\text{softmax}()$ is the **softmax function**, defined as

$$\text{softmax}(\mathbf{a}) \triangleq \left[\frac{e^{a_1}}{\sum_{c'=1}^C e^{a_{c'}}}, \dots, \frac{e^{a_C}}{\sum_{c'=1}^C e^{a_{c'}}} \right] \quad (15.136)$$

If we define the logits as $\boldsymbol{\eta}_n = \mathbf{W}\mathbf{x}_n + \mathbf{b}$, the probabilities as $\boldsymbol{\mu}_n = \text{softmax}(\boldsymbol{\eta}_n)$, 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:

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^N \prod_{c=1}^C \mu_{nc}^{y_{nc}} = \sum_{n=1}^N \sum_{c=1}^C y_{nc} \log \mu_{nc} \quad (15.137)$$

15.3.3 Dealing with class imbalance and the long tail

In many problems, some classes are much rarer than others; this problem is called **class imbalance**. In such a setting, standard maximum likelihood training may not work well, since it is designed to minimize (a bound on) the 0-1 loss, which can be dominated by the most frequent classes. A natural alternative is to consider the **balanced error rate**, which computes the average of the per-class error rates of classifier f :

$$\text{BER}(f) = \frac{1}{C} \sum_{y=1}^C p_{\mathbf{x}|y}^* \left(y \notin \operatorname{argmax}_{y' \in \mathcal{Y}} f_{y'}(\mathbf{x}) \right) \quad (15.138)$$

where p^* is the true distribution. The classifier that optimizes this loss, f^* , must satisfy

$$\operatorname{argmax}_{y \in \mathcal{Y}} f_y^*(\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} p_{\text{bal}}^*(y|\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} p^*(\mathbf{x}|y) \quad (15.139)$$

where $p_{\text{bal}}^*(y|\mathbf{x}) \propto \frac{1}{C} p^*(\mathbf{x}|y)$ is the predictor when using balanced classes. Thus to minimize the BER, we should use the class-conditional likelihood, $p(\mathbf{x}|y)$, rather than the class posterior, $p(y|\mathbf{x})$.

In [Men+21], they propose a simple scheme called **logit adjustment** that can achieve this optimal classifier. We assume the model computes logits using $f_y(\mathbf{x}) = \mathbf{w}_y^\top \phi(\mathbf{x})$, where $\phi(\mathbf{x}) = \mathbf{x}$ for a GLM. In the post-hoc version, the model is trained in the usual way, and then at prediction time, we use

$$\operatorname{argmax}_{y \in \mathcal{Y}} p(\mathbf{x}|y) = \operatorname{argmax}_{y \in \mathcal{Y}} \frac{p(y|\mathbf{x})p(\mathbf{x})}{p(y)} = \operatorname{argmax}_{y \in \mathcal{Y}} \frac{\exp(\mathbf{w}_y^\top \phi(\mathbf{x}))}{\pi_y} \quad (15.140)$$

where $\pi_y = p(y)$ is the empirical label prior. In practice, it is helpful to introduce a tuning parameter $\tau > 0$ and to use the predictor

$$\hat{f}(\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} f_y(\mathbf{x}) - \tau \log \pi_y \quad (15.141)$$

Alternatively, we can change the loss function used during training, by using the following **logit adjusted softmax cross-entropy loss**:

$$\ell(y, f(\mathbf{x})) = -\log \frac{e^{f_y(\mathbf{x}) + \tau \log \pi_y}}{\sum_{y'=1}^C e^{f_{y'}(\mathbf{x}) + \tau \log \pi_{y'}}} \quad (15.142)$$

This is like training with a predictor of the form $g_y(\mathbf{x}) = f_y(\mathbf{x}) + \tau \log \pi_y$, and then at test time using $\operatorname{argmax}_y f_y(\mathbf{x}) = \operatorname{argmax}_y g_y(\mathbf{x}) - \tau \log \pi_y$, as above.

We can also combine the above loss with a prior on the parameters and perform Bayesian inference, as we discuss below. (The use a non-standard likelihood can be justified using the generalized Bayesian inference framework, as discussed in Section 14.1.3.)

15.3.4 Parameter 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

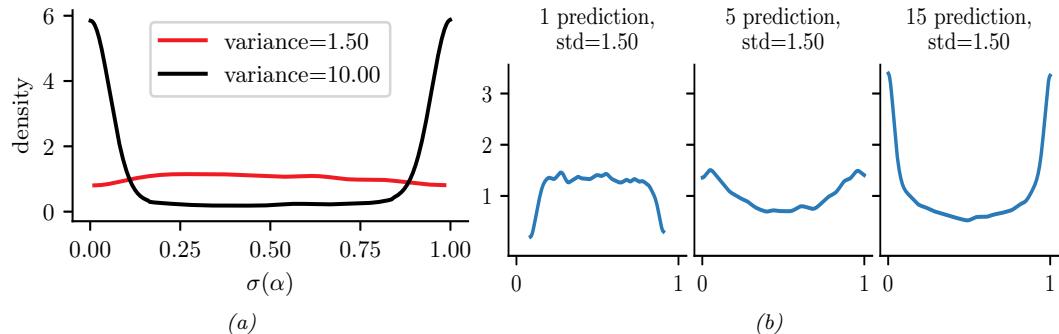


Figure 15.5: (a) Prior on logistic regression output when using $\mathcal{N}(0, \omega)$ prior for the offset term, for $\omega = 10$ or $\omega = 1.5$. Adapted from Figure 11.3 of [McE20]. Generated by `logreg_prior_offset.ipynb`. (b) Distribution over the fraction of 1s we expect to see when using 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.ipynb`.

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:

$$p(y|\boldsymbol{\theta}) = \text{Ber}(y|\sigma(\alpha)) \quad (15.143)$$

$$p(\alpha) = \mathcal{N}(\alpha|0, \omega) \quad (15.144)$$

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 Figure 15.5b. We see that the induced prior is initially flat, but eventually becomes skewed towards the extreme values of 0 and 1. To avoid this, we should standardize the inputs, and scale the variance of the prior by $1/\sqrt{D}$. We can also use a heavier tailed distribution, such as a Cauchy or Student [Gel+08; GLM15], instead of the Gaussian prior.

15.3.5 Laplace approximation to the posterior

Unfortunately, we cannot compute the posterior analytically, unlike with linear regression, since there is no corresponding conjugate prior. (This mirrors the case with MLE, where we have a closed form solution for linear regression, but not for logistic regression.) Fortunately, there are a range of approximate inference methods we can use.

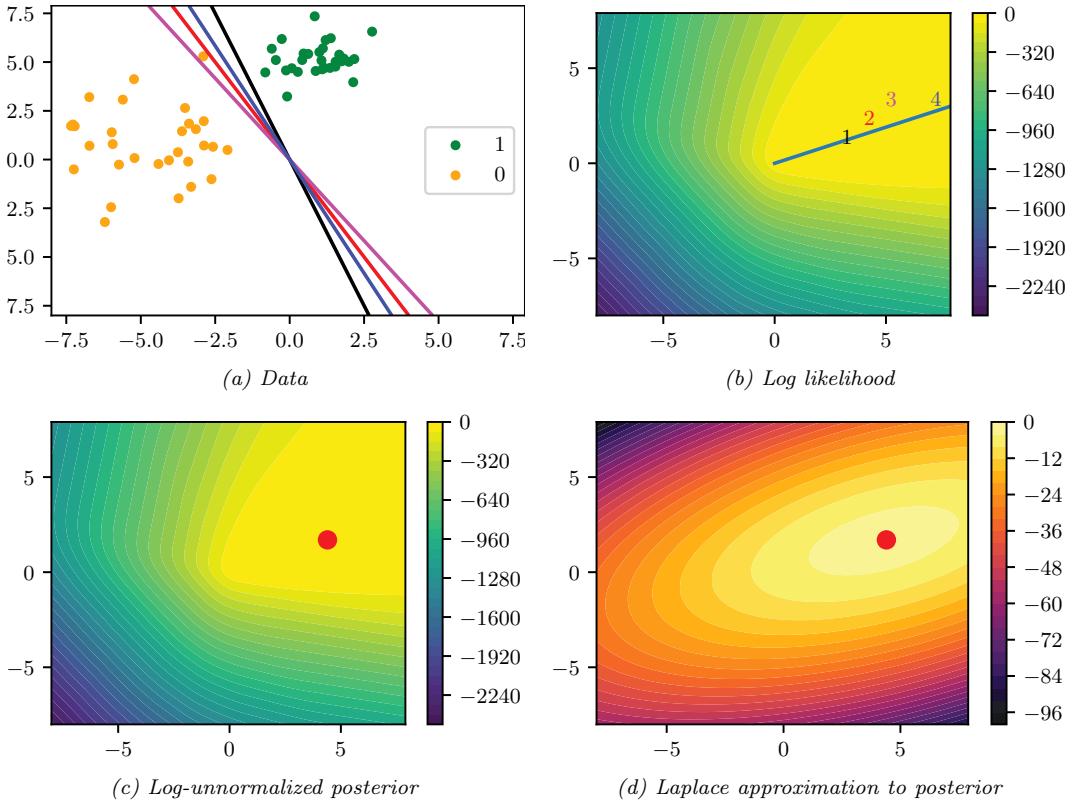


Figure 15.6: (a) Illustration of the data and some decision boundaries. (b) Log-likelihood for a logistic regression model. The line is drawn from the origin in the direction of the MLE (which is at infinity). The numbers correspond to 4 points in parameter space, corresponding to the colored lines in (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.ipynb](#).

In this section, we use the Laplace approximation. As we explain in Section 7.4.3, this approximates the posterior using a Gaussian. The mean of the Gaussian is equal to the MAP estimate $\hat{\mathbf{w}}$, and the covariance is equal to the inverse Hessian \mathbf{H} computed at the MAP estimate, i.e.,

$$p(\mathbf{w}|\mathcal{D}) \approx \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \mathbf{H}^{-1}), \hat{\mathbf{w}} = \arg \min -\log p(\mathbf{w}, \mathcal{D}), \mathbf{H} = -\nabla_{\mathbf{w}}^2 \log p(\mathbf{w}, \mathcal{D})|_{\hat{\mathbf{w}}} \quad (15.145)$$

We can find the mode using a standard optimization method, and we can then compute the Hessian at the mode analytically or using automatic differentiation.

As an example, consider the binary data illustrated in Figure 15.6(a). There are many parameter settings that correspond to lines that perfectly separate the training data; we show 4 example lines. For each decision boundary in Figure 15.6(a), we plot the corresponding parameter vector as point in the log likelihood surface in Figure 15.6(b). These parameters values are $\mathbf{w}_1 = (3, 1)$, $\mathbf{w}_2 = (4, 2)$, $\mathbf{w}_3 = (5, 3)$, and $\mathbf{w}_4 = (7, 3)$. These points all approximately satisfy $\mathbf{w}_i(1)/\mathbf{w}_i(2) \approx \hat{\mathbf{w}}_{\text{mle}}(1)/\hat{\mathbf{w}}_{\text{mle}}(2)$,

and hence are close to the orientation of the maximum likelihood decision boundary. The points are ordered by increasing weight norm (3.16, 4.47, 5.83, and 7.62). The unconstrained MLE has $\|\mathbf{w}\| = \infty$, so is infinitely far to the top right.

To ensure a unique solution, we use a (spherical) Gaussian prior centered at the origin, $\mathcal{N}(\mathbf{w} | \mathbf{0}, \sigma^2 \mathbf{I})$. The value of σ^2 controls the strength of the prior. If we set $\sigma^2 = \infty$, we force the MAP estimate to be $\mathbf{w} = \mathbf{0}$; this will result in maximally uncertain predictions, since all points \mathbf{x} will produce a predictive distribution of the form $p(y = 1 | \mathbf{x}) = 0.5$. If we set $\sigma^2 = 0$, the MAP estimate becomes the MLE, resulting in minimally uncertain predictions. (In particular, all positively labeled points 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 data is separable.) As a compromise (to make a nice illustration), we pick the value $\sigma^2 = 100$.

Multiplying this prior by the likelihood results in the unnormalized posterior shown in Figure 15.6(c). The MAP estimate is shown by the blue dot. The Laplace approximation to this posterior is shown in Figure 15.6(d). We see that it gets the mode correct (by construction), but the shape of the posterior is somewhat distorted. (The southwest-northeast orientation captures uncertainty about the magnitude of \mathbf{w} , and the southeast-northwest orientation captures uncertainty about the orientation of the decision boundary.)

15.3.6 Approximating the posterior predictive distribution

Next we need to convert the posterior over the parameters into a posterior over predictions, as follows:

$$p(y | \mathbf{x}, \mathcal{D}) = \int p(y | \mathbf{x}, \mathbf{w}) p(\mathbf{w} | \mathcal{D}) d\mathbf{w} \quad (15.146)$$

The simplest way to evaluate this integral is to use a Monte Carlo approximation. For example, in the case of binary logistic regression, we have

$$p(y = 1 | \mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \sigma(\mathbf{w}_s^\top \mathbf{x}) \quad (15.147)$$

where $\mathbf{w}_s \sim p(\mathbf{w} | \mathcal{D})$ are posterior samples.

However, we can also use deterministic approximations to the integral, which are often faster. Let $\mathbf{f}_* = f(\mathbf{x}_*, \mathbf{w})$ be the predicted logits, before the sigmoid/softmax layer, given test point \mathbf{x}_* . If the posterior over the parameters is Gaussian, $p(\mathbf{w} | \mathcal{D}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then the predictive distribution over logits is also Gaussian:

$$p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}) = \int \delta(\mathbf{f}_* - f(\mathbf{x}_*, \mathbf{w})) \mathcal{N}(\mathbf{w} | \mathcal{D}) d\mathbf{w} = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*^\top \mathbf{x}_*, \mathbf{x}_*^\top \boldsymbol{\Sigma} \mathbf{x}_*) \triangleq \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \quad (15.148)$$

In the case of binary logistic regression, we can approximate the sigmoid with the probit function Φ (see Section 15.4), which allows us to solve the integral analytically:

$$p(y_* | \mathbf{x}_*) \approx \int \Phi(f_*) \mathcal{N}(f_* | \boldsymbol{\mu}_*, \sigma_*^2) df_* = \sigma \left(\frac{\boldsymbol{\mu}_*}{\sqrt{1 + \frac{\pi}{8} \sigma_*^2}} \right) \quad (15.149)$$

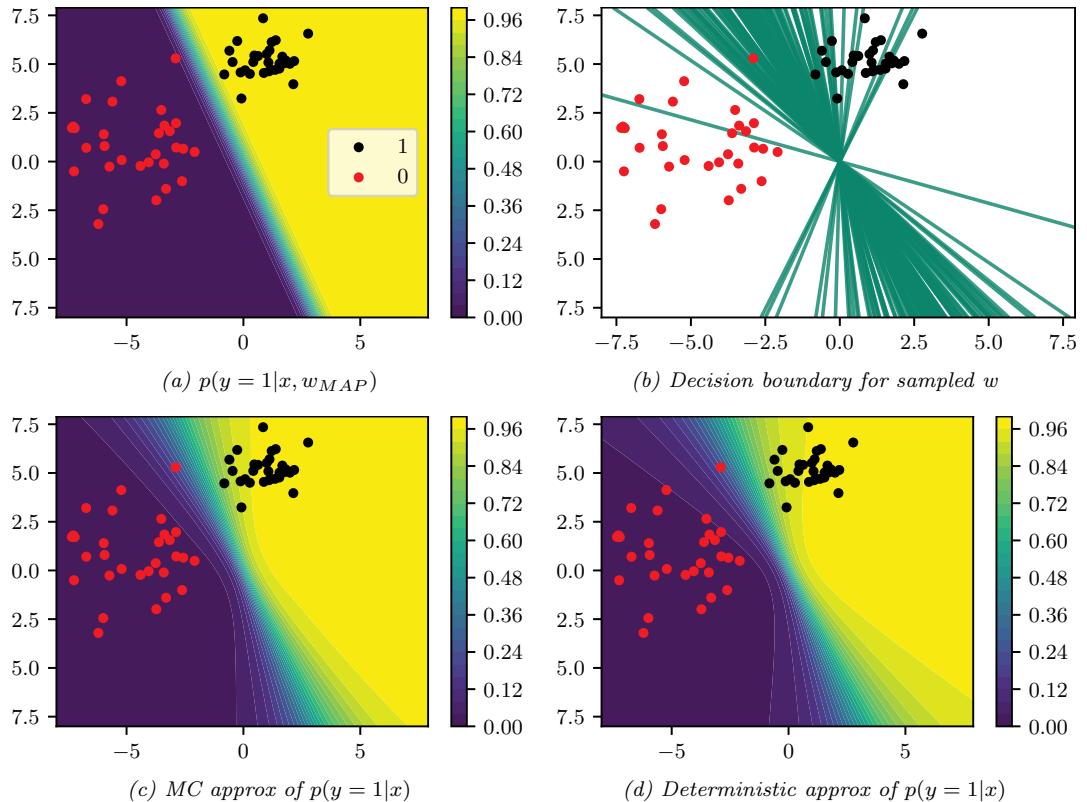


Figure 15.7: Posterior predictive distribution for a logistic regression model in 2d. (a) Contours of $p(y = 1|x, \hat{w}_{MAP})$. (b) Samples from the posterior predictive distribution. (c) Averaging over these samples. (d) Moderated output (probit approximation). Generated by [logreg_laplace_demo.ipynb](#).

This is called the **probit approximation** [SL90]. In [Gib97], a generalization to the multiclass case was provided. This is known as the **generalized probit approximation**, and has the form

$$p(\mathbf{y}_*|\mathbf{x}_*) \approx \int \text{softmax}(\mathbf{f}_*) \mathcal{N}(\mathbf{f}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) d\mathbf{f}_* = \text{softmax} \left(\left\{ \frac{\mu_{*,c}}{\sqrt{1 + \frac{\pi}{8} \Sigma_{*,cc}}} \right\} \right) \quad (15.150)$$

This ignores the correlations between the logits, because it only depends on the diagonal elements of $\boldsymbol{\Sigma}_*$. Nevertheless it can work well, even in the case of neural net classifiers [LIS20]. Another deterministic approximation, known as the **Laplace bridge**, is discussed in Section 17.3.10.2.

We now illustrate the posterior predictive for our binary example. Figure 15.7(a) shows the plugin approximation using the MAP estimate. We see that there is no uncertainty about the location of the decision boundary, even though we are generating probabilistic predictions over the labels. Figure 15.7(b) shows what happens when we plug in samples from the Gaussian posterior. Now we see that there is 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

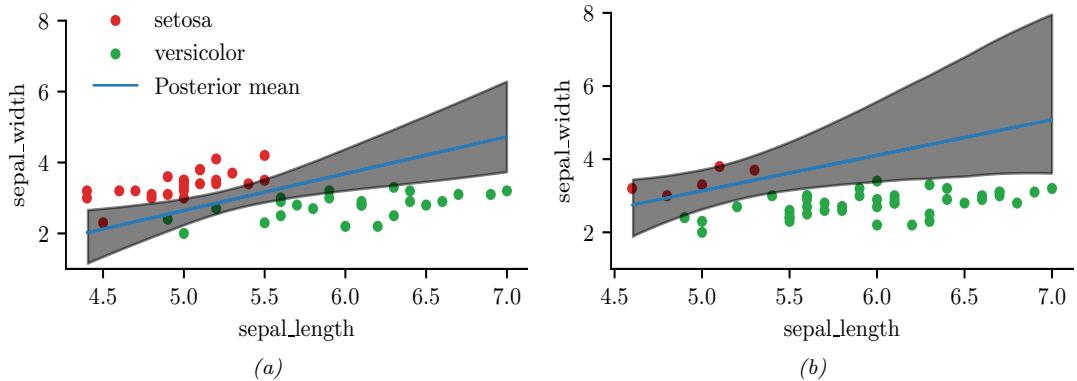


Figure 15.8: Illustration of the posterior over the decision boundary for classifying iris flowers (setosa vs versicolor) using 2 input features. (a) 25 examples per class. Adapted from Figure 4.5 of [Mar18]. (b) 5 examples of class 0, 45 examples of class 1. Adapted from Figure 4.8 of [Mar18]. Generated by `logreg_iris_bayes_2d.ipynb`.

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.

15.3.7 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. It is explained in depth in Chapter 12, but the output is a set of (correlated) samples from the posterior, which gives the following non-parametric approximation:

$$q(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s) \quad (15.151)$$

where $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$. Once we have the samples, we can plug them into Equation (15.147) to approximate the posterior predictive distribution.

A common MCMC method is known as Hamiltonian Monte Carlo (Section 12.5); this can leverage our ability to compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\mathcal{D}, \boldsymbol{\theta})$, for improved efficiency. 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:

$$x_2^* = -\frac{b}{w_2} + \left(-\frac{w_1}{w_2} x_1^* \right) \quad (15.152)$$

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,

Dept. D_i	Gender G_i	# Admitted A_i	# Rejected R_i	# Applications N_i
A	male	512	313	825
A	female	89	19	108
B	male	353	207	560
B	female	17	8	25
C	male	120	205	325
C	female	202	391	593
D	male	138	279	417
D	female	131	244	375
E	male	53	138	191
E	female	94	299	393
F	male	22	351	373
F	female	24	317	341

Table 15.1: Admissions data for UC Berkeley from [BHO75].

to get $p(x_1^*, x_2^* | \mathcal{D})$. The results of this method (using a vague Gaussian prior for the parameters) are shown in Figure 15.8a. The solid line is the posterior mean, and the shaded interval is a 95% credible interval. As before, we see that the uncertainty about the location of the boundary is higher as we move away from the training data.

In Figure 15.8b, we show what happens to the decision boundary when we have unbalanced classes. We notice two things. First, the posterior uncertainty increases, because we have less data from the blue class. Second, we see that the posterior mean of the decision boundary shifts towards the class with less data. This follows from linear discriminant analysis, where one can show that changing the class prior changes the location of the decision boundary, so that more of the input space gets mapped to the class which is higher a priori. (See [Mur22, Sec 9.2] for details.)

15.3.8 Other approximate inference methods

There are many other approximate inference methods we can use, as we discuss in Part II. A common approach is variational inference (Section 10.1), which converts approximate inference into an optimization problem. It does this by choosing an approximate distribution $q(\mathbf{w}; \psi)$ and optimizing the variational parameters ψ to maximize the evidence lower bound (ELBO). This has the effect of making $q(\mathbf{w}; \psi) \approx p(\mathbf{w} | \mathcal{D})$ in the sense that the KL divergence is small. There are several ways to tackle this: use a stochastic estimate of the ELBO (see Section 10.2.1), use the conditionally conjugate VI method of Supplementary Section 10.3.1.2, or use a “local” VI method that creates a quadratic lower bound to the logistic function (see Supplementary Section 15.1).

In the online setting, we can use assumed density filtering (ADF) to recursively compute a Gaussian approximate posterior $p(\mathbf{w} | \mathcal{D}_{1:t})$, as we discuss in Section 8.6.3.

15.3.9 Case study: is Berkeley admissions biased against women?

In this section, we consider a simple but interesting example of logistic regression from [McE20, Sec 11.1.4]. The question of interest is whether admission to graduate school at UC Berkeley is biased against women. The dataset comes from a famous paper [BHO75], which collected statistics for 6 departments for men and women. The data table only has 12 rows, shown in Table 15.1, although the total sample size (number of observations) is 4526. We conduct a regression analysis to try to determine if gender “causes” imbalanced admissions rates.

An obvious way to attempt to answer the question of interest is to fit a binomial logistic regression model, in which the outcome is the admissions rate, and the input is a binary variable representing the gender of each sample (male or female). One way to write this model is as follows:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.153)$$

$$\text{logit}(\mu_i) = \alpha + \beta \text{MALE}[i] \quad (15.154)$$

$$\alpha \sim \mathcal{N}(0, 10) \quad (15.155)$$

$$\beta \sim \mathcal{N}(0, 1.5) \quad (15.156)$$

Here A_i is the number of admissions for sample i , N_i is the number of applications, and $\text{MALE}[i] = 1$ iff the sample is male. So the log odds is α for female cases, and $\alpha + \beta$ for male candidates. (The choice of prior for these parameters is discussed in Section 15.3.4.)

The above formulation is asymmetric in the genders. In particular, the log odds for males has two random variables associated with it, and hence is a priori more uncertain. It is often better to rewrite the model in the following symmetric way:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.157)$$

$$\text{logit}(\mu_i) = \alpha_{\text{GENDER}[i]} \quad (15.158)$$

$$\alpha_j \sim \mathcal{N}(0, 1.5), j \in \{1, 2\} \quad (15.159)$$

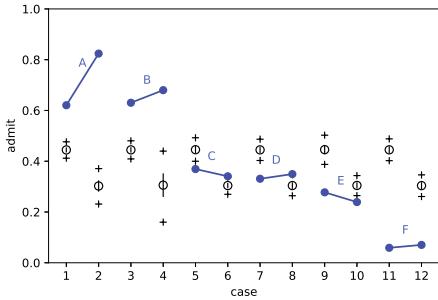
Here $\text{GENDER}[i]$ is the gender (1 for male, 2 for female), so the log odds is α_1 for males and α_2 for females.

We can perform posterior inference using a variety of methods (see Chapter 7). Here we use HMC (Section 12.5). We find the 89% credible interval for α_1 is $[-0.29, 0.16]$ and for α_2 is $[-0.91, 0.75]$.² The corresponding distribution for the difference in probability, $\sigma(\alpha_1) - \sigma(\alpha_2)$, is $[0.12, 0.16]$, with a mean of 0.14. So it seems that Berkeley is biased in favor of men.

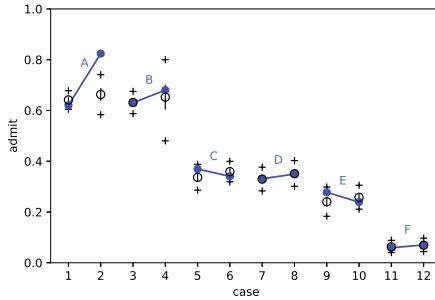
However, before jumping to conclusions, we should check if the model is any good. In Figure 15.9a, we plot the posterior predictive distribution, along with the original data. We see the model is a very bad fit to the data (the blue data dots are often outside the black predictive intervals). In particular, we see that the empirical admissions rate for women is actually higher in all the departments except for C and E, yet the model says that women should have a 14% lower chance of admission.

The trouble is that men and women did not apply to the same departments in equal amounts. Women tended not to apply to departments, like A and B, with high admissions rates, but instead applied more to departments, like F, with low admissions rates. So even though less women were accepted *overall*, within in each department, women tended to be accepted at about the same rate.

². McElreath uses 89% interval instead of 95% to emphasize the arbitrary nature of these values. The difference is insignificant.



(a)



(b)

Figure 15.9: Blue dots are admission rates for each of the 6 departments (A-F) for males (left half of each dyad) and females (right half). The circle is the posterior mean of μ_i , the small vertical black lines indicate 1 standard deviation of μ_i . The + marks indicate 95% predictive interval for A_i . (a) Basic model, only taking gender into account. (b) Augmented model, adding department specific offsets. Adapted from Figure 11.5 of [McE20]. Generated by [logreg_ucb_admissions_numpyro.ipynb](#).

We can get a better understanding if we consider the DAG in Figure 15.10a. This is intended to be a causal model of the relevant factors. We discuss causality in more detail in Chapter 36, but the basic idea should be clear from this picture. In particular, we see that there is an indirect causal path $G \rightarrow D \rightarrow A$ from gender to acceptance, so to infer the direct affect $G \rightarrow A$, we need to condition on D and close the indirect path. We can do this by adding department id as another feature:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.160)$$

$$\text{logit}(\mu_i) = \alpha_{\text{GENDER}[i]} + \gamma_{\text{DEPT}[i]} \quad (15.161)$$

$$\alpha_j \sim \mathcal{N}(0, 1.5), j \in \{1, 2\} \quad (15.162)$$

$$\gamma_k \sim \mathcal{N}(0, 1.5), k \in \{1, \dots, 6\} \quad (15.163)$$

Here $j \in \{1, 2\}$ (for gender) and $k \in \{1, \dots, 6\}$ (for department). Note that there 12 parameters in this model, but each combination (slice of the data) has a fairly large sample size of data associated with it, as we see in Table 15.1.

In Figure 15.9b, we plot the posterior predictive distribution for this new model; we see the fit is now much better. We find the 89% credible interval for α_1 is $[-1.38, 0.35]$ and for α_2 is $[-1.31, 0.42]$. The corresponding distribution for the difference in probability, $\sigma(\alpha_1) - \sigma(\alpha_2)$, is $[-0.05, 0.01]$. So it seems that there is no bias after all.

However, the above conclusion is based on the correctness of the model in Figure 15.10a. What if there are **unobserved confounders** U , such as academic ability, influencing both admission rate and department choice? This hypothesis is shown in Figure 15.10b. In this case, conditioning on the collider D opens up a non-causal path between gender and admissions, $G \rightarrow D \leftarrow U \rightarrow A$. This invalidates any causal conclusions we may want to draw.

The point of this example is to serve as a cautionary tale to those trying to draw causal conclusions from predictive models. See Chapter 36 for more details.



Figure 15.10: Some possible causal models of admissions rates. G is gender, D is department, A is acceptance rate. (a) No hidden confounders. (b) Hidden confounder (small dot) affects both D and A . Generated by `logreg_uci_admissions_numpyro.ipynb`.

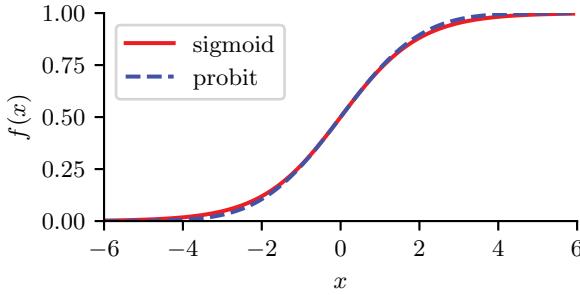


Figure 15.11: The logistic (sigmoid) function $\sigma(x)$ in solid red, with the Gaussian cdf function $\Phi(\lambda x)$ in dotted blue superimposed. Here $\lambda = \sqrt{\pi/8}$, which was chosen so that the derivatives of the two curves match at $x = 0$. Adapted from Figure 4.9 of [Bis06]. Generated by `probit_plot.ipynb`.

15.4 Probit regression

In this section, we discuss **probit regression**, which is similar to binary logistic regression except 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 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 inverse of the Gaussian cdf is known as the **probit function**.

The Gaussian cdf Φ is very similar to the logistic function, as shown in Figure 15.11. Thus probit regression and “regular” logistic regression behave very similarly. However, probit regression has some advantages. In particular, it has a simple interpretation as a latent variable model (see Section 15.4.1), which arises from the field of **choice theory** as studied in economics (see e.g., [Koo03]). This also simplifies the task of Bayesian parameter inference.

15.4.1 Latent variable interpretation

We can interpret $a_n = \mathbf{w}^\top \mathbf{x}_n$ as a factor that is proportional to how likely a person is respond positively (generate $y_n = 1$) given input \mathbf{x}_n . However, typically there are other unobserved factors that

influence someone's response. Let us model these hidden factors by Gaussian noise, $\epsilon_n \sim \mathcal{N}(0, 1)$. Let the combined preference for positive outcomes be represented by the latent variable $z_n = \mathbf{w}^\top \mathbf{x}_n + \epsilon_n$. We assume that the person will pick the positive label iff this latent factor is positive rather than negative, i.e.,

$$y_n = \mathbb{I}(z_n \geq 0) \quad (15.164)$$

When we marginalize out z_n , we recover the probit model:

$$p(y_n = 1 | \mathbf{x}_n, \mathbf{w}) = \int \mathbb{I}(z_n \geq 0) \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) dz_n \quad (15.165)$$

$$= p(\mathbf{w}^\top \mathbf{x}_n + \epsilon_n \geq 0) = p(\epsilon_n \geq -\mathbf{w}^\top \mathbf{x}_n) \quad (15.166)$$

$$= 1 - \Phi(-\mathbf{w}^\top \mathbf{x}_n) = \Phi(\mathbf{w}^\top \mathbf{x}_n) \quad (15.167)$$

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:

$$f(y|\mu, s) \triangleq \frac{e^{-\frac{y-\mu}{s}}}{s(1+e^{-\frac{y-\mu}{s}})^2} = \frac{1}{4s} \operatorname{sech}^2\left(\frac{y-\mu}{s}\right) \quad (15.168)$$

where the mean is μ and the variance is $\frac{s^2 \pi^2}{3}$. The cdf of this distribution is given by

$$F(y|\mu, s) = \frac{1}{1+e^{-\frac{y-\mu}{s}}} \quad (15.169)$$

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

$$\mathbf{g}_n \triangleq \frac{d}{d\mathbf{w}} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \frac{d\mu_n}{d\mathbf{w}} \frac{d}{d\mu_n} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \mathbf{x}_n \frac{\tilde{y}_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \quad (15.170)$$

where ϕ is the standard normal pdf, and Φ is its cdf. Similarly, the Hessian for a single case is given by

$$\mathbf{H}_n = \frac{d}{d\mathbf{w}^2} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = -\mathbf{x}_n \left(\frac{\phi(\mu_n)^2}{\Phi(\tilde{y}_n \mu_n)^2} + \frac{\tilde{y}_n \mu_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \right) \mathbf{x}_n^\top \quad (15.171)$$

This can be passed to any gradient-based optimizer.

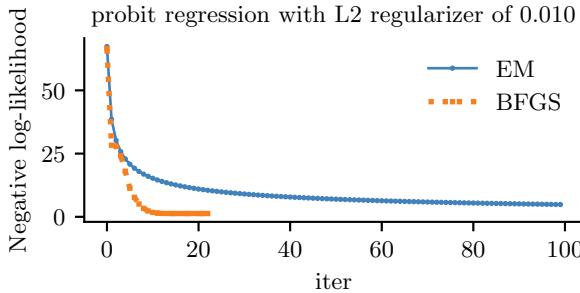


Figure 15.12: Fitting a probit regression model in 2d using a quasi-Newton method or EM. Generated by `probit_reg_demo.ipynb`.

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} :

$$\ell(\mathbf{z}, \mathbf{w} | \mathbf{V}_0) = \log p(\mathbf{y} | \mathbf{z}) + \log \mathcal{N}(\mathbf{z} | \mathbf{X}\mathbf{w}, \mathbf{I}) + \log \mathcal{N}(\mathbf{w} | \mathbf{0}, \mathbf{V}_0) \quad (15.172)$$

$$= \sum_n \log p(y_n | z_n) - \frac{1}{2} (\mathbf{z} - \mathbf{X}\mathbf{w})^\top (\mathbf{z} - \mathbf{X}\mathbf{w}) - \frac{1}{2} \mathbf{w}^\top \mathbf{V}_0^{-1} \mathbf{w} \quad (15.173)$$

The posterior in the E step is a **truncated Gaussian**:

$$p(z_n | y_n, \mathbf{x}_n, \mathbf{w}) = \begin{cases} \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n > 0) & \text{if } y_n = 1 \\ \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n < 0) & \text{if } y_n = 0 \end{cases} \quad (15.174)$$

In Equation (15.173), 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

$$\mathbb{E}[z_n | \mathbf{w}, \mathbf{x}_n] = \begin{cases} \mu_n + \frac{\phi(\mu_n)}{1 - \Phi(-\mu_n)} = \mu_n + \frac{\phi(\mu_n)}{\Phi(\mu_n)} & \text{if } y_n = 1 \\ \mu_n - \frac{\phi(\mu_n)}{\Phi(-\mu_n)} = \mu_n - \frac{\phi(\mu_n)}{1 - \Phi(\mu_n)} & \text{if } y_n = 0 \end{cases} \quad (15.175)$$

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

$$\hat{\mathbf{w}} = (\mathbf{V}_0^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{\mu} \quad (15.176)$$

The EM algorithm is simple, but can be much slower than direct gradient methods, as illustrated in Figure 15.12. 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 to derive a simple Gibbs sampling algorithm for approximating the posterior $p(\mathbf{w}|\mathcal{D})$ (see e.g., [AC93; HH06]).

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

$$p(z_i|y_i, \mathbf{x}_i, \mathbf{w}) = \begin{cases} \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i > 0) & \text{if } y_i = 1 \\ \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i < 0) & \text{if } y_i = 0 \end{cases} \quad (15.177)$$

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

$$p(\mathbf{w}|\mathcal{D}, \mathbf{z}, \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{w}_N, \mathbf{V}_N) \quad (15.178)$$

$$\mathbf{V}_N = (\mathbf{V}_0^{-1} + \mathbf{X}^T \mathbf{X})^{-1} \quad (15.179)$$

$$\mathbf{w}_N = \mathbf{V}_N(\mathbf{V}_0^{-1}\mathbf{w}_0 + \mathbf{X}^T \mathbf{z}) \quad (15.180)$$

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

$$y_n = j \quad \text{if} \quad \gamma_{j-1} < z_n \leq \gamma_j \quad (15.181)$$

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.

Finding the MLEs for this model is a bit trickier than for binary probit regression, since we need to optimize for \mathbf{w} and $\boldsymbol{\gamma}$, and the latter must obey an ordering constraint. See e.g., [KL09] for an approach based on EM. It is also possible to derive a simple Gibbs sampling algorithm for this model (see e.g., [Hof09, p216]).

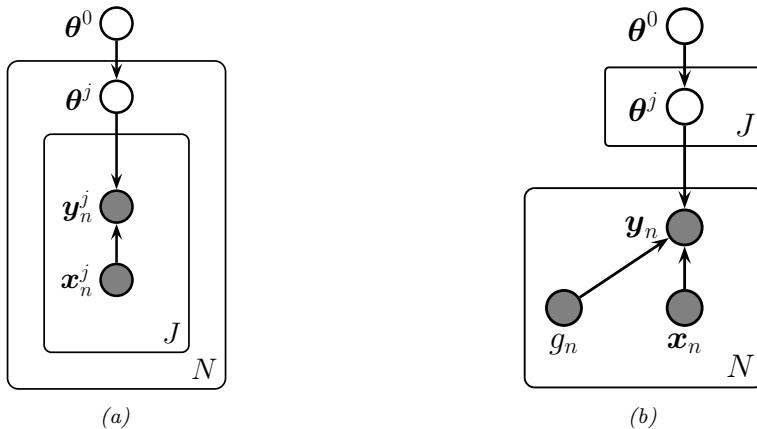


Figure 15.13: Hierarchical Bayesian discriminative models with J groups. (a) Nested formulation. (b) Non-nested formulation, with group indicator $g_n \in \{1, \dots, J\}$.

15.4.5 Multinomial probit models

Now consider the case where the response variable can take on C unordered categorical values, $y_n \in \{1, \dots, C\}$. The **multinomial probit** model is defined as follows:

$$z_{nc} = \mathbf{w}_c^\top \mathbf{x}_{nc} + \epsilon_{nc} \quad (15.182)$$

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \quad (15.183)$$

$$y_n = \arg \max_c z_{nc} \quad (15.184)$$

See e.g., [DE04; GR06b; Sco09; FSF10] for more details on the model and its connection to multinomial logistic regression.

If instead of setting $y_n = \operatorname{argmax}_c z_{ic}$ we use $y_{nc} = \mathbb{I}(z_{nc} > 0)$, we get a model known as **multivariate probit**, which is one way to model C correlated binary outcomes (see e.g., [TMD12]).

15.5 Multilevel (hierarchical) GLMs

Suppose we have a set of J related datasets, each of which contains a series of N_j datapoints $\mathcal{D}_j = \{(\mathbf{x}_n^j, \mathbf{y}_n^j) : n = 1 : N_j\}$. There are 3 main ways to fit models in such a setting: we could fit J 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 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 we can use a **hierarchical Bayesian model**, also called a **multilevel model** or **partially pooled model**, in which we assume each group has its own parameters, θ^j , but that these have something in common, as modeled by a shared global prior $p(\theta^0)$. (Note that each group could be a single individual.) The overall model has the form

$$p(\theta^{0:J}, \mathcal{D}) = p(\theta^0) \prod_{j=1}^J \left[p(\theta^j | \theta^0) \prod_{n=1}^{N_j} p(\mathbf{y}_n^j | \mathbf{x}_n^j, \theta^j) \right] \quad (15.185)$$

See Figure 15.13a, which represents the model using nested plate notation.

It is often more convenient to represent the model as in Figure 15.13b, which eliminates the nested plates (and hence the double indexing of variables) by associating a group indicator variable $g_n \in \{1, \dots, J\}$, which specifies which set of parameters to use for each datapoint. Thus the model now has the form

$$p(\boldsymbol{\theta}^{0:J}, \mathcal{D}) = p(\boldsymbol{\theta}^0) \left[\prod_{j=1}^J p(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0) \right] \left[\prod_{n=1}^N p(\mathbf{y}_n | \mathbf{x}_n, g_n, \boldsymbol{\theta}) \right] \quad (15.186)$$

where

$$p(\mathbf{y}_n | \mathbf{x}_n, g_n, \boldsymbol{\theta}) = \prod_{j=1}^J p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}^j)^{\mathbb{I}(g_n=j)} \quad (15.187)$$

If the likelihood function is a GLM, this hierarchical model is called a **hierarchical GLM** [LN96]. This class of models is very widely used in applied statistics. For much more details, see e.g., [GH07; GHV20b; Gel+22].

15.5.1 Generalized linear mixed models (GLMMs)

Suppose that the prior on the per-group parameters is Gaussian, so $p(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0) = \mathcal{N}(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0, \boldsymbol{\Sigma}^j)$. If we have a GLM likelihood, the model becomes

$$p(\mathbf{y}_n | \mathbf{x}_n, g_n = j, \boldsymbol{\theta}) = p(\mathbf{y}_n | \ell(\eta_n)) \quad (15.188)$$

$$\eta_n = \mathbf{x}_n^\top \boldsymbol{\theta}^j = \mathbf{x}_n^\top (\boldsymbol{\theta}^0 + \boldsymbol{\epsilon}^j) = \mathbf{x}_n^\top \boldsymbol{\theta}^0 + \mathbf{x}_n^\top \boldsymbol{\epsilon}^j \quad (15.189)$$

where ℓ is the link function, and $\boldsymbol{\epsilon}^j \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. This is known as a **generalized linear mixed model (GLMM)** or **mixed effects model**. The shared (common) parameters $\boldsymbol{\theta}^0$ are called **fixed effects**, and the group-specific offsets $\boldsymbol{\epsilon}^j$ 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.

For model fitting, we can use any of the Bayesian inference methods that we discussed in Section 15.1.4.

15.5.2 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/.

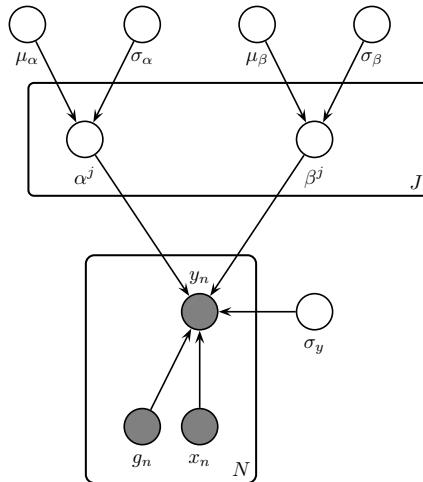


Figure 15.14: A hierarchical Bayesian linear regression model for the radon problem.

a binary feature representing whether the house has a basement or not. We use a dataset consisting of $J = 85$ counties in Minnesota; each county has between 2 and 80 measurements.

We assume the following likelihood:

$$p(y_n|x_n, g_n = j, \theta) = \mathcal{N}(y_n|\alpha_j + \beta_j x_n, \sigma_y^2) \quad (15.190)$$

where $g_n \in \{1, \dots, J\}$ is the county for house i , and $x_n \in \{0, 1\}$ indicates if the floor is at level 0 (i.e., in the basement) or level 1 (i.e., above ground). Intuitively we expect the radon levels to be lower in houses without basements, since they are more insulated from the earth which is the source of the radon.

Since some counties have very few datapoints, we use a hierarchical prior in which we assume $\alpha_j \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2)$, and $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$. We use weak priors for the parameters: $\mu_\alpha \sim \mathcal{N}(0, 1)$, $\mu_\beta \sim \mathcal{N}(0, 1)$, $\sigma_\alpha \sim \mathcal{C}_+(1)$, $\sigma_\beta \sim \mathcal{C}_+(1)$, $\sigma_y \sim \mathcal{C}_+(1)$. See Figure 15.14 for the graphical model.

15.5.2.1 Posterior inference

Figure 15.15 shows the posterior marginals for μ_α , μ_β , α_j and β_j . We see that μ_β is close to -0.6 with high probability, which confirms our suspicion that having $x = 1$ (i.e., no basement) decreases the amount of radon in the house. We also see that the distribution of the α_j parameters is quite variable, due to different base rates across the counties.

Figure 15.16 shows predictions from the hierarchical and non-hierarchical model for 3 different counties. We see that the predictions from the hierarchical model are more consistent across counties, and work well even if there are no examples of certain feature combinations for a given county (e.g., there are no houses without basements in the sample from Cass county). If we sample data from the posterior predictive distribution, and compare it to the real data, we find that the RMSE is 0.13 for the non-hierarchical model and 0.08 for the hierarchical model, indicating that the latter fits better.

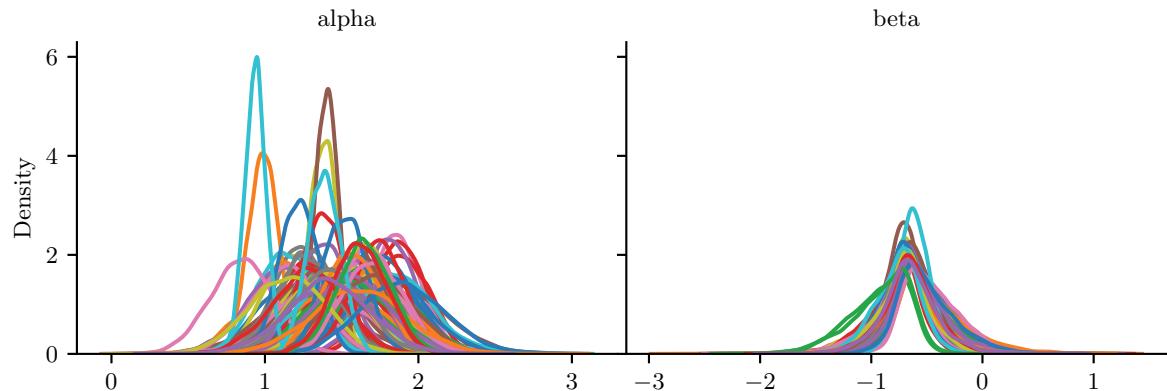


Figure 15.15: Posterior marginals for α_j and β_j for each county j in the radon model. Generated by [linreg_hierarchical_non_centered.ipynb](#).

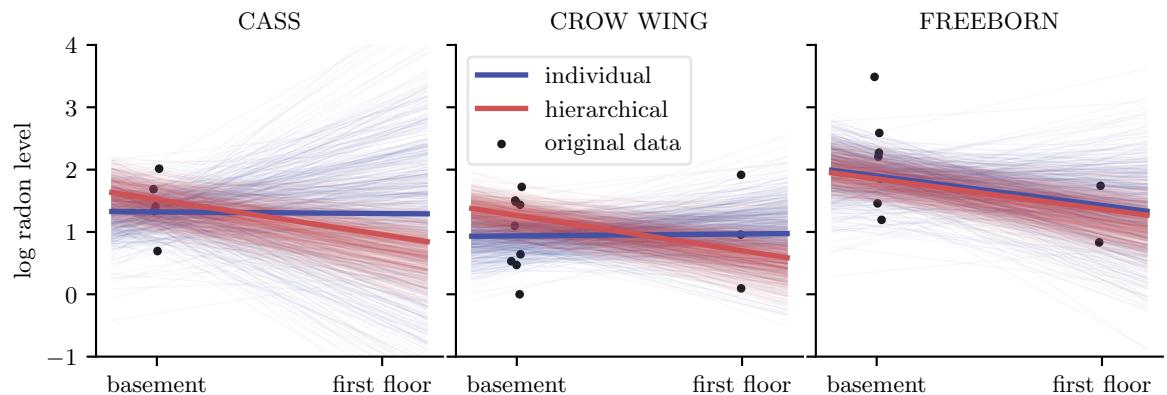


Figure 15.16: Predictions from the radon model for 3 different counties in Minnesota. Black dots are observed datapoints. Red represents results of hierarchical (shared) prior, blue represents results of non-hierarchical prior. Thick lines are the result of using the posterior mean, thin lines are the result of using posterior samples. Generated by [linreg_hierarchical_non_centered.ipynb](#).

15.5.2.2 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 15.17a gives an example where we plot $p(\beta_j, \sigma_\beta | \mathcal{D})$ for some specific county j . If we believe that σ_β is large, then β_j 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_j 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 sampling algorithms) to explore parts of parameter space at the bottom of the funnel. This is evident from the marginal posterior for σ_β shown (as a histogram) on the right hand side of the plot: we see

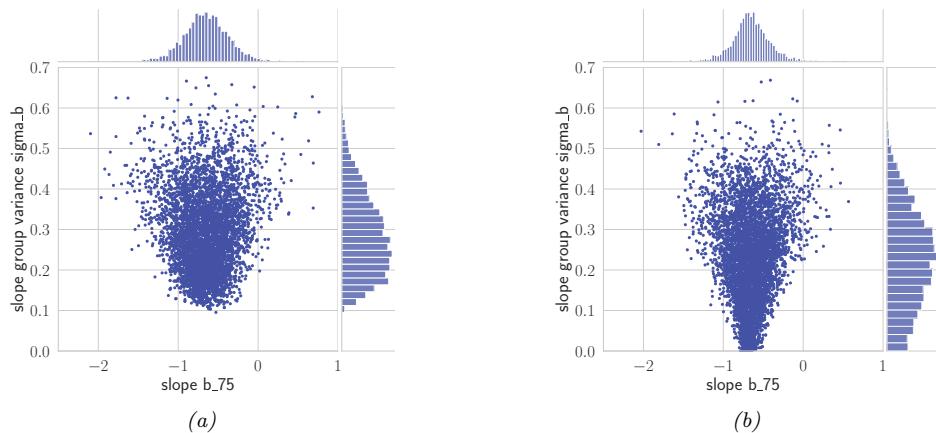


Figure 15.17: (a) Bivariate posterior $p(\beta_j, \sigma_\beta | \mathcal{D})$ for the hierarchical radon model for county $j = 75$ using centered parameterization. (b) Similar to (a) except we plot $p(\tilde{\beta}_j, \sigma_\beta | \mathcal{D})$ for the non-centered parameterization. Generated by [linreg_hierarchical_non_centered.ipynb](#).

that it excludes the interval $[0, 0.1]$, thus ruling out models in which we shrink β_j all the way to 0. In cases where a covariate has no useful predictive role, we would like to be able to induce sparsity, so we need to overcome this problem.

A simple solution to this is to use a **non-centered parameterization** [PR03]. That is, we replace $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$ with $\beta_j = \mu_\beta + \tilde{\beta}_j \sigma_\beta$, where $\tilde{\beta}_j \sim \mathcal{N}(0, 1)$ represents the *offset* from the global mean, μ_β . The correlation between $\tilde{\beta}_j$ and σ_β is much less, as shown in Figure 15.17b. See Section 12.6.5 for more details.

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 $p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\mathbf{x}; \boldsymbol{\theta})))$, 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 $q(\mathbf{z}|f(\mathcal{D}; \boldsymbol{\phi}))$, where \mathbf{z} 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 10.1.5. Note that in this latter, setting the joint model $p(\mathbf{z}, \mathcal{D})$ 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; Raf22], as well as a multitude of online courses. For a more theoretical treatment, see e.g., [Ber+21; 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} .



Figure 16.1: An articial “neuron”, the most basic building block of a DNN. (a) The output y is a weighted combination of the inputs \mathbf{x} , where the weights vector is denoted by \mathbf{w} . (b) Alternative depiction of the neuron’s behavior. The bias term b can be emulated by defining $w_N = b$ and $X_N = 1$.

16.2.1 Linear layers

The most basic building block of a DNN is a single “**neuron**”, which corresponds to a real-valued signal y computed by multiplying a vector-valued input signal \mathbf{x} by a weight vector \mathbf{w} , and then adding a bias term b . That is,

$$y = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^\top \mathbf{x} + b \quad (16.1)$$

where $\boldsymbol{\theta} = (\mathbf{w}, b)$ are the parameters for the function f . This is depicted in Figure 16.1. (The bias term is omitted for clarity.)

It is common to group a set of neurons together into a **layer**. We can then represent the activations 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 an output vector \mathbf{y} by multiplying by a weight matrix \mathbf{W} , an adding an offset vector or bias term \mathbf{b} to get

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (16.2)$$

where $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$ are the parameters for the function f . This is called a **linear layer**, or **fully connected layer**.

It is common to prepend the bias vector onto the first column of the weight matrix, and to append 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 $\tilde{\mathbf{x}} = [\mathbf{x}, 1]$. This allows us to ignore the bias term from our notation if we want to.

16.2.2 Nonlinearities

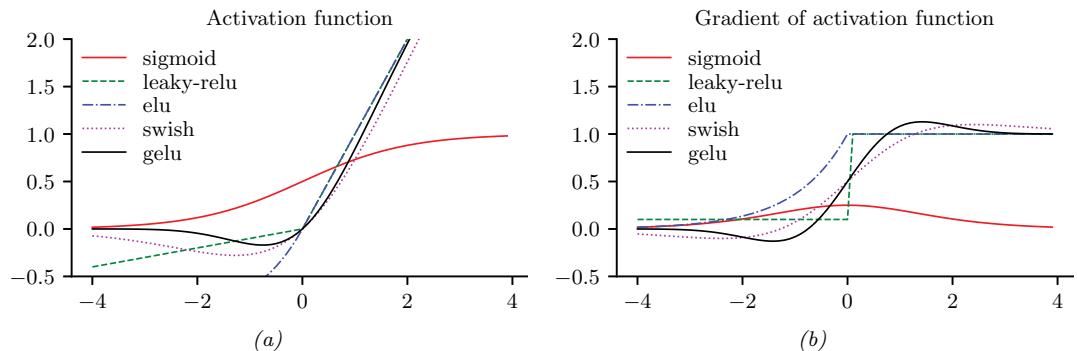
A stack of linear layers is equivalent to a single linear layer where we multliply together all the weight matrices. To get more expressive power we can transform each layer by passing it elementwise (pointwise) through a nonlinear function called an **activation function**. This is denoted by

$$\mathbf{y} = \varphi(\mathbf{x}) = [\varphi(x_1), \dots, \varphi(x_D)] \quad (16.3)$$

See Table 16.1 for a list of some common activation functions, and Figure 16.2 for a visualization. For more details, see e.g., [Mur22, Sec 13.2.3].

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; KSH12a]
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]
Sine	$\sin(a)$	$[-1, 1]$	[Sit+20]

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.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

1. Note that, technically speaking, we are using **cross correlation** rather than convolution. However, these terms are used interchangeably in deep learning.

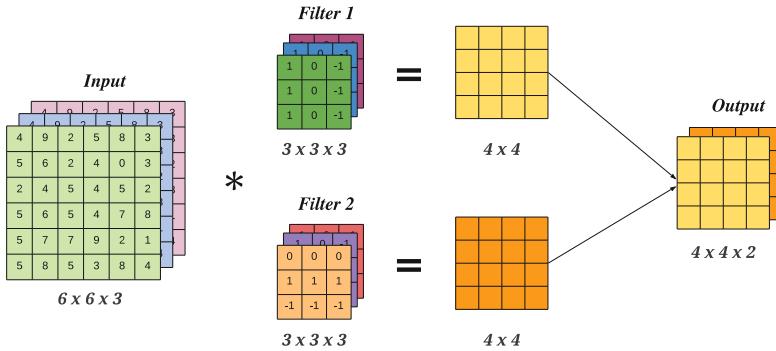


Figure 16.3: A 2d convolutional layer with 3 input channels and 2 output channels. The kernel has size 3×3 and we use stride 1 with 0 padding, so the 6×6 input gets mapped to the 4×4 output.

3d in an analogous manner. Note that the spatial size of the outputs may be smaller than inputs, 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

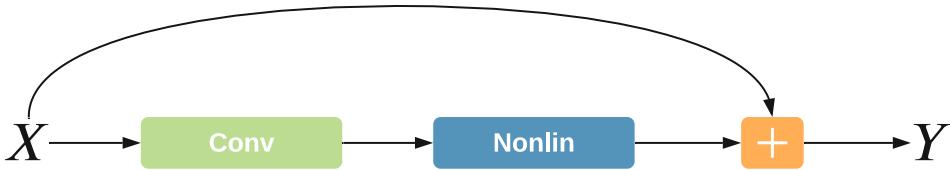


Figure 16.4: A residual connection around a convolutional layer.

allow the signal to skip one or more layers, which prevents it from being modified. For example, 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+14a], 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

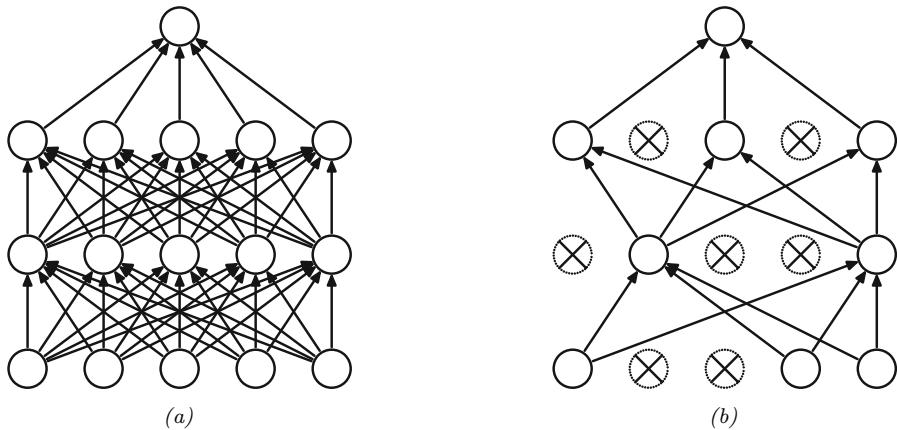


Figure 16.5: Illustration of dropout. (a) A standard neural net with 2 hidden layers. (b) An example of a thinned net produced by applying dropout with $p = 0.5$. Units that have been dropped out are marked with an x . From Figure 1 of [Sri+14a]. Used with kind permission of Geoff Hinton.

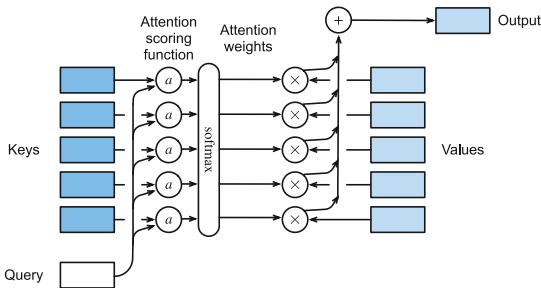


Figure 16.6: Attention layer. (a) Mapping a single query to a single output, given a set of keys and values. From Figure 10.3.1 of [Zha+20a]. Used with kind permission of Aston Zhang.

of weights will be updated on each step. The result is an **ensemble** of networks, each with slightly different sparse graph structures.

At test time, we usually turn the dropout noise off, so the model acts deterministically. To ensure the weights have the same expectation at test time as they did during training (so the input activation 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 noise, we have $\mathbb{E}[\epsilon] = 1 - p$, so we should multiply the weights by the keep probability, $1 - p$, before making predictions. We can, however, use dropout at test time if we wish. This is called **Monte Carlo dropout** (see Section 17.3.1).

16.2.7 Attention layers

In all of the neural networks we have considered so far, the hidden activations are a linear combination of the input activations, followed by a nonlinearity: $\mathbf{Z} = \varphi(\mathbf{XW})$, where $\mathbf{X} \in \mathbb{R}^{n \times d}$ are the hidden

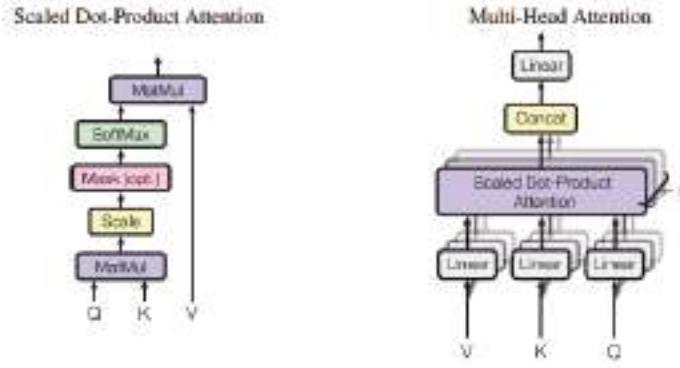


Figure 16.7: (a) Scaled dot-product attention in matrix form. (b) Multi-head attention. From Figure 2 of [Vas+17b]. Used with kind permission of Ashish Vaswani.

feature vectors, and $\mathbf{W} \in \mathbb{R}^{d_v \times d_v}$ are a fixed set of weights that are learned on a training set to produce $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$ outputs. However, we can imagine a more flexible model in which the weights depend on the inputs, i.e., $\mathbf{Z} = \varphi(\mathbf{X}\mathbf{W}(\mathbf{X}))$, where $\mathbf{W}(\mathbf{X})$ is a function to be defined below. This kind of **multiplicative interaction** is called **attention**.

We can better understand attention by comparing it to non-parametric kernel based prediction methods, such as Gaussian processes (Chapter 18). In this approach we compare the input query $\mathbf{x} \in \mathbb{R}^d$ to each of the training examples $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ using a kernel to get a vector of similarity scores, $\boldsymbol{\alpha} = [\mathcal{K}(\mathbf{x}, \mathbf{x}_i)]_{i=1}^n$. We then use this to retrieve a weighted combination of the corresponding m target values $\mathbf{y}_i \in \mathbb{R}^{d_v}$ to compute the predicted output, as follows:

$$\hat{\mathbf{y}} = \sum_{i=1}^n \alpha_i \mathbf{y}_i \quad (16.7)$$

See Section 18.3.7 for details.

We can make a differentiable and parametric version of this as follows (see [Tsa+19] for details). First we replace the stored examples matrix \mathbf{X} with a learned embedding, to create a set of stored **keys**, $\mathbf{K} = \mathbf{X}\mathbf{W}_k \in \mathbb{R}^{n \times d_k}$. Similarly we replace the stored output matrix \mathbf{Y} with a learned embedding, to create a set of stored **values**, $\mathbf{V} = \mathbf{Y}\mathbf{W}_v \in \mathbb{R}^{n \times d_v}$. Finally we embed the input to 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.

Next, we replace fixed kernel function with a soft **attention layer**. More precisely, we define the weighted output for query \mathbf{q} to be

$$\text{Attn}(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_n, \mathbf{v}_n)) = \text{Attn}(\mathbf{q}, (\mathbf{k}_{1:n}, \mathbf{v}_{1:n})) = \sum_{i=1}^n \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) \mathbf{v}_i \quad (16.8)$$

where $\alpha_i(\mathbf{q}, \mathbf{k}_{1:n})$ is the i 'th **attention weight**; these weights satisfy $0 \leq \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) \leq 1$ for each i and $\sum_i \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) = 1$.

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:

$$\alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) = \text{softmax}_i([a(\mathbf{q}, \mathbf{k}_1), \dots, a(\mathbf{q}, \mathbf{k}_n)]) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^n \exp(a(\mathbf{q}, \mathbf{k}_j))} \quad (16.10)$$

See Figure 16.6 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 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.

For efficiently, we usually compute all n vectors in parallel. Let the corresponding matrices of queries, keys and values be denoted by $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{n \times d_k}$, $\mathbf{V} \in \mathbb{R}^{n \times d_v}$. Let

$$\mathbf{z}_j = \sum_{i=1}^n \alpha_i(\mathbf{q}_j, \mathbf{K}) \mathbf{v}_i \quad (16.11)$$

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

$$\mathbf{Z} = \text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \quad (16.12)$$

where the softmax function softmax is applied row-wise. See Figure 16.7 (left) for an illustration.

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

$$\mathbf{Z} = \text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\mathbf{h}_1, \dots, \mathbf{h}_h)\mathbf{W}^O \quad (16.14)$$

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 16.3.5.

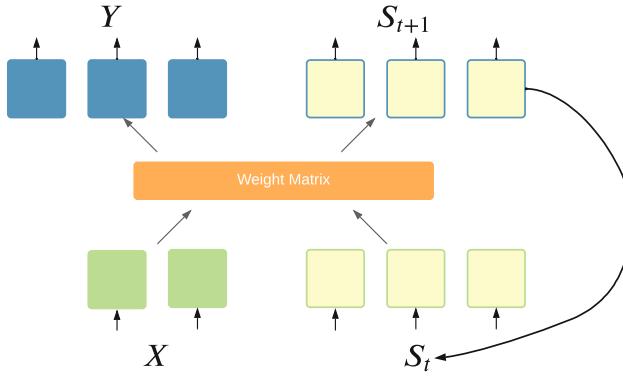


Figure 16.8: Recurrent layer.

16.2.8 Recurrent layers

We can make the model be **stateful** by augmenting the input \mathbf{x} with the current state \mathbf{s}_t , and then computing the output and the new state using some kind of function:

$$(\mathbf{y}, \mathbf{s}_{t+1}) = f(\mathbf{x}, \mathbf{s}_t) \quad (16.15)$$

This is called a **recurrent layer**, as shown in Figure 16.8. This forms the basis of **recurrent neural networks**, discussed in Section 16.3.4. 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

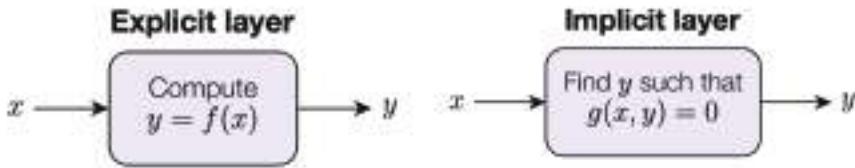


Figure 16.9: Explicit vs implicit layers.

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 $\mathbf{W}' = \mathbf{z}^T \mathbf{W} + \mathbf{V}$ 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 16.3.4) as a form of multiplicative interaction. In particular, if the hypernetwork computes the diagonal matrix $\mathbf{W}' = \sigma(\mathbf{z}^T \mathbf{W} + \mathbf{V}) = \text{diag}(a_1, \dots, a_n)$, 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 $y = f(\mathbf{x})$. 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.

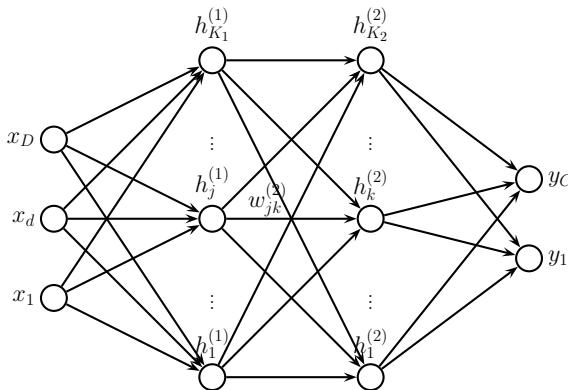


Figure 16.10: A feedforward neural network with D inputs, K_1 hidden units in layer 1, K_2 hidden units in layer 2, and C outputs. $w_{jk}^{(l)}$ is the weight of the connection from node j in layer $l-1$ to node k in layer l .

16.3.1 Multilayer perceptrons (MLPs)

A **multilayer 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. However, it is also common for the final layer to have linear activations, in which case the outputs are interpreted as logits; the loss function used during training then converts to (log) probabilities internally.

We can also use MLPs for regression. Figure 16.11 shows how we can make a model for **heteroskedastic** nonlinear regression. (The term “heteroskedastic” just means that the predicted output variance is input-dependent, rather than a constant.) This function has two outputs which compute $f_\mu(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}, \boldsymbol{\theta}]$ and $f_\sigma(\mathbf{x}) = \sqrt{\mathbb{V}[y|\mathbf{x}, \boldsymbol{\theta}]}$. We can share most of the layers (and hence parameters) between these two functions by using a common “**backbone**” and two output “**heads**”, as shown in Figure 16.11. For the μ head, we use a linear activation, $\varphi(a) = a$. For the σ head, we use a softplus activation, $\varphi(a) = \sigma_+(a) = \log(1 + e^a)$. If we use linear heads and a nonlinear backbone, the overall model is given by

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y | \mathbf{w}_\mu^\top f(\mathbf{x}; \mathbf{w}_{\text{shared}}), \sigma_+(\mathbf{w}_\sigma^\top f(\mathbf{x}; \mathbf{w}_{\text{shared}}))) \quad (16.22)$$

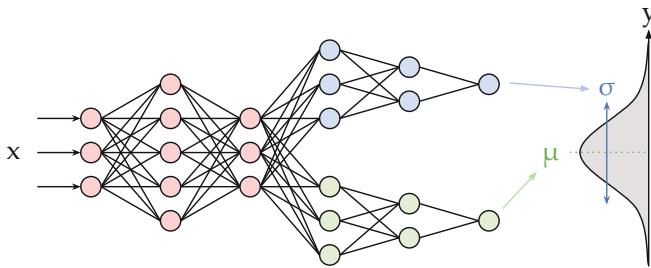


Figure 16.11: Illustration of an MLP with a shared “backbone” and two output “heads”, one for predicting the mean and one for predicting the variance. From <https://brendanhasz.github.io/2019/07/23/bayesian-density-net.html>. Used with kind permission of Brendan Hasz.

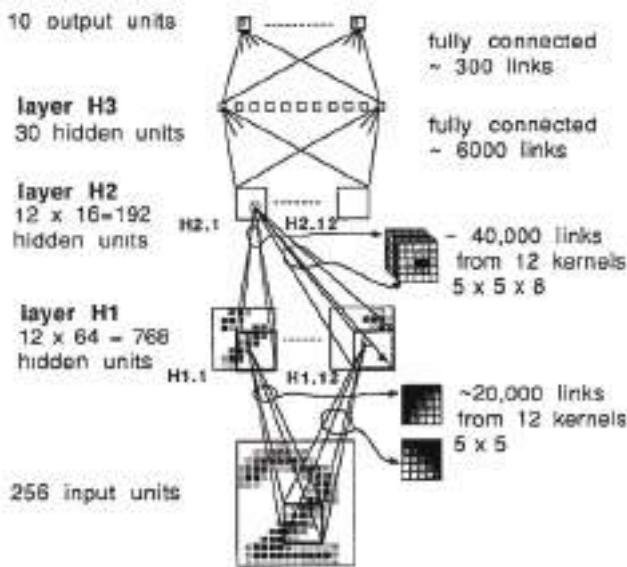


Figure 16.12: One of the first CNNs ever created, for classifying MNIST images. From Figure 3 of [LeC+89]. For a “modern” implementation, see lecun1989.ipynb.

16.3.2 Convolutional neural networks (CNNs)

A vanilla **convolutional neural network** or **CNN** consists of a series of convolutional layers, pooling layers, linear layers, and nonlinearities. See Figure 16.12 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+22b] 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 more details on CNNs.

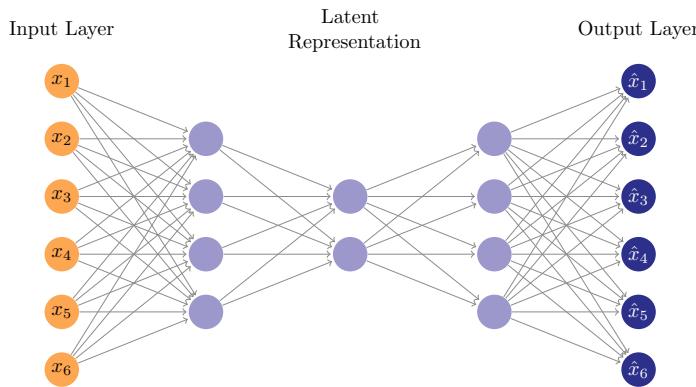


Figure 16.13: Illustration of an autoencoder with 3 hidden layers.



Figure 16.14: (a) Some MNIST digits. (b) Reconstruction of these images using a convolutional autoencoder. (c) t-SNE visualization of the 20-d embeddings. The colors correspond to class labels, which were not used during training. Generated by `ae_mnist_conv_jax.ipynb`.

16.3.3 Autoencoders

An **autoencoder** is a neural network that maps inputs \mathbf{x} to a low-dimensional latent space using an **encoder**, $\mathbf{z} = f_e(\mathbf{x})$, and then attempts to reconstruct the inputs using a **decoder**, $\hat{\mathbf{x}} = f_d(\mathbf{z})$. The model is trained to minimize

$$\mathcal{L}(\boldsymbol{\theta}) = \|\mathbf{r}(\mathbf{x}) - \mathbf{x}\|_2^2 \quad (16.23)$$

where $\mathbf{r}(\mathbf{x}) = f_d(f_e(\mathbf{x}))$. (We can also replace squared error with more general conditional log likelihoods.) See Figure 16.13 for an illustration of a 3 layer AE.

For image data, we can make the encoder be a convolutional network, and the decoder be a transpose convolutional network. We can use this to compute low dimensional embeddings of image

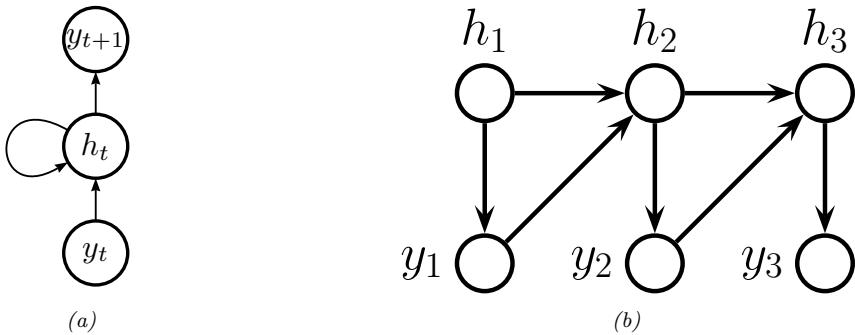


Figure 16.15: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

data. For example, suppose we fit such a model to some MNIST digits. We show the reconstruction abilities of such a model in Figure 16.14b. In Figure 16.14c, we show a 2d visualization of the 20-dimensional embedding space computed using t-SNE. The colors correspond to class labels, which were not used during training. We see fairly good separation, showing that images which are visually similar are placed close to each other in the embedding space, as desired. (See also Section 21.2.3, where we compare AEs with variational AEs.)

16.3.4 Recurrent neural networks (RNNs)

A **recurrent neural network (RNN)** is a network with a recurrent layer, as in Equation (16.15). This is illustrated in Figure 16.15. Formally this defines the following probability distribution over sequences:

$$p(\mathbf{y}_{1:T}) = \sum_{\mathbf{h}_{1:T}} p(\mathbf{y}_{1:T}, \mathbf{h}_{1:T}) = \sum_{\mathbf{h}_{1:T}} \mathbb{I}(\mathbf{h}_1 = \mathbf{h}_1^*) p(\mathbf{y}_1 | \mathbf{h}_1) \prod_{t=2}^T p(\mathbf{y}_t | \mathbf{h}_t) \mathbb{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1})) \quad (16.24)$$

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 more details on RNNs.

16.3.5 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

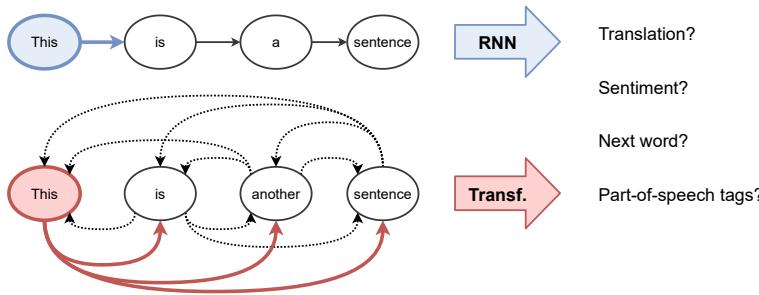


Figure 16.16: Visualizing the difference between an RNN and a transformer. From [Jos20]. Used with kind permission of Chaitanya Joshi.

the word at location t , \mathbf{z}_t , depends on the hidden state of the network, 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 16.16.

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 22.4.1). We can combine the encoder and decoder to create a conditional sequence-to-sequence model, $p(\mathbf{y}_{1:T_y} | \mathbf{x}_{1:T_x})$, as proposed in the original transformer paper [Vas+17c]. See Supplementary Section 16.1.1 and [PH22] for more details.

It has been found that large transformers are very flexible sequence-to-sequence function approximators, if trained on enough data (see e.g., [Lin+21a] 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 insights can be found in, e.g., [Rag+21; WGY21; Nel21; BP21]. See also Supplementary Section 16.1.2.5 where we discuss the connection with graph neural networks.

16.3.6 Graph neural networks (GNNs)

It is possible to define neural networks for working with graph-structured data. These are called **graph neural networks** or **GNNs**. See Supplementary Section 16.1.2 for details.

17 Bayesian neural networks

This chapter is coauthored with Andrew Wilson.

17.1 Introduction

Deep neural networks (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. (This phenomenon is known as **underspecification** (see e.g., [D'A+20]; see Figure 17.11 for an illustration.) Considering all of these different models together can lead to improved accuracy and uncertainty representation. This can be done by computing the posterior predictive distribution using Bayesian model averaging:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (17.1)$$

where $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$.

The main challenges in applying Bayesian inference to DNNs are specifying suitable priors, and efficiently computing the posterior, which is challenging due to the large number of parameters and the large datasets. The application of Bayesian inference to DNNs is sometimes called **Bayesian deep learning** or **BDL**. By contrast, the term **deep Bayesian learning** or **DBL** refers to the use of deep models to help speed up Bayesian inference of “classical” models, usually by training amortized inference networks that can be used as part of a variational inference or importance sampling algorithm, as discussed in Section 10.1.5.) For more details on the topic of BDL, 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; For22] discusses the issue of prior selection at length. Here we just give a brief summary of common approaches.

17.2.1 Gaussian priors

Consider an MLP with one hidden layer with activation function φ and a linear output:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_2 \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \quad (17.2)$$

(If the output is nonlinear, such as a softmax transform, we can fold it into the loss function during training.) If we have two hidden layers this becomes

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_3 (\varphi(\mathbf{W}_2 \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)) + \mathbf{b}_3 \quad (17.3)$$

In general, with $L - 1$ hidden layers and a linear output, we have

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_L (\cdots \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_L \quad (17.4)$$

We need to specify the priors for \mathbf{W}_l and \mathbf{b}_l for $l = 1 : L$. The most common choice is to use a factored Gaussian prior:

$$\mathbf{W}_\ell \sim \mathcal{N}(\mathbf{0}, \alpha_\ell^2 \mathbf{I}), \mathbf{b}_\ell \sim \mathcal{N}(\mathbf{0}, \beta_\ell^2 \mathbf{I}) \quad (17.5)$$

The **Xavier initialization** or **Glorot initialization**, named after the first author of [GB10], is to set

$$\alpha_\ell^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \quad (17.6)$$

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

$$\alpha_\ell^2 = \frac{1}{n_{\text{in}}} \quad (17.7)$$

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:

$$\mathbf{W}_\ell = \alpha_\ell \boldsymbol{\eta}_\ell, \boldsymbol{\eta}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \mathbf{b}_\ell = \beta_\ell \boldsymbol{\epsilon}_\ell, \boldsymbol{\epsilon}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (17.8)$$

Hence every setting of the prior hyperparameters specifies the following random function:

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \alpha_L \boldsymbol{\eta}_L (\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x} + \beta_1 \boldsymbol{\epsilon}_1)) + \beta_L \boldsymbol{\epsilon}_L \quad (17.9)$$

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 1d 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

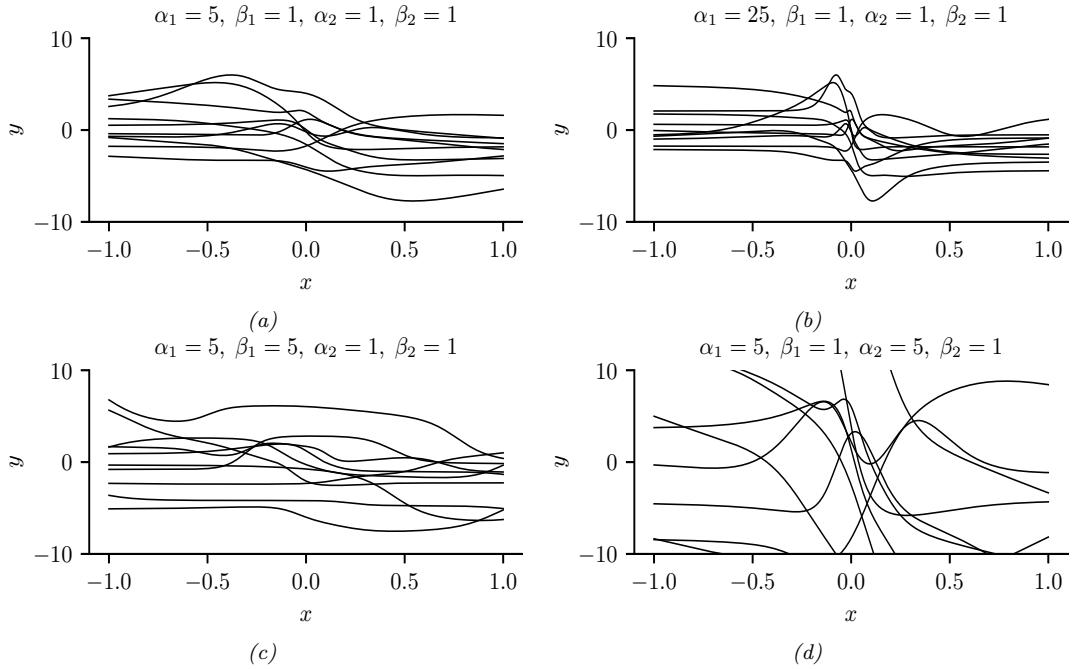


Figure 17.1: The effects of changing the hyperparameters on an MLP with one hidden layer. (a) Random functions sampled from a Gaussian prior with hyperparameters $\alpha_1 = 5, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$. (b) Increasing α_1 by a factor of 5. (c) Increasing β_1 by a factor of 5. (d) Increasing α_2 by a factor of 5. Generated by `mlp_priors_demo.ipynb`.

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 (17.9) simplifies to

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta} = \mathbf{0}) = \alpha_L \boldsymbol{\eta}_L(\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x})) = \alpha_L \cdots \alpha_1 \boldsymbol{\eta}_L(\cdots \varphi(\boldsymbol{\eta}_1 \mathbf{x})) \quad (17.10)$$

$$= \alpha_L \cdots \alpha_1 f(\mathbf{x}; (\boldsymbol{\alpha} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0})) \quad (17.11)$$

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 preactivation $z \geq 0$). 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 (cf. Section 15.2.6). For details, see [Hoe+21].

17.2.3 Learning the prior

We have seen how different priors for the parameters correspond to different priors over functions. We could in principle set the hyperparameters (e.g., the α and β parameters of the Gaussian prior) using grid search to optimize cross-validation loss. However, cross-validation can be slow, particularly if we allow different priors for each layer of the network, as our grid search will grow exponentially with the number of hyperparameters we wish to determine.

An alternative is to use gradient based methods to optimize the marginal likelihood

$$\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \int \log p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\alpha}, \boldsymbol{\beta})d\boldsymbol{\theta} \quad (17.12)$$

This approach is known as empirical Bayes (Section 3.7) or **evidence maximization**, since $\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta})$ is also called the evidence [Mac92a; WS93; Mac99]. This can give rise to sparse models, as we discussed in the context of automatic relevancy determination (Section 15.2.8). Unfortunately, computing the marginal likelihood is computationally difficult for large neural networks.

Learning the prior is more meaningful if we can do it on a separate, but related dataset. In [SZ+22] they propose to train a model on an initial, large dataset \mathcal{D}_1 (possibly unsupervised) to get a point estimate, $\hat{\boldsymbol{\theta}}_1$, from which they can derive an approximate low-rank Gaussian posterior, using the SWAG method (Section 17.3.8). They then use this informative prior when fine-tuning the model on a downstream dataset \mathcal{D}_2 . The fine-tuning can either be a MAP estimate $\hat{\boldsymbol{\theta}}_2$ or some approximate posterior, $p(\boldsymbol{\theta}_2|\mathcal{D}_2, \mathcal{D}_1)$, e.g., computed using MCMC (Section 17.3.7). They call this technique “**Bayesian transfer learning**”. (See Section 19.5.1 for more details on transfer learning.)

17.2.4 Priors in function space

Typically, the relationship between the prior distribution over parameters and the functions preferred by the prior is not transparent. In some cases, it can be possible to pick more informative priors based on principles such as desired invariances that we want the function to satisfy (see e.g., [Nal18]). [FBW21] introduces *residual pathway priors*, providing a mechanism for encoding high level concepts into prior distributions, such as locality, independencies, and symmetries, without constraining model flexibility. A different approach to encoding interpretable priors over functions leverages kernel methods such as Gaussian processes (e.g., [Sun+19a]), as we discuss in Section 18.1.

17.2.5 Architectural priors

Beyond specifying the parametric prior, it is important to note that the architecture of the model can have an even larger effect on the induced distribution over functions, as argued in Wilson and Izmailov [WI20] and Izmailov et al. [Izm+21b]. For example, a CNN architecture encodes prior knowledge about translation equivariance, due to its use of convolution, and hierarchical structure, due to its use of multiple layers. Other forms of inductive bias are induced by different architectures, such as RNNs. (Models such as transformers have weaker inductive bias, but consequently often need more data to perform well.) Thus we can think of the field of **neural architecture search** (reviewed in [EMH19]) as a form of structural prior learning.

In fact, with a suitable architecture, we can often get good results using random (untrained) models. For example, Ulyanov, Vedaldi, and Lempitsky [UVL18] showed that an untrained CNN with random parameters (sampled from a Gaussian) often works very well for low-level image processing tasks, such as image denoising, super-resolution, and image inpainting. The resulting prior over functions has been called the **deep image prior**. Similarly, Pinto and Cox [PC12] showed that untrained CNNs with the right structure can do well at face recognition. Moreover, Zhang et al. [Zha+17] show that randomly initialized CNNs can process data to provide features that greatly improve the performance of other models, such as kernel methods.

17.3 Posteriors for BNNs

There are a large number of different approximate inference schemes that have been applied to Bayesian neural networks, with different strengths and limitations. In the sections below, we briefly describe some of these.

17.3.1 Monte Carlo dropout

Monte Carlo dropout (MCD) [GG16; KG17] is a very simple and widely used method for approximating the Bayesian predictive distribution. Usually stochastic dropout layers are added as a form of regularization, and are “turned off” at test time, as described in Section 16.2.6. However, the idea in MCD is to also perform random sampling at test time. More precisely, 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:

$$p(y|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(y|\mathbf{x}, \boldsymbol{\theta}^s) \quad (17.13)$$

where $\boldsymbol{\theta}^s$ is a version of the MAP parameter estimate where we randomly drop out some connections.

We give an example of this process in action in Figure 17.2. We see that it successfully captures uncertainty due to “out of distribution” inputs. (See Section 19.3.2 for more discussion of OOD detection.)

One drawback of MCD is that it is slow at test time. However this can be overcome by “distilling” the model’s predictions into a deterministic “student” network, as we discuss in Section 17.3.10.3.

A more fundamental problem is that MCD does not give proper uncertainty estimates, as argued in [Osb16; LF+21]. The problem is the following. Although MCD can be viewed as a form of variational

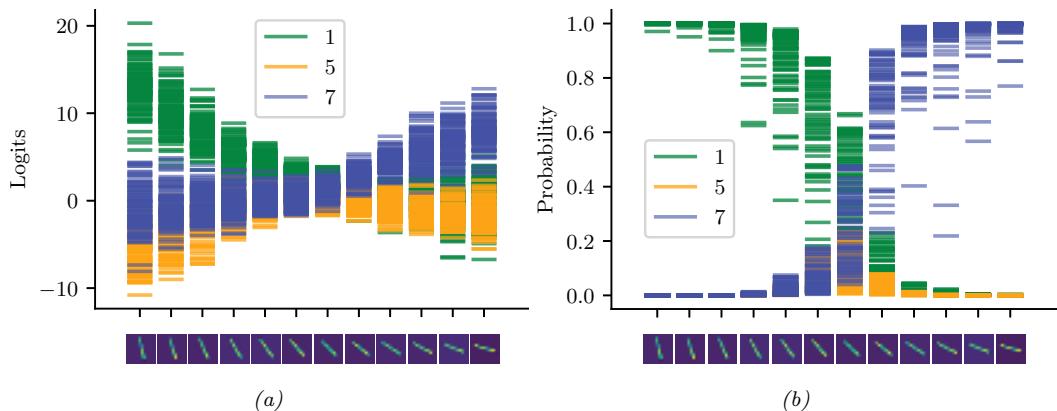


Figure 17.2: Illustration of MC dropout applied to the LeNet architecture. The inputs are some rotated images of the digit 1 from the MNIST dataset. (a) Softmax inputs (logits). (b) Softmax outputs (probabilities). We see that the inputs are classified as digit 7 for the last three images (as shown by the probabilities), even though the model has high uncertainty (as shown by the logits). Adapted from Figure 4 of [GG16]. Generated by [mnist_classification_mc_dropout.ipynb](#)

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 [Osb16]. Fortunately this pathology can be fixed if the noise rate is optimized [GHK17]. For more details, see e.g., [HMG18; NHL19; LF+21].

17.3.2 Laplace approximation

In Section 7.4.3, we introduced the Laplace approximation, which computes a Gaussian approximation to the posterior, $p(\boldsymbol{\theta}|\mathcal{D})$, centered at the MAP estimate, $\boldsymbol{\theta}^*$. The posterior prediction matrix is equal to the Hessian of the negative log joint computed at the mode. The benefits of this approach are that it is simple, and it can be used to derive a Bayesian estimate from a pretrained model. The main disadvantage is that computing the Hessian can be expensive. In addition, it may not be positive definite, since the log likelihood of DNNs is non-convex. It is therefore common to use a Gauss-newton approximation to the Hessian instead, as we explain below.

Following the notation of [Dax+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 vector. Let $\mathbf{r}(\mathbf{y}; \mathbf{f}) = \nabla_{\mathbf{f}} \log p(\mathbf{y}|\mathbf{f})$ be the residual¹, and $\Lambda(\mathbf{y}; \mathbf{f}) = -\nabla_{\mathbf{f}}^2 \log p(\mathbf{y}|\mathbf{f})$ be the per-input 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 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

1. In the Gaussian case, this term becomes $\nabla_{\mathbf{f}} \|\mathbf{y} - \mathbf{f}\|^2 = 2\|\mathbf{y} - \mathbf{f}\|$, so it can be interpreted as a residual error.

of the log likelihood are given by the following [IKB21]:

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) \quad (17.14)$$

$$\nabla_{\boldsymbol{\theta}}^2 \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{H}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) - \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{\Lambda}(\mathbf{y}; \mathbf{f}) \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{\theta}) \quad (17.15)$$

Since the network Hessian \mathbf{H} is usually intractable to compute, it is usually dropped, leaving only the Jacobian term. This is called the **generalized Gauss-Newton** or **GGN** approximation [Sch02; Mar20]. The GGN approximation is guaranteed to be positive definite. By contrast, this is not true for the original Hessian in Equation (17.15), since the objective is not convex. Furthermore, computing the Jacobian term is cheaper to compute than the Hessian.

Putting it all together, for a Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}_0, \mathbf{S}_0)$, the Laplace approximation becomes $p(\boldsymbol{\theta}|\mathcal{D}) \approx (\mathcal{N}|\boldsymbol{\theta}^*, \boldsymbol{\Sigma}_{\text{GGN}})$, where

$$\boldsymbol{\Sigma}_{\text{GGN}}^{-1} = \sum_{n=1}^N \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n)^T \mathbf{\Lambda}(\mathbf{y}_n; \mathbf{f}_n) \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n) + \mathbf{S}_0^{-1} \quad (17.16)$$

Unfortunately inverting this matrix takes $O(P^3)$ time, so for models with many parameters, further approximations are usually used. The simplest is to use a diagonal approximation, which takes $O(P)$ 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.

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 17.3.3) can provide more accurate approximations for comparable cost.

17.3.3 Variational inference

In fixed-form variational inference (Section 10.2), we choose a distribution for the posterior approximation $q_{\psi}(\boldsymbol{\theta})$ and minimize $D_{\text{KL}}(q \parallel p)$, with respect to ψ . We often choose a Gaussian approximate posterior, $q_{\psi}(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, which lets us use the reparameterization trick to create a low variance estimator of the gradient of the ELBO (see Section 10.2.1). 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 17.3.2).

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 17.3. 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**). This is essentially identical to the SVI algorithm in Algorithm 10.2, except the likelihood becomes $p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$ from the DNN, and the prior $p_{\xi}(\boldsymbol{\theta})$ and variational posterior $q_{\psi}(\boldsymbol{\theta})$ are Gaussians.

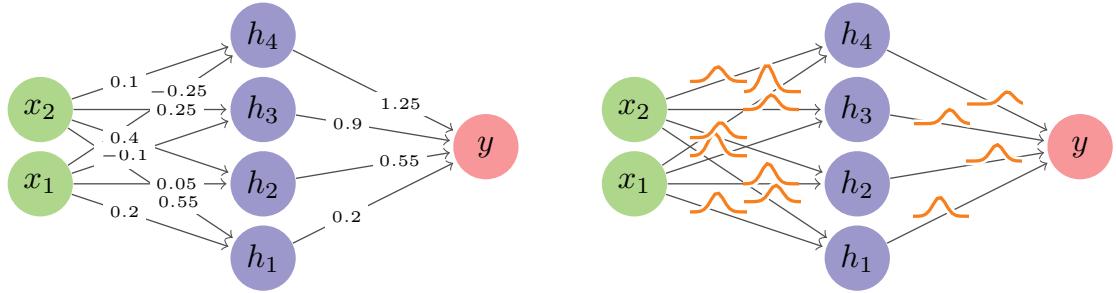


Figure 17.3: Illustration of an MLP with (left) a point estimate for each weight, (right) a marginal distribution for each weight, corresponding to a fully factored posterior approximation.

Many extensions of the BBB have been proposed. In [KSW15], they propose the **local reparameterization trick**, that samples the activations $\mathbf{a} = \mathbf{W}\mathbf{z}$ at each layer, instead of the weights \mathbf{W} , which results in a lower variance estimate of the ELBO gradient. In [Osa+19a], they used the **variational online Gauss-Newton (VOGN)** method of [Kha+18], for improved scalability. VOGN is a noisy version of natural gradient descent, where the extra noise emulates the effect of variational inference. In [Mis+18], they replaced the diagonal approximation with a low-rank plus diagonal approximation, and used VOGN for fitting. In [Tra+20b], they use a rank-one plus diagonal approximation known as **NAGVAC** (see Section 10.2.1.3). In this case, there are only 3 times as many parameters as when computing a point estimate (for the variational mean, variance, and rank-one vector), making the approach very scalable. In addition, in this case it is possible to analytically compute the natural gradient, which speeds up model fitting (see Section 6.4). Many other variational methods have also been proposed (see e.g., [LW16; Zha+18; Wu+19a; HHK19]). See also Section 17.5.4 for a discussion of online VI for DNNs.

17.3.4 Expectation propagation

Expectation propagation (EP) is similar to variational inference, except it locally optimizes $D_{\text{KL}}(p \parallel q)$ instead of $D_{\text{KL}}(q \parallel p)$, where p is the exact posterior and q is the approximate posterior. For details, see Section 10.7.

A special case of EP is the assumed density filtering (ADF) algorithm of Section 8.6, which is equivalent to the first pass of ADF. In Section 8.6.3 we show how to apply ADF to online logistic regression. In [HLA15a], they extend ADF to the case of BNNs; they called their method probabilistic backpropagation or **PBP**. They approximate every parameter in the model by a Gaussian factor, as in Figure 17.3. See Section 17.5.3 for the details.

17.3.5 Last layer methods

A very simple approximation to the posterior is to only “be Bayesian” about the weights in the final layer, and to use MAP estimates for all the other parameters. This is called the **neural-linear** approximation [RTS18]. In more detail, let $\mathbf{z} = f(\mathbf{x}; \boldsymbol{\theta})$ be the predicted outputs (e.g., logits) of the model before any optional final nonlinearity. We assume this has the form $\mathbf{z} = \mathbf{w}_L^\top \phi(\mathbf{x}; \boldsymbol{\theta})$,

where $\phi(\mathbf{x})$ are the features extracted by the first $L - 1$ layers. This gives us a Bayesian GLM. We can use standard techniques, such as the Laplace approximation (Section 15.3.5), to compute $p(\mathbf{w}_L | \mathcal{D}) = \mathcal{N}(\boldsymbol{\mu}_L, \boldsymbol{\Sigma}_L)$, given $\phi()$. To estimate the parameters of the feature extractor, we can optimize the log-likelihood in the usual way. Given the posterior over the last layer weights, we can compute the posterior predictive distribution over the logits using

$$p(\mathbf{z} | \mathbf{x}, \mathcal{D}) = \mathcal{N}(\mathbf{z} | \boldsymbol{\mu}_L \phi(\mathbf{x}), \phi(\mathbf{x}) \boldsymbol{\Sigma}_L \phi(\mathbf{x})^\top) \quad (17.17)$$

This can be passed through the final softmax layer to compute $p(\mathbf{y} | \mathbf{x}, \mathcal{D})$ as described in Section 15.3.6.

In [KHH20] they show this can reduce overconfidence in predictions for inputs that are far from the training data. However, this approach ignores uncertainty introduced by the earlier feature extraction layers, where most of the parameters reside. We discuss a solution to this in Section 17.3.6.

17.3.6 SNGP

It is possible to combine DNNs with Gaussian process (GP) models (Chapter 18), by using the DNN to act as a feature extractor, which is then fed into the kernel in the final layer. This is called “deep kernel learning” (see Section 18.6.6).

One problem with this is that the feature extractor may lose information which is not needed for classification accuracy, but which is needed for robust performance on out-of-distribution inputs (see Section 17.4.6.2). The basic problem is that, in a classification problem, there is no reduction in training accuracy (log likelihood) if points which are far away are projected close together, as long as they are on the correct side of the decision boundary. Thus the distances between two inputs can be erased by the feature extraction layers, so that OOD inputs appear to the final layer to be close to the training set.

One solution to this is to use the **SNGP** (spectrally normalized Gaussian process) method of [Liu+20d; Liu+22a]. This constrains the feature extraction layers to be “distance preserving”, so that two inputs that are far apart in input space remain far apart after many layers of feature extraction, by using spectral normalization of the weights to bound the Lipschitz constant of the feature extractor. The overall approach ensures that information that is relevant for computing the confidence of a prediction, but which might be irrelevant to computing the label of a prediction, is not lost. This can help performance in tasks such as out-of-distribution detection (Section 17.4.6.2).

17.3.7 MCMC methods

Some of the earliest work on inference for BNNs was done by Radford Neal, who proposed to use Hamiltonian Monte Carlo (Section 12.5) to approximate the posterior [Nea96]. This is generally considered the gold standard method, since it does not make strong assumptions about the form of the posterior. For more recent work on scaling up HMC for BNNs, see e.g., [Izm+21b; CJ21].

We give a simple example of vanilla HMC in Figure 17.4, where we fit a shallow MLP to a small 2d binary dataset. We plot the mean and standard deviation of the posterior predictive distribution, $p(y = 1 | \mathbf{x}; \mathcal{D})$. We see that the uncertainty is higher as we move away from the training data. (Compare to Bayesian logistic regression in 1d in Figure 15.8a.)

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, such as

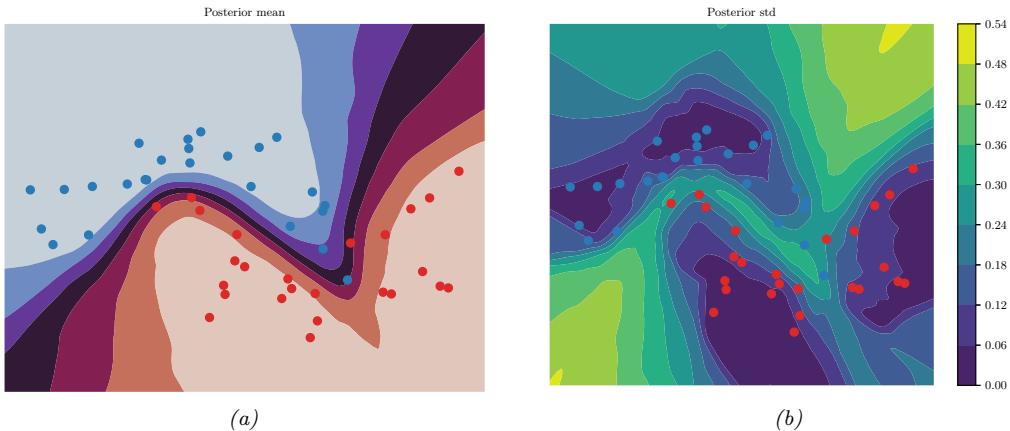


Figure 17.4: Illustration of an MLP fit to the two-moons dataset using HMC. (a) Posterior mean. (b) Posterior standard derivation. The uncertainty increases as we move away from the training data. Generated by [bnn_mlp_2d_hmc.ipynb](#).

SGLD, operate instead using mini-batches of data, offering a scalable alternative, as we discuss in Section 12.7.1. For an example of SGLD applied to an MLP, see Section 19.3.3.1.

17.3.8 Methods based on the SGD trajectory

In [MHB17; SL18; CS18], it was shown that, under some assumptions, the iterates produced by stochastic gradient descent (SGD), when run at a fixed learning rate, correspond to samples from a Gaussian approximation to the posterior centered at a local mode, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}, \Sigma)$. We can therefore use SGD to generate approximate posterior samples. This is similar to SG-MCMC methods, except we do not add 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 17.5. 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 $\bar{\boldsymbol{\theta}} = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s$. 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 (see Figure 17.10), 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 $\boldsymbol{\theta}_s$ after increasing and decreasing the learning rate multiple times in a cyclical fashion, and then computing the *average of 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 computing the *average of the parameters* and

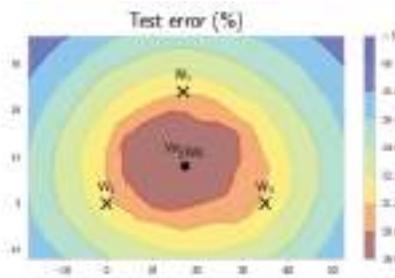


Figure 17.5: Illustration of stochastic weight averaging (SWA). The three crosses represent different SGD solutions. The star in the middle is the average of these parameter values. From Figure 1 of [Izm+18]. Used with kind permission of Andrew Wilson.

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.

17.3.9 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,

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}_m) \quad (17.18)$$

where M is the number of models, and $\hat{\boldsymbol{\theta}}_m$ is the MAP estimate for model m . See Figure 17.6 for a

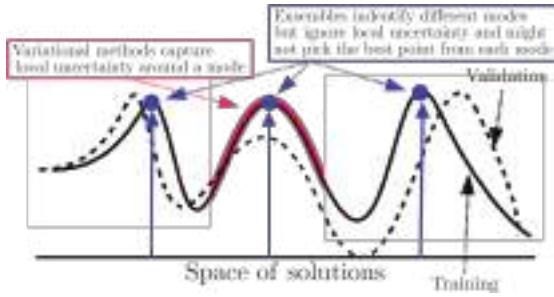


Figure 17.6: Cartoon illustration of the NLL as it varies across the parameter space. Subspace methods (red) model the local neighborhood around a local mode, whereas ensemble methods (blue) approximate the posterior using a set of distinct modes. From Figure 1 of [FHL19]. Used with kind permission of Balaji Lakshminarayanan.

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. In addition, [DF21; TB22] discusses how to add an explicit repulsive term to ensure functional diversity between the ensemble members. This way, each member corresponds to a distinct prediction function. Combining these is more effective than combining multiple samples from the same basin of attraction, especially in the presence of dataset shift [Ova+19].

17.3.9.1 Multi-SWAG

We can further improve on this approach by fitting a Gaussian to each local mode using the SWAG method from Section 17.3.8 to get a mixture of Gaussians approximation:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}_m, \Sigma_m) \quad (17.19)$$

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.

17.3.9.2 Deep ensembles with random priors

The standard way to fit each member of a deep ensemble is to initialize them each with a different random set of parameters, but them to train them all on the same data. Unfortunately this can result in the predictions from each ensemble member being rather similar, which reduces the benefit of the approach. One way to increase diversity is to train each member on a different subset of the data; this is called **bootstrap sampling**. Another approach is to define the i 'th ensemble member $g_i(\mathbf{x})$ to be the addition of a trainable model $t_i(\mathbf{x})$ and a fixed, but random, **prior network**, $p_i(\mathbf{x})$, to get

$$g_i(\mathbf{x}; \boldsymbol{\theta}_i) = t_i(\mathbf{x}; \boldsymbol{\theta}_i) + \beta p_i(\mathbf{x}) \quad (17.20)$$

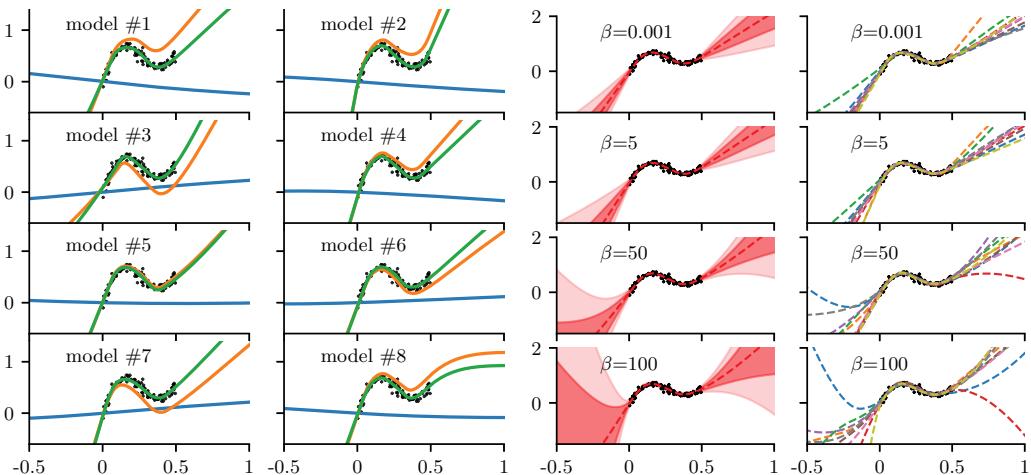


Figure 17.7: Deep ensemble with random priors. (a) Individual predictions from each member. Blue is the fixed random prior function, orange is the trainable function, green is the combination of the two. (b) Overall prediction from the ensemble, for increasingly large values of β . On the left we show (in red) the posterior mean and pointwise standard deviation, and on the right we show samples from the posterior. As β increases, we trust the random priors more, and pay less attention to the data, thus getting a more diffuse posterior. Generated by [randomized_priors.ipynb](#).

where $\beta \geq 0$ controls the amount of data-independent variation between the members. The trainable network learns to model the residual error between the true output and the value predicted by the prior. This is called a **random prior deep ensemble** [OAC18]. See Figure 17.7 for an illustration.

17.3.9.3 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. 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.

17.3.9.4 Deep ensembles vs classical ensembles

Note that deep ensembles are slightly different from classical ensemble methods (see e.g., [Die00]), such as bagging and random forests, which obtain diversity of their 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 input features. The diversity arises due to different starting parameters, different

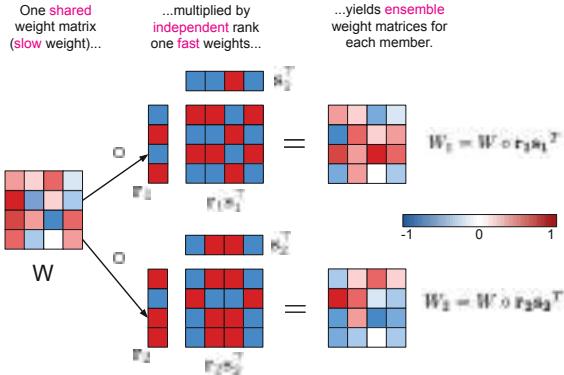


Figure 17.8: Illustration of batch ensemble with 2 ensemble members. From Figure 2 of [WTB20]. Used with kind permission of Paul Vicol.

random seeds, and SGD noise, which induces different solutions due to the nonconvex loss. It is also possible to explicitly enforce diversity of the ensemble members, which can provably improve performance [TB22].

17.3.9.5 Deep ensembles vs mixtures of experts and stacking

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. By contrast, in deep ensembles, we always use M equally weighted models. Thus we see that Bayes model averaging is not the same as model ensembling [Min00b]. Indeed, ensembling 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:

$$p(y|\mathbf{x}, \mathcal{D}) = \sum_m w_m(\mathbf{x}) p(y|\mathbf{x}, \boldsymbol{\theta}_m) \quad (17.21)$$

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.

17.3.9.6 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 17.8. 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 l , 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 :

$$y_n^m = \varphi(\mathbf{W}_m^\top \mathbf{x}_n) = \varphi((\mathbf{W} \odot \mathbf{s}_m \mathbf{r}_m^\top)^\top \mathbf{x}_n) = \varphi((\mathbf{W}^\top (\mathbf{x}_n \odot \mathbf{s}_m) \odot \mathbf{r}_m)) \quad (17.22)$$

We can vectorize this for a minibatch of inputs \mathbf{X} by replicating \mathbf{r}_m and \mathbf{s}_m along the B rows in the batch to form matrices, giving

$$\mathbf{Y}_m = \varphi(((\mathbf{X} \odot \mathbf{S}_m) \mathbf{W}) \odot \mathbf{R}_m) \quad (17.23)$$

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.

17.3.10 Approximating the posterior predictive distribution

Once we have approximated the parameter posterior, $q(\boldsymbol{\theta}) \approx p(\boldsymbol{\theta}|\mathcal{D})$, we can use it to approximate the posterior predictive distribution:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int q(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) d\boldsymbol{\theta} \quad (17.24)$$

We often approximate this integral using Monte Carlo:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^s)) \quad (17.25)$$

where $\boldsymbol{\theta}^s \sim q(\boldsymbol{\theta}|\mathcal{D})$. We discuss some extensions of this approach below.

17.3.10.1 A linearized approximation

In [IKB21] they point out that samples from an approximate posterior, $q(\boldsymbol{\theta})$, can result in bad predictions when plugged into the model if the posterior puts probability density “in the wrong places”. This is because $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ is a highly nonlinear function of $\boldsymbol{\theta}$ that might behave quite differently when $\boldsymbol{\theta}$ is far from the MAP estimate on which $q(\boldsymbol{\theta})$ is centered. To avoid this problem, they propose to replace $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ with a linear approximation centered at the MAP estimate $\boldsymbol{\theta}^*$:

$$\mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^*) + \mathbf{J}(\mathbf{x})(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \quad (17.26)$$

where $\mathbf{J}_{\theta^*}(\mathbf{x}) = \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta}|_{\theta^*}$ is the $P \times C$ Jacobian matrix, where P is the number of parameters, and C is the number of outputs. Such a model is well behaved around θ^* , and so the approximation

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{f}_{\text{lin}}^{\theta^*}(\mathbf{x}, \theta^s)) \quad (17.27)$$

often works better than Equation (17.25).

Note that $\mathbf{z} = \mathbf{f}_{\text{lin}}^{\theta^*}(\mathbf{x}, \theta)$ is a linear function of the parameters θ , but a nonlinear function of the inputs \mathbf{x} . Thus $p(\mathbf{y}|\mathbf{f}_{\text{lin}}^{\theta^*}(\mathbf{x}, \theta))$ is a generalized linear model (Section 15.1), so [IKB21] call this approximation the **GLM predictive distribution**.

If we have a Gaussian approximation to the parameter, $p(\theta|\mathcal{D}) \approx \mathcal{N}(\theta|\mu, \Sigma)$, then we can “push this through” the linear approximation to get

$$p(\mathbf{z}|\mathbf{x}, \mathcal{D}) \approx \mathcal{N}(\mathbf{z}|\mathbf{f}(\mathbf{x}, \mu), \mathbf{J}(\mathbf{x})^\top \Sigma \mathbf{J}(\mathbf{x})) \quad (17.28)$$

where \mathbf{z} are the logits. (Alternatively, we can use the last layer method of Equation (17.17) to get a Gaussian approximation to $p(\mathbf{z}|\mathbf{x}, \mathcal{D})$.) If we approximate the final softmax layer with a probit function, we can analytically pass this Gaussian through the final softmax layer to deterministically compute the predictive probabilities $p(y=c|\mathbf{x}, \mathcal{D})$, using Equation (15.150). Alternatively, we can use the Laplace bridge approximation in Section 17.3.10.2.

17.3.10.2 The Laplace bridge approximation

Just using a point estimate of the probability of each class label, $p_c = p(y=c|\mathbf{x}, \mathcal{D})$, can be unreliable, since it does not convey any sense of uncertainty in the probability value, even though we may have taken the uncertainty of the parameters into account (e.g., using the methods of Section 17.3.10.1). An alternative is to represent the output over labels as a Dirichlet distribution, $\text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha})$, rather than a categorical distribution, $\text{Cat}(\mathbf{y}|\mathbf{p})$, where $\mathbf{p} = \text{softmax}(\mathbf{z})$. This is more appropriate if we view each datapoint as being annotated with a “soft” vector of probabilities (e.g., representing consensus votes from human raters), rather than a one-hot encoding with a single “ground truth” value. This can be useful for settings where the true label is ambiguous (see e.g., [Bey+20; Dum+18]).

We can either train the model to predict the Dirichlet parameters directly (as in the **prior network** approach of [MG18]), or we can train the model to predict softmax outputs in the usual way, and then derive the Dirichlet parameters from a Gaussian approximation to the posterior. The latter approach is known as the **Laplace bridge** [HKh22], and has the advantage that it can be used as a post-processing method. It works as follows. First we compute a Gaussian approximation to the logits, $p(\mathbf{z}|\mathbf{x}, \mathcal{D}) = \mathcal{N}(\mathbf{z}|\mathbf{m}, \mathbf{V})$ using Equation (17.28) or Equation (17.17). Then we compute

$$\alpha_i = \frac{1}{V_{ii}} \left(1 - \frac{2}{C} + \frac{\exp(m_i)}{C^2} \sum_{j=1}^C \exp(-m_j) \right) \quad (17.29)$$

where C is the number of classes. We can then derive the probabilities of each class label using $p_c = \mathbb{E}[\pi_c] = \alpha_c/\alpha_0$, where $\alpha_0 = \sum_{c=1}^C \alpha_c$.

Note that the derivation of the above result assumes that the Gaussian terms sum to zero, since the Gaussian has one less degree of freedom compared to the Dirichlet. To ensure this, it is necessary

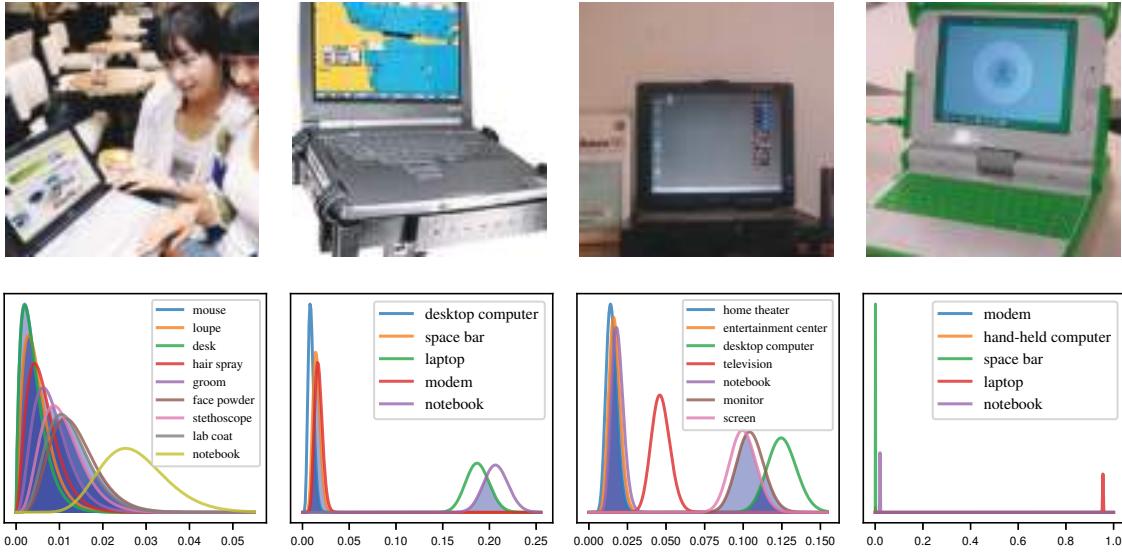


Figure 17.9: Illustration of uncertainty about individual labels in an image classification problem. Top row: images from the “laptop” class of ImageNet. Bottom row: beta marginals for the top- k predictions for the respective image. First column: high uncertainty about all the labels. Second column: “notebook” and “laptop” have high confidence. Third column: “desktop”, “screen” and “monitor” have high confidence. Fourth column: only “laptop” has high confidence. (Compare to Figure 14.4.) From Figure 6 of [HKH22]. Used with kind permission of Philipp Hennig.

to first project the Gaussian distribution onto this constraint surface, yielding

$$p(\mathbf{z}|\mathbf{x}, \mathcal{D}) = \mathcal{N}\left(\mathbf{z}|\mathbf{m} - \frac{\mathbf{V}\mathbf{1}\mathbf{1}^\top\mathbf{m}}{\mathbf{1}^\top\mathbf{V}_*\mathbf{1}}, \mathbf{V} - \frac{\mathbf{V}\mathbf{1}\mathbf{1}^\top\mathbf{V}}{\mathbf{1}^\top\mathbf{V}\mathbf{1}}\right) = \mathcal{N}(\mathbf{z}|\mathbf{m}', \mathbf{V}') \quad (17.30)$$

where $\mathbf{1}$ is the ones vector of size C . To avoid potential problems where $\boldsymbol{\alpha}$ is sparse, [HKH22] propose to also scale the posterior (after the zero-sum constraint) by using $\mathbf{m}'' = \mathbf{m}'/\sqrt{c}$ and $\mathbf{V}'' = \mathbf{V}'/c$, where $c = (\sum_{ii} V'_{ii})/\sqrt{C/2}$.

One useful property of the Laplace bridge approximation, compared to the probit approximation, is that we can easily compute a marginal distribution over the probability of each label being present. This is because the marginals of a Dirichlet are beta distributions. We can use this to adaptively compute a top- k prediction set; this is similar in spirit to conformal prediction (Section 14.3.1), but is Bayesian, in the sense that it represents per-instance uncertainty. The method works as follows. First we sort the class labels in decreasing order of expected probability, to get $\tilde{\boldsymbol{\alpha}}$; next we compute the marginal distribution over the probability for the top label,

$$p(\pi_1|\mathbf{x}, \mathcal{D}) = \text{Beta}(\tilde{\alpha}_1, \alpha_0 - \tilde{\alpha}_1) \quad (17.31)$$

where $\alpha_0 = \sum_c \alpha_c$. We then compute the marginal distributions for the other labels in a similar way,

and return all labels that have significant overlap with the top label. As we see from the examples in Figure 17.9, this approach can return variable-sized outputs, reflecting uncertainty in a natural way.

17.3.10.3 Distillation

The MC approximation to the posterior predictive is S times slower than a standard, deterministic plug-in approximation. One way to speed this up is to use **distillation** to approximate the semi-parametric “teacher” model p_t from Equation (17.25) by a parametric “student” model p_s by minimizing $\mathbb{E}[D_{\text{KL}}(p_t(\mathbf{y}|\mathbf{x}) \parallel p_s(\mathbf{y}|\mathbf{x}))]$ wrt p_s . This approach was first proposed in [HVD14], who called the technique “**dark knowledge**”, because the teacher has “hidden” information in its predictive probabilities (logits) than is not apparent in the raw one-hot labels.

In [Kor+15], this idea was used to distill the predictions from a teacher whose parameter posterior was computed using HMC; this is called “**Bayesian dark knowledge**”. A similar idea was used in [BPK16; GBP18], who distilled the predictive distribution derived from MC dropout (Section 17.3.1).

Since the parametric student is typically less flexible than the semi-parametric teacher, it may be overconfident, and lack diversity in its predictions. To avoid this overconfidence, it is safer to make the student be a mixture distribution [SG05; Tra+20a].

17.3.11 Tempered and cold posteriors

When working with BNNs for classification problems, the likelihood is usually taken to be

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\mathbf{x}; \boldsymbol{\theta}))) \quad (17.32)$$

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 logistic regression (Section 15.3.2); the only difference is that f is a nonlinear function of $\boldsymbol{\theta}$.

However, in practice, it is often found (see e.g., [Zha+18; Wen+20b; LST21; Noc+21]) that BNNs give better predictive accuracy if the likelihood function is scaled by some power α . That is, instead 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**, $p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})^\alpha p(\boldsymbol{\theta})$. In log space, we have

$$\log p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) = \alpha \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const} \quad (17.33)$$

This is also called an **α -posterior** or **power posterior** [Med+21].

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 space,

$$\log p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{T} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \frac{1}{T} \log p(\boldsymbol{\theta}) + \text{const} \quad (17.34)$$

If $T < 1$, we say that the posterior is “cold”. Note that, in the case of a Gaussian prior, using the cold posterior is the same as using the tempered posterior with a different hyperparameter, since $\frac{1}{T} \log p_{\text{cold}}(\boldsymbol{\theta})$ is given by

$$\frac{1}{T} \log \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{cold}}^2 \mathbf{I}) = -\frac{1}{2T\sigma_{\text{cold}}^2} \sum_i \theta_i^2 + \text{const} = \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{tempered}}^2 \mathbf{I}) + \text{const} \quad (17.35)$$

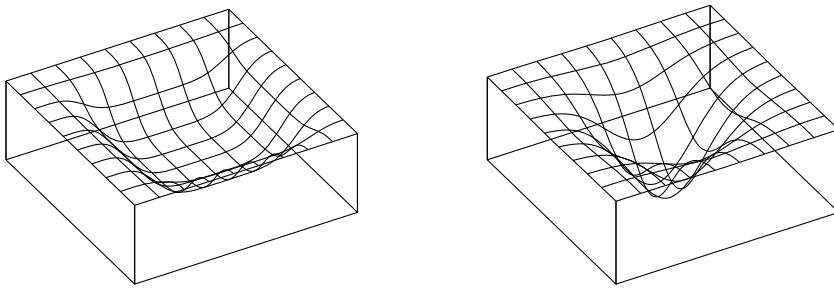


Figure 17.10: Flat vs sharp minima. From Figures 1 and 2 of [HS97]. Used with kind permission of Jürgen Schmidhuber.

which equals $\log p_{\text{tempered}}(\boldsymbol{\theta})$ if we set $\sigma_{\text{tempered}}^2 = T\sigma_{\text{cold}}^2$. Thus both methods are effectively the same, and just reweight the likelihood by $\alpha = 1/T$.

Cold posteriors in Bayesian neural network classifiers are a consequence of underrepresenting aleatoric (label) uncertainty, as shown by [Kap+22]. On benchmarks such as CIFAR-100, we should have essentially no uncertainty about the labels of the training images, yet Bayesian classifiers with softmax likelihoods have very high uncertainty for these points. Moreover, [Izm+21b] showed that the cold posterior effect in all the examples of [Wen+20b] when data augmentation is removed. [Kap+22] show that with the SGLD inference in [Wen+20b], data augmentation has the effect of raising the likelihood to a power $1/K$ for minibatches of size K . Cold posteriors exactly counteract this effect, more honestly representing our beliefs about aleatoric uncertainty, by sharpening the likelihood. However, tempering is not required, and [Kap+22] show that by using a Dirichlet observation model to explicitly represent (lack of) label noise, there is no cold posterior effect, even with data augmentation. The curation hypotheses of [Ait21] can be considered a special case of the above explanation, where curation has the effect of increasing our confidence about training labels.

In Section 14.1.3, we discuss generalized variational inference, which gives a general framework for understanding whether and how the likelihood or prior could benefit from tempering. Tempering is particularly useful if (as is usually the case) the model is misspecified [KJD21].

17.4 Generalization in Bayesian deep learning

In this section, we discuss why ‘‘being Bayesian’’ can improve predictive accuracy and generalization performance.

17.4.1 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 17.10(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 17.10(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 3.8.1. 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 Section 17.3.8). The SWA method (Section 17.3.8) can be seen as finding a center of mass in the posterior based on these SGD samples, finding solutions that generalize better than picking a single SGD point.

If we must use a single solution, a flat one will help us better approximate the Bayesian model average in the integral of Equation (17.1). However, by attempting to perform a more complete Bayesian model average, we will select for flatness without having to deal with the messiness of having to worry about flatness definitions, or the effects of reparameterization, or unknown implicit regularization, as the model average will automatically weight regions with the greatest volume.

17.4.2 Mode connectivity and the loss landscape

In DNNs there are often many low-loss solutions, which provide complementary explanations of the data. Moreover, in [Gar+18c] they showed that two independently trained SGD solutions can be connected by a curve in a subspace, along which the training loss remains near-zero, known as **mode connectivity**. Despite having the same training loss, these different parameter settings give rise to very different functions, as illustrated in Figure 17.11, where we show predictions on a 1d regression problem coming from different points in parameter space obtained by interpolating along a mode connecting curve between two distinct MAP estimates. Using a Bayesian model average, we can combine these functions together to provide much better performance over a single flat solution [Izm+19].

Recently, it has been discovered [Ben+21b] that there are in fact large multidimensional simplexes of low loss solutions, which can be combined together for significantly improved performance. These results further motivate the Bayesian approach (Equation (17.1)), where we perform a posterior weighted model average.

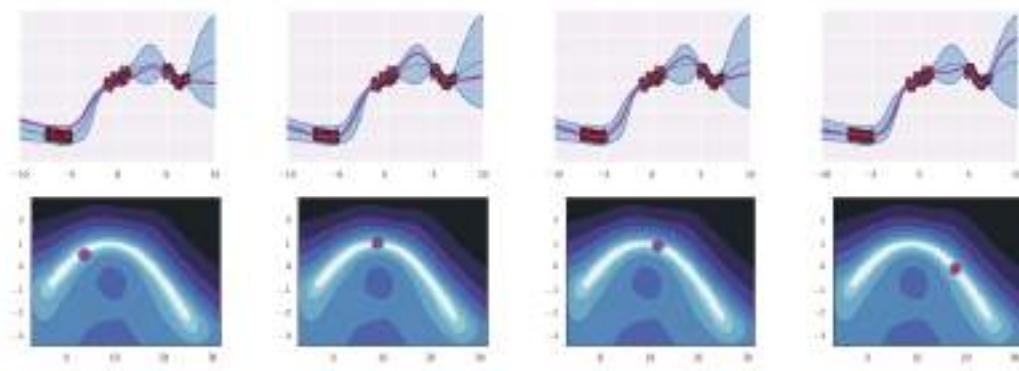


Figure 17.11: Diversity of high performing functions sampled from the posterior. Top row: we show predictions on the 1d input domain for 4 different functions. We see that they extrapolate in different ways outside of the support of the data. Bottom row: we show a 2d subspace spanning two distinct modes (MAP estimates), and connected by a low-loss curved path computed as in [Gar+18c]. From Figure 8 of [WI20]. Used with kind permission of Andrew Wilson.

17.4.3 Effective dimensionality of a model

Modern DNNs have millions of parameters, but these parameters are often not well-determined by the data, i.e., there can be a lot of posterior uncertainty. By averaging over the posterior, we reduce the chance of overfitting, because we do not use “degrees of freedom” that are not needed or warranted.

To quantify the number of degrees of freedom, or **effective dimensionality** [Mac92b], we follow [MBW20] and define

$$N_{\text{eff}}(\mathbf{H}, c) = \sum_{i=1}^k \frac{\lambda_i}{\lambda_i + c}, \quad (17.36)$$

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 17.12 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

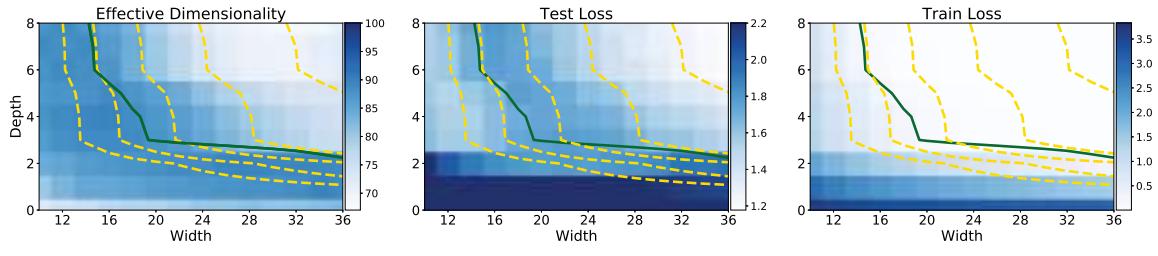


Figure 17.12: Left: effective dimensionality as a function of model width and depth for a CNN on CIFAR-100. Center: test loss as a function of model width and depth. Right: train loss as a function of model width and depth. Yellow level curves represent equal parameter counts (1.6×10^6 , 2×10^5 , 4×10^4 , 1.6×10^4). 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.

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.

17.4.4 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

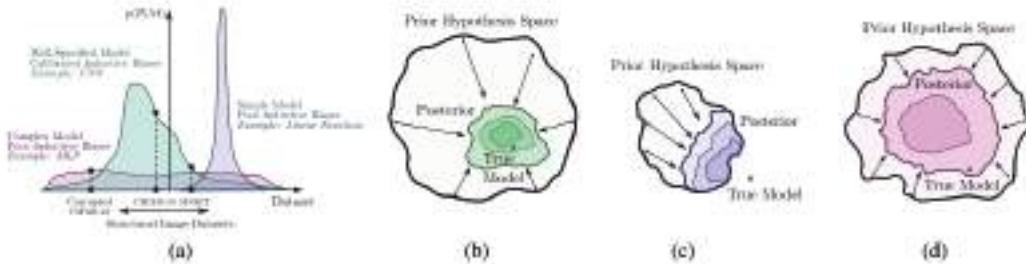


Figure 17.13: Illustration of the behavior of different kinds of model families and the prior distributions they induce over datasets. (a) The purple model is a simple linear model that has small support, and can only represent a few kinds of datasets. The pink model is an unstructured MLP: this has support over a large range of datasets with a fairly uninformative (broad) prior. Finally the green model is a CNN; this has support over a large range of datasets but the prior is more concentrated on certain kinds of datasets that have compositional structure. (b) The posterior for the green model (CNN) rapidly collapses to the true model, since it is consistent with the data. (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.

seeing a small amount of data. This idea is sketched in Figure 17.13.

17.4.5 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

$$\sqrt{\frac{D_{\text{KL}}(Q \parallel P) + c}{2(N-1)}}, \quad (17.37)$$

where c is a constant, N is the number of training points, P is the prior distribution over the 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 loose, 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.

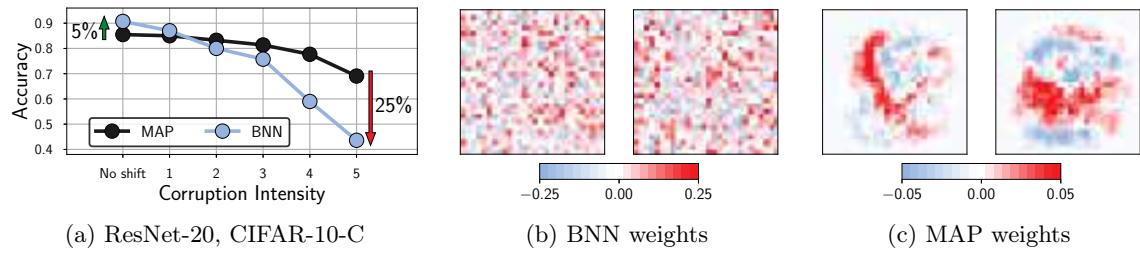


Figure 17.14: Bayesian neural networks under covariate shift. a: Performance of a ResNet-20 on the pixelate corruption in CIFAR-10-C. For the highest degree of corruption, a Bayesian model average underperforms a MAP solution by 25% (44% against 69%) accuracy. See Izmailov et al. [Izm+21b] for details. 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 is Figure 1 of Izmailov et al. [Izm+21a].

17.4.6 Out-of-distribution generalization for BNNs

Bayesian methods are often assumed to be more robust in the context of distribution shift (discussed in Chapter 19), because they capture more uncertainty than methods based on point estimation. However, there are some subtleties, some of which we discuss below.

17.4.6.1 BMA can give poor results with default priors

Many approximate inference methods, especially deep ensembles, are significantly less overconfident (more well calibrated) in the presence of some kinds of covariate shifts [Ova+19]. However, in [Izm+21b], it was noted that HMC, which arguably offers the most accurate approximation to the posterior, often works poorly under distribution shift.

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, and a poor choice of prior. As an illustrative special case, MNIST digits all have black corner pixels. Weights in the first layer of a neural network connected to these pixels are multiplied by zero, and thus can take any value without affecting the outputs of the network. Classical MAP training or deep ensembles of MAP solutions with a Gaussian prior will therefore drive these parameters to zero, since they don't help with the data fit, and the resulting network will be robust to corruptions on these pixels. On the other hand, the posterior for these parameters will be the same as the prior, and so a Bayesian model average will multiply corruptions by random numbers sampled from the prior, leading to degraded predictive performance.

Figure 17.14(b, c) visualizes this example, showing the first-layer weights of a fully-connected network for the MAP solution and a BNN posterior sample, on MNIST. The MAP weights corresponding to zero intensity pixels near the boundary are near zero, while the BNN weights look noisy, sampled from a Gaussian prior.

Izmailov et al. [Izm+21a] prove that this issue is a special case of a much more general problem, whenever there are linear dependencies in the input features of the training data, both for fully-

connected and convolutional networks. In this case, the data live on a hyperplane. If a covariate or domain shift, moves orthogonal to this hyperplane, the posterior will be the same as the prior in the direction of the shift. The posterior model average will thus be highly vulnerable to shifts that do not particularly affect the underlying semantic structure of the problem (such as corruptions), whereas the MAP solution will be entirely robust to such shifts.

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. With this improved prior they are able to obtain a method that is much more robust to distribution shift.

17.4.6.2 BNNs can be overconfident on OOD inputs

An important problem in practice is how a predictive model will behave when it is given an input that is “out of distribution” or OOD. Ideally we would like the model to express that it is not confident in its prediction, so that the system can abstain from predicting (see Section 19.3.3). Using “exact” inference methods, such as MCMC, for BNNs can give this behavior in some cases. For example, in Section 19.3.3.1 we showed that an MLP which was fit to MNIST using SGLD would be less overconfident than a point estimate (computed using SGD) when presented with inputs from fashion MNIST. However, this behavior does not always occur reliably.

To illustrate the problem, consider the 2d nonlinear binary classification dataset shown in Figure 17.15. In addition to the two training classes, we have highlighted (in green) a set of OOD inputs that are far from the support of the training set. Intuitively we would expect the model to predict a probability of 0.5 (corresponding to “don’t know”) for such inputs that are far from the training set. However we see that the only methods that do so are the Gaussian process (GP) classifier (see Section 18.4) and the SNGP model (Section 17.3.6), which contains a GP layer on top of the feature extractor.

The lesson we learn from this simple example is that “being Bayesian” only helps if we are using a good hypothesis class. If we only consider a single MLP classifier, with standard Gaussian priors on the weights, it is extremely unlikely that we will learn the kind of compact decision boundary shown in Figure 17.15g, because that function has negligible support under our prior (c.f. Section 17.4.4). Instead we should embrace the power of Bayes to avoid overfitting and use as complex a model class as we can afford.

17.4.7 Model selection for BNNs

Historically, the marginal likelihood (aka Bayesian evidence) has been used for model selection problems, such as choosing neural architectures or hyperparameter values [Mac92a]. Recent methods based on the Laplace approximation, such as [Imm+21; Dax+21], have made this scalable to large BNNs. However, [Lot+22] argue that it is much better to use the conditional marginal likelihood, which we discuss in Section 3.8.5.

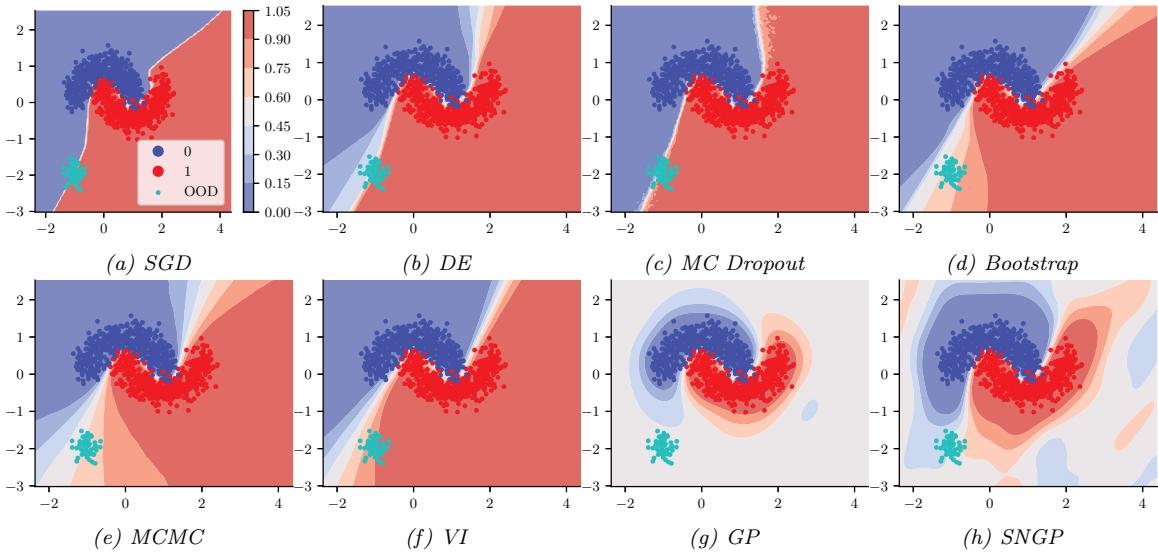


Figure 17.15: Predictions made by various (B)NNs when presented with the training data shown in blue and red. The green blob is an example of some OOD inputs. Methods are: (a) standard SGD; (b) deep Ensemble of 10 models with different random initializations; (c) MC dropout with 50 samples; (d) bootstrap training, where each of the 10 models is initialized identically but given different versions of the data, obtained by resampling with replacement; (e) MCMC using NUTS algorithm with 3000 warmup steps and 3000 samples; (f) variational inference; (g) Gaussian process classifier using RBF kernel; (h) SNGP. The model is an MLP with 8,16,16,8 units in the hidden layers and ReLu activation. The output layer has 1 neuron with sigmoid activation. Generated by [makemoons_comparison.ipynb](#)

17.5 Online inference

In Section 17.3, we have focused on batch or offline inference. However, an important application of Bayesian inference is in sequential settings, where the data arrives in a continuous stream, and the model has to “keep up”. This is called **sequential Bayesian inference**, and is one approach to **online learning** (see Section 19.7.5). In this section, we discuss some algorithmic approaches to this problem in the context of DNNs. These methods are widely used for continual learning, which we discuss Section 19.7.

17.5.1 Sequential Laplace for DNNs

In [RBB18b], they extended the Laplace method of Section 17.3.2 to the sequential setting. Specifically, 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 the precision matrix is Kronecker factored. We now compute the new mean by solving the MAP

problem

$$\boldsymbol{\mu}_t = \operatorname{argmax} \log p(\mathcal{D}_t | \boldsymbol{\theta}) + \log p(\boldsymbol{\theta} | \mathcal{D}_{t-1}) \quad (17.38)$$

$$= \operatorname{argmax} \log p(\mathcal{D}_t | \boldsymbol{\theta}) - \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \boldsymbol{\Lambda}_{t-1}^{-1} (\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \quad (17.39)$$

Once we have computed $\boldsymbol{\mu}_t$, we compute the approximate Hessian at this point, and get the new posterior precision

$$\boldsymbol{\Lambda}_t = \lambda \mathbf{H}(\boldsymbol{\mu}_t) + \boldsymbol{\Lambda}_{t-1} \quad (17.40)$$

where $\lambda \geq 0$ is a weighting factor that trades off how much the model pays attention to the new data vs old data.

Now suppose we use a diagonal approximation to the posterior prediction matrix. From Equation (17.39), we see that this amounts to adding a quadratic penalty to each new MAP estimate, to encourage it to remain close to the parameters from previous tasks. This approach is called **elastic weight consolidation (EWC)** [Kir+17].

17.5.2 Extended Kalman filtering for DNNs

In Section 29.7.2, we showed how Kalman filtering can be used to incrementally compute the exact posterior for the weights of a linear regression model with known variance, i.e., we compute $p(\boldsymbol{\theta} | \mathcal{D}_{1:t}, \sigma^2)$, where $\mathcal{D}_{1:t} = \{(\mathbf{u}_i, y_i) : i = 1 : t\}$ is the data seen so far, and

$$p(y_t | \mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t | \boldsymbol{\theta}^\top \mathbf{u}_t, \sigma^2) \quad (17.41)$$

is the linear regression likelihood. The application of KF to this model is known as recursive least squares.

Now consider the case of nonlinear regression:

$$p(y_t | \mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t | f(\boldsymbol{\theta}, \mathbf{u}_t), \sigma^2) \quad (17.42)$$

where $f(\boldsymbol{\theta}, \mathbf{u}_t)$ is some nonlinear function, such as an MLP. We can use the extended Kalman filter (Section 8.3.2) 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; PF03]). To see this, note that we can set the dynamics model to the identity function, $f(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t$, so the parameters are propagated through unchanged, and the observation model to the input-dependent 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 $\mathbf{Q}_t = q\mathbf{I}$, where q is a small constant, to allow the parameters to slowly drift according to artificial **process noise**. (In practice it can be useful to anneal q from a large initial value to something near 0.)

17.5.2.1 Example

We now give an example of this process in action. We sample a synthetic dataset from the true function

$$h^*(u) = x - 10 \cos(u) \sin(u) + u^3 \quad (17.43)$$

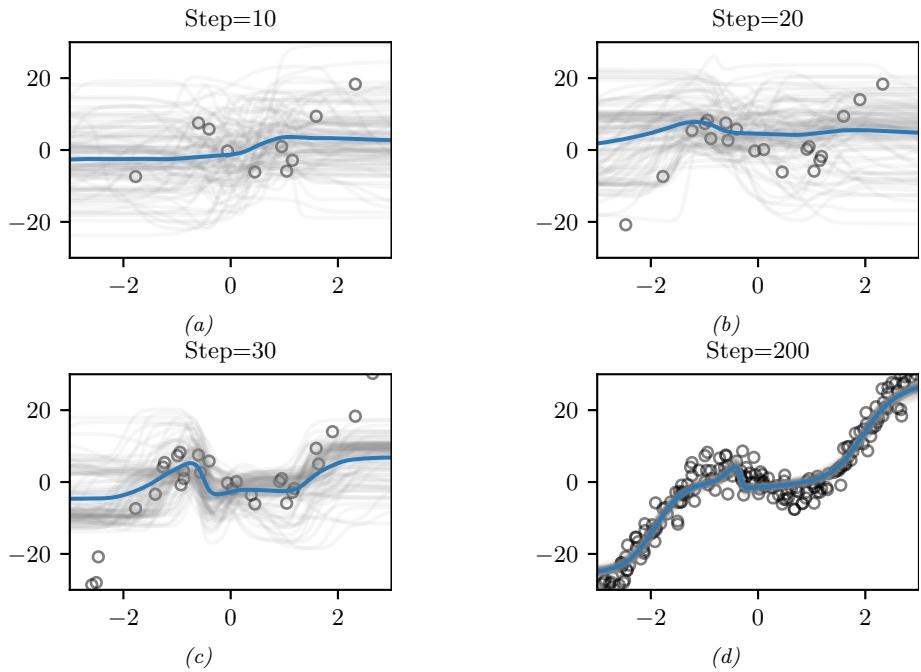


Figure 17.16: Sequential Bayesian inference for the parameters of an MLP using the extended Kalman filter. We show results after seeing the first 10, 20, 30 and 200 observations. (For a video of this, see <https://bit.ly/3wXnWaM>.) Generated by `ekf_mlp.ipynb`.

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

$$f(\boldsymbol{\theta}, \mathbf{u}) = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{u} + \mathbf{b}_1) + \mathbf{b}_2 \quad (17.44)$$

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 17.16. 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.

17.5.2.2 Setting the variance terms

In the above example, we set the variance terms by hand. In general we need to estimate the noise variance σ , which determines \mathbf{R}_t and hence the learning rate, as well as the strength of the prior $\boldsymbol{\Sigma}_0$, which controls the amount of regularization. Some methods for doing this are discussed in [FNG00].

17.5.2.3 Reducing the computational complexity

The naive EKF method described above takes $O(N_z^3)$ time, which is prohibitive for large neural networks. A simple approximation, known as the **decoupled EKF**, was proposed in [PF91; SPD92] (see [PF03] for a review). This partitions the weights into G groups or blocks, and estimates the relevant matrices for each group g independently. If $G = 1$, this reduces the standard global EKF. If we put each weight into its own group, we get a fully diagonal approximation. In practice this does not work any better than SGD, since it ignores correlations between the parameters. A useful compromise is to put all the weights corresponding to each neuron into its own group; this is called **node decoupled EKF**, which has been used in [Sim02] to train RBF networks and [GUK21] to train exponential family matrix factorization models (widely used in recommender systems). For more details on DEKF, [Supplementary](#) Section 17.1.

Another approach to increasing computational efficiency is to leverage the fact that the effective dimensionality of a DNN is often quite low (see Section 17.4.3). Indeed we can approximate the model parameters by using a low dimensional vector of coefficients that specify the point in a linear manifold corresponding to weight space; the basis set defining this linear manifold can either be chosen randomly [Li+18b; GARD18; Lar+22], or can be estimated using PCA applied to the SGD iterates [Izm+19]. We can exploit this observation to perform EKF in this low-dimensional subspace, which significantly speeds up inference, as discussed in [DMKM22].

17.5.3 Assumed density filtering for DNNs

In Section 8.6.3, we discussed how to use assumed density filtering (ADF) to perform online (binary) logistic regression. In this section, we generalize this to nonlinear predictive models, such as DNNs. The key is to perform Gaussian moment matching of the hidden activations at each layer of the model. This provides an alternative to the EKF approach in Section 17.5.2, which is based on linearization of the network.

We will assume the following likelihood:

$$p(\mathbf{y}_t | \mathbf{u}_t, \mathbf{w}_t) = \text{Expfam}(\mathbf{y}_t | \ell^{-1}(f(\mathbf{u}_t; \mathbf{w}_t))) \quad (17.45)$$

where $f(\mathbf{x}; \mathbf{w})$ is the DNN, ℓ^{-1} is the inverse link function, and $\text{Expfam}()$ is some exponential family distribution. For example, if f is linear and we are solving a binary classification problem, we can write

$$p(y_t | \mathbf{u}_t, \mathbf{w}_t) = \text{Ber}(y_t | \sigma(\mathbf{u}_t^\top \mathbf{w}_t)) \quad (17.46)$$

We discussed using ADF to fit this model in Section 8.6.3.

In [HLA15b], they propose **probabilistic backpropagation (PBP)**, which is an instance of ADF applied to MLPs. The basic idea is to approximate the posterior over the weights in each layer using a fully factorized distribution

$$p(\mathbf{w}_t | \mathcal{D}_{1:t}) \approx p_t(\mathbf{w}_t) = \prod_{l=1}^L \prod_{i=1}^{D_l} \prod_{j=1}^{D_{l-1}+1} \mathcal{N}(w_{ijl} | \mu_{ijl}^t, \tau_{ijl}^t) \quad (17.47)$$

where L is the number of layers, and D_l is the number of neurons in layer l . (The **expectation backpropagation** algorithm of [SHM14] is a special case of this, where the variances are fixed to $\tau = 1$.)

Suppose the parameters are static, so $\mathbf{w}_t = \mathbf{w}_{t-1}$. Then the new posterior, after conditioning on the t 'th observation, is given by

$$\hat{p}_t(\mathbf{w}) = \frac{1}{Z_t} p(\mathbf{y}_t | \mathbf{u}_t, \mathbf{w}) \mathcal{N}(\mathbf{w} | \boldsymbol{\mu}^{t-1}, \boldsymbol{\Sigma}^{t-1}) \quad (17.48)$$

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 the new (approximate) posterior, $p_t(\mathbf{w})$. This can be done by computing the following means and variances [Min01a]:

$$\mu_{ijl}^t = \mu_{ijl}^{t-1} + \tau_{ijl}^{t-1} \frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \quad (17.49)$$

$$\tau_{ijl}^t = \tau_{ijl}^{t-1} - (\tau_{ijl}^{t-1})^2 \left[\left(\frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \right)^2 - 2 \frac{\partial \ln Z_t}{\partial \tau_{ijl}^{t-1}} \right] \quad (17.50)$$

In the forwards pass, we compute Z_t by propagating the input \mathbf{u}_t through the model. Since we have a Gaussian distribution over the weights, instead of a point estimate, this induces an (approximately) Gaussian distribution over the values of the hidden units. For certain kinds of activation functions (such as ReLU), the relevant integrals (to compute the means and variances) can be solved analytically, as in GP-neural networks (Section 18.7). The result is that we get a Gaussian distribution over the final layer of the form $\mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\eta}_t = f(\mathbf{u}_t; \mathbf{w}_t)$ is the output of the neural network before the GLM link function induced by $p_t(\mathbf{w}_t)$. Hence we can approximate the partition function using

$$Z_t \approx \int p(\mathbf{y}_t | \boldsymbol{\eta}_t) \mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma}) d\boldsymbol{\eta}_t \quad (17.51)$$

We now discuss how to compute this integral. In the case of probit classification, 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 then use the following analytical result

$$\int \Phi(y\eta) \mathcal{N}(h|\mu, \sigma) d\eta = \Phi\left(\frac{y\mu}{\sqrt{1+\sigma}}\right) \quad (17.52)$$

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, we can use the probit approximation from Section 15.3.6. For the multiclass case, where $\mathbf{y} \in \{0, 1\}^C$ (one-hot encoding), we have $p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \text{Cat}(\mathbf{y}|\text{softmax}(\boldsymbol{\eta}))$. A variational lower bound to $\log Z_t$ for this case is given in [GDFY16].

Once we have computed Z_t , we can take gradients and update the Gaussian posterior moments, before moving to the next step.

17.5.4 Online variational inference for DNNs

A natural approach to online learning is to use variational inference, where the prior is the posterior from the previous step. This is known as **streaming variational Bayes** [Bro+13]. In more detail,

at step t , we compute

$$\psi_t = \underset{\psi}{\operatorname{argmin}} \underbrace{\mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + D_{\text{KL}}(q(\theta|\psi) \parallel q(\theta|\psi_{t-1}))}_{-\hat{L}_t(\psi)} \quad (17.53)$$

$$= \underset{\psi}{\operatorname{argmin}} \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta) + \log q(\theta|\psi) - \log q(\theta|\psi_{t-1})] \quad (17.54)$$

where $\ell_t(\theta) = -\log p(\mathcal{D}_t|\theta)$ is the negative log likelihood (or, more generally, some loss function) of the data batch at step t .

When applied to DNNs, this approach is called **variational continual learning** or **VCL** [Ngu+18]. (We discuss continual learning in Section 19.7.) An efficient implementation of this, known as **FOOB-VB** (“fixed-point operator for online variational Bayes”) is given in [Zen+21].

One problem with the VCL objective in Equation (17.53) is that the KL term can cause the model to become too sparse, which can prevent the model from adapting or learning new tasks. This problem is called **variational overpruning** [TT17]. More precisely, the reason this happens as is as follows: some weights might not be needed to fit a given dataset, so their posterior will be equal to the prior, but sampling from these high-variance weights will add noise to the likelihood; to reduce this, the optimization method will prefer to set the bias term to a large negative value, so the corresponding unit is “turned off”, and thus has no effect on the likelihood. Unfortunately, these “dead units” become stuck, so there is not enough network capacity to learn the next task.

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

$$\hat{L}_t = \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + \beta D_{\text{KL}}(q(\theta|\psi) \parallel q(\theta|\psi_{t-1})) \quad (17.55)$$

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. In particular if we use a diagonal variational posterior, this reduces to online EWC method of [Sch+18]; if we use a block-diagonal and Kronecker factored posterior, this reduces to the online structured Laplace method of [RBB18b]; and if we use a low-rank posterior precision matrix, this reduces to the SOLA method of [Yin+20].

The second step is to replace the prior and posterior by using tempering, which is useful when the model is misspecified, as discussed in Section 17.3.11. 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

$$\hat{L}_t = \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + \beta D_{\text{KL}}(q(\theta|\psi)^\lambda \parallel q(\theta|\psi_{t-1})^\lambda) \quad (17.56)$$

This can be optimized using SGD, assuming the posterior is reparameterizable (see Section 10.2.1).

17.6 Hierarchical Bayesian neural networks

In some problems, we have multiple related datasets, such as a set of medical images from different hospitals. Some aspects of the data (e.g., the shape of healthy vs diseased cells) is generally 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

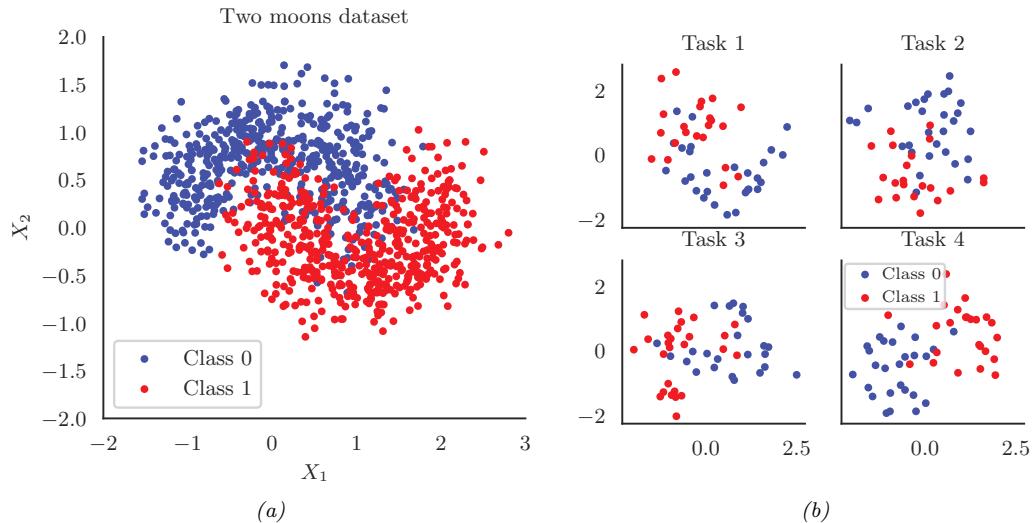


Figure 17.17: (a) Two moons synthetic dataset. (b) Multi-task version, where we rotate the data to create 18 related tasks (groups). Each dataset has 50 training and 50 test points. Here we show the first 4 tasks. Generated by [bnn_hierarchical.ipynb](#).

a common prior (to capture shared effects). This is the setup we considered in Section 15.5, where we discuss hierarchical Bayesian GLMs. In this section, we extend this to nonlinear predictors based on neural networks. (The setup is very similar to domain generalization, discussed in Section 19.6.2, except here we care about performance on all the domains, not just a held-out target domain.)

17.6.1 Example: multimoons classification

In this section, we consider an example² where we want to solve multiple related nonlinear binary classification problems coming from J different environments or distributions. We assume that each environment has its own unique decision boundary $p(y|\mathbf{x}, \mathbf{w}^j)$, so this is a form of concept shift (see Section 19.2.3). However we assume the overall shape of each boundary is similar to a common shared boundary, denote $p(y|\mathbf{x}, \mathbf{w}^0)$. We only have a small number N_j of examples from each environment, $\mathcal{D}^j = \{(\mathbf{x}_n^j, y_n^j) : n = 1 : N_j\}$, but we can utilize their common structure to do better than fitting J separate models.

To illustrate this, we create some synthetic 2d data for the $J = 18$ tasks. We start with the two-moons dataset, illustrated in Figure 17.17a. Each task is obtained by rotating the 2d inputs by a different amount, to create 18 related classification problems (see Figure 17.17b). See Figure 17.17b 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 ($N_j = 50$ examples

2. This example is from https://tweicki.io/blog/2018/08/13/hierarchical_bayesian_neural_network/. For a real-world example of a similar approach applied to a gesture recognition task, see [Jos+17].

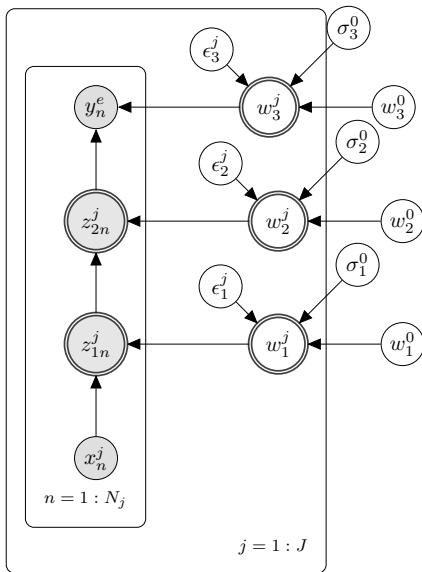


Figure 17.18: Illustration of a hierarchical Bayesian MLP with 2 hidden layers. There are J different models, each with N_j observed samples, and a common set of global shared parent parameters denoted with the 0 superscript. Nodes which are shaded are observed. Nodes with double ringed circles are deterministic functions of their parents.

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, so mixing the data together from different “concepts” confuses the model. Instead we adopt a hierarchical Bayesian approach.

Our modeling assumptions are shown in Figure 17.18. In particular, we assume the weight from unit i to unit k in layer l for environment j , denoted $w_{i,k,l}^j$, comes from a common prior value $w_{i,k,l}^0$, with a random offset. We use the non-centered parameterization from Section 12.6.5 to write

$$w_{i,k,l}^j = w_{i,k,l}^0 + \epsilon_{i,k,l}^j \times \sigma_l^0 \quad (17.57)$$

where $\epsilon_{i,k,l}^j \sim \mathcal{N}(0, 1)$. By allowing a different σ_l^0 per layer l , we let the model control the degree of shrinkage to the prior for each layer separately. (We could also make the σ_l^j parameters be environment specific, which would allow for different amounts of distribution shift from the common parent.) For the hyper-parameters, we put $\mathcal{N}(0, 1)$ priors on $w_{i,k,l}^0$, and $\mathcal{N}_+(1)$ priors on σ_l^0 .

We compute the posterior $p(\epsilon_{1:L}^{1:J}, \mathbf{w}_{1:L}^0, \boldsymbol{\sigma}_{1:L}^0 | \mathcal{D})$ using HMC (Section 12.5). We then evaluate this model using a fresh set of labeled samples from each environment. The average classification accuracy on the train and test sets for the non-hierarchical model (one MLP per environment, 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 17.19(top) shows the results for the nonhierarchical models; we see that the method

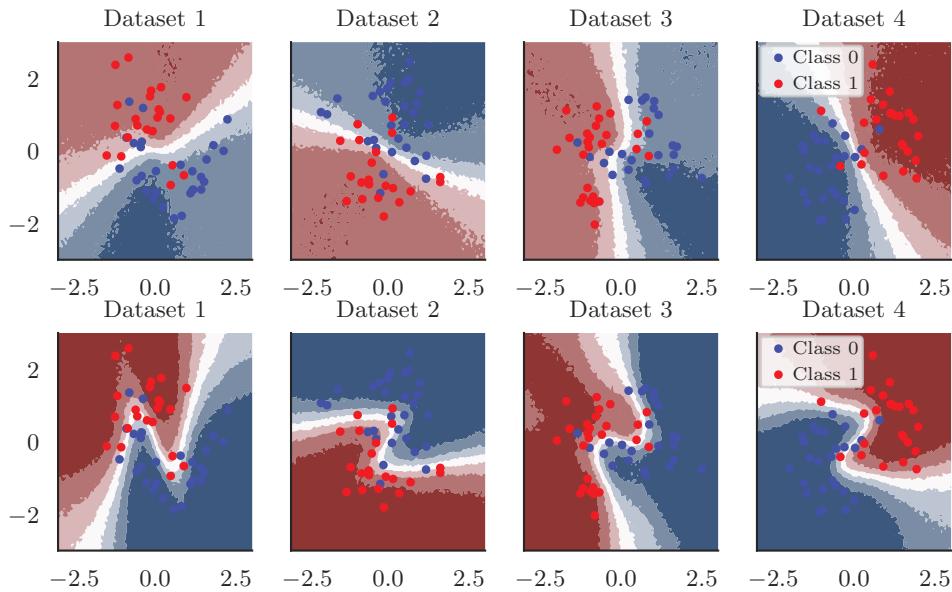


Figure 17.19: Top: Results of fitting separate MLPs on each dataset. Bottom: Results of fitting hierarchical MLP on all datasets jointly. Generated by [bnn_hierarchical.ipynb](#).

fails to learn the common underlying Z-shaped decision boundary. By contrast, Figure 17.19(bottom) shows that the hierarchical method has correctly recovered the common pattern, while still allowing group variation.

18 Gaussian processes

This chapter is coauthored 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. In particular, we will use a **Gaussian process** to represent the prior $p(f)$; we then use Bayes' rule to derive the posterior $p(f|\mathcal{D})$, which is another GP, as we explain below. More details on GPs can be found the excellent book [RW06], as well as the interative tutorial at <https://distill.pub/2019/visual-exploration-gaussian-processes>. See also Chapter 31 for other examples of Bayesian nonparametric models.

18.1.1 GPs: what and why?

To explain GPs 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 18.2.1.1 for details).

We denote the corresponding GP by

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), \mathcal{K}(\mathbf{x}, \mathbf{x}')) \tag{18.1}$$

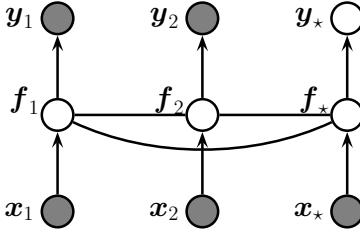


Figure 18.1: A Gaussian process for 2 training points, \mathbf{x}_1 and \mathbf{x}_2 , and 1 testing point, \mathbf{x}_* , represented as a graphical model representing $p(\mathbf{y}, \mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | m(\mathbf{X}), \mathcal{K}(\mathbf{X})) \prod_i p(y_i | f_i)$. The hidden nodes $f_i = f(\mathbf{x}_i)$ represent the value of the function at each of the datapoints. 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

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \quad (18.2)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))^\top] \quad (18.3)$$

This means that, for any finite set of points $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, we have

$$p(\mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | \boldsymbol{\mu}_X, \mathbf{K}_{X,X}) \quad (18.4)$$

where $\boldsymbol{\mu}_X = (m(\mathbf{x}_1), \dots, m(\mathbf{x}_N))$ 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 $y_i \sim p(y_i | f(\mathbf{x}_i))$, 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 28.3.7.

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 and 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 by considering infinite limits of neural networks that converge to Gaussian processes with particular covariance functions, which we describe in Section 18.7.

So Gaussian processes are nonparametric models which can scale and do representation learning. But why, in the age of deep learning, should we want to use a Gaussian process? There are several compelling reasons to prefer a GP, including:

- Gaussian processes typically provide well-calibrated predictive distributions, with a good characterization of epistemic (model) uncertainty — uncertainty arising from not knowing which of many solutions is correct. For example, as we move away from the data, there are a greater variety of consistent solutions, and so we expect greater uncertainty.
- Gaussian processes are often state-of-the-art for continuous regression problems, especially spatiotemporal problems, such as weather interpolation and forecasting. In regression, Gaussian process inference can also typically be performed in closed form.
- The marginal likelihood of a Gaussian process provides a powerful mechanism for flexible kernel learning. Kernel learning enables us to provide long-range extrapolations, but also tells us interpretable properties of the data that we didn't know before, towards scientific discovery.
- Gaussian processes are often used as a probabilistic surrogate for optimizing expensive objectives, in a procedure known as **Bayesian optimization** (Section 6.6).

18.2 Mercer kernels

The generalization properties of Gaussian processes boil down to how we encode prior knowledge about the similarity of two input vectors. If we know that \mathbf{x}_i is similar to \mathbf{x}_j , then we can encourage the model to make the predicted output at both locations (i.e., $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$) to be similar.

To define similarity, we introduce the notion of a **kernel function**. The word “kernel” has many different meanings in mathematics; here we consider a **Mercer kernel**, also called a **positive definite kernel**. This is any symmetric function $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$ such that

$$\sum_{i=1}^N \sum_{j=1}^N \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) c_i c_j \geq 0 \quad (18.5)$$

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 $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) > 0$, so that we can only achieve equality in the above equation if $c_i = 0$ for all i .

Another way to understand this condition is the following. Given a set of N datapoints, let us define the **Gram matrix** as the following $N \times N$ similarity matrix:

$$\mathbf{K} = \begin{pmatrix} \mathcal{K}(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ \mathcal{K}(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} \quad (18.6)$$

We say that \mathcal{K} is a Mercer kernel iff the Gram matrix is positive definite for any set of (distinct) inputs $\{\mathbf{x}_i\}_{i=1}^N$.

We discuss several popular Mercer kernels below. More details can be found at [Wil14] and <https://www.cs.toronto.edu/~duvenaud/cookbook/>. See also Section 18.6 where we discuss how to learn kernels from data.

18.2.1 Stationary kernels

For real-valued inputs, $\mathcal{X} = \mathbb{R}^D$, it is common to use **stationary kernels** (also called **shift-invariant kernels**), which are functions of 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 between the inputs. (See Section 18.2.2 for a discussion of non-stationary kernels.) Furthermore, in many cases, all that matters is the magnitude of the difference:

$$r = \|\mathbf{r}\|_2 = \|\mathbf{x} - \mathbf{x}'\| \quad (18.7)$$

We give some examples below. (See also Figure 18.3 and Figure 18.4 for some visualizations of these kernels.)

18.2.1.1 Squared exponential (RBF) kernel

The **squared exponential** (SE) kernel, also sometimes called the **exponentiated quadratic** kernel or the **radial basis function** (RBF) kernel, is defined as

$$\mathcal{K}(r; \ell) = \exp\left(-\frac{r^2}{2\ell^2}\right) \quad (18.8)$$

Here ℓ corresponds to the **length-scale** of the kernel, i.e., the distance over which we expect differences to matter.

From Equation (18.7) we can rewrite this kernel as

$$\mathcal{K}(\mathbf{x}, \mathbf{x}'; \ell) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \quad (18.9)$$

This is the RBF kernel we encountered earlier. It is also sometimes called the **Gaussian kernel**.

See Figure 18.3(f) and Figure 18.4(f) for a visualization in 1D.

18.2.1.2 ARD kernel

We can generalize the RBF kernel by replacing Euclidean distance with Mahalanobis distance, as follows:

$$\mathcal{K}(\mathbf{r}; \boldsymbol{\Sigma}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \mathbf{r}^\top \boldsymbol{\Sigma}^{-1} \mathbf{r}\right) \quad (18.10)$$

where $\mathbf{r} = \mathbf{x} - \mathbf{x}'$. If $\boldsymbol{\Sigma}$ is diagonal, this can be written as

$$\mathcal{K}(\mathbf{r}; \boldsymbol{\ell}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} r_d^2\right) = \prod_{d=1}^D \mathcal{K}(r_d; \ell_d, \sigma^{2/d}) \quad (18.11)$$

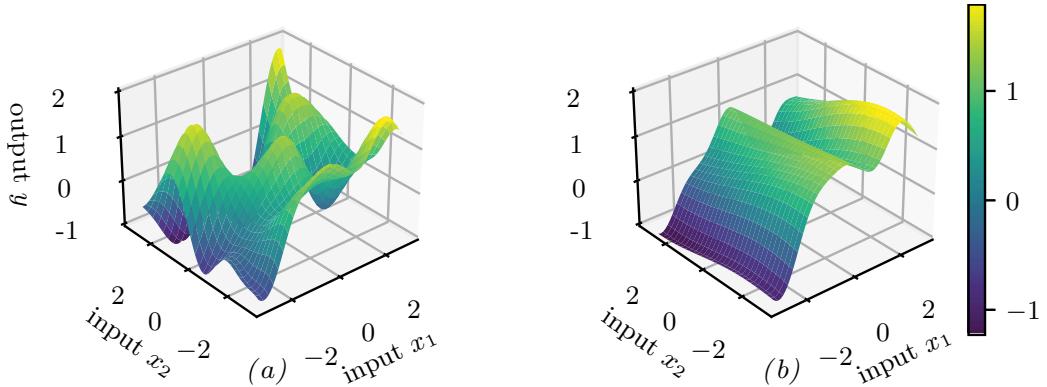


Figure 18.2: Function samples from a GP with an ARD kernel. (a) $\ell_1 = \ell_2 = 1$. Both dimensions contribute to the response. (b) $\ell_1 = 1, \ell_2 = 5$. The second dimension is essentially ignored. Adapted from Figure 5.1 of [RW06]. Generated by [gpr_demo_ard.ipynb](#).

where

$$\mathcal{K}(r; \ell, \tau^2) = \tau^2 \exp\left(-\frac{1}{2} \frac{1}{\ell^2} r^2\right) \quad (18.12)$$

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 15.2.8). 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.

18.2.1.3 Matérn 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 **Matérn kernel**, which gives rise to “rougher” functions, which can better model local “wiggles” without having to make the overall length scale very small.

The Matérn kernel has the following form:

$$\mathcal{K}(r; \nu, \ell) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu}r}{\ell} \right) \quad (18.13)$$

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.

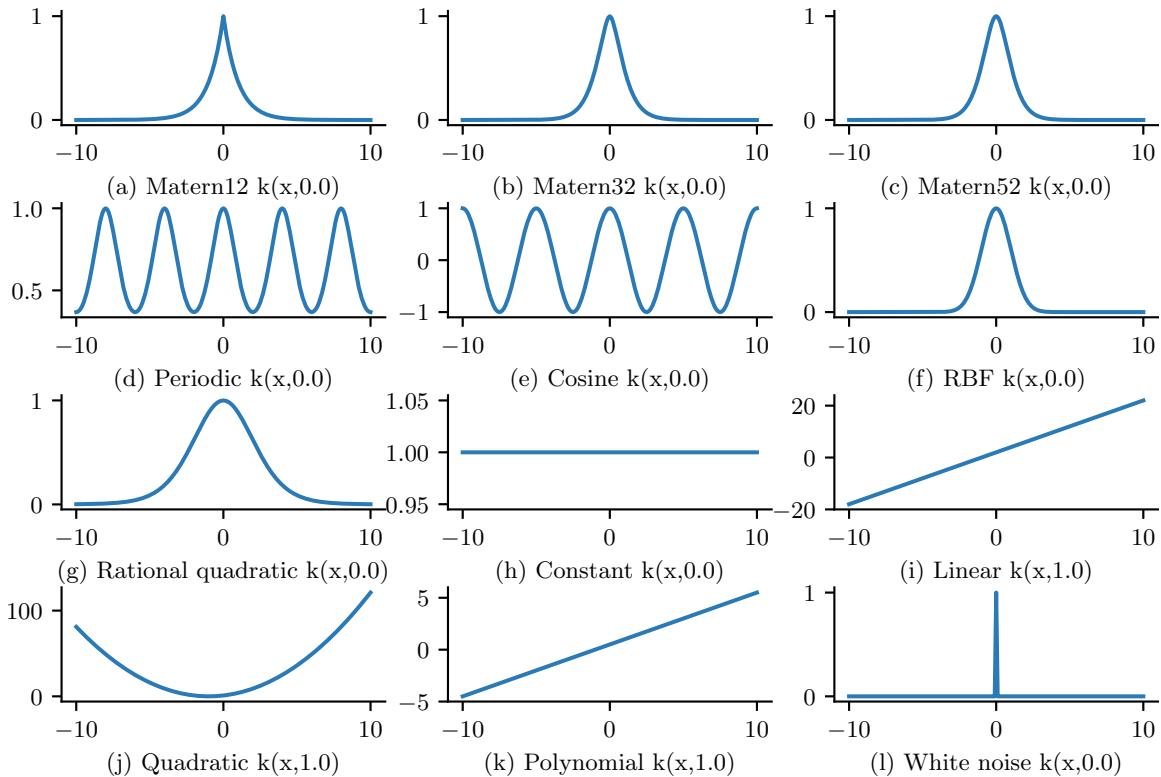


Figure 18.3: GP kernels evaluated at $k(x, 0)$ as a function of x . Generated by `gpKernelPlot.ipynb`.

For values $\nu \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}\}$, the function simplifies as follows:

$$\mathcal{K}(r; \frac{1}{2}, \ell) = \exp\left(-\frac{r}{\ell}\right) \quad (18.14)$$

$$\mathcal{K}(r; \frac{3}{2}, \ell) = \left(1 + \frac{\sqrt{3}r}{\ell}\right) \exp\left(-\frac{\sqrt{3}r}{\ell}\right) \quad (18.15)$$

$$\mathcal{K}(r; \frac{5}{2}, \ell) = \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}r}{\ell}\right) \quad (18.16)$$

See Figure 18.3(a-c) and Figure 18.4(a-c) for a visualization.

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”.

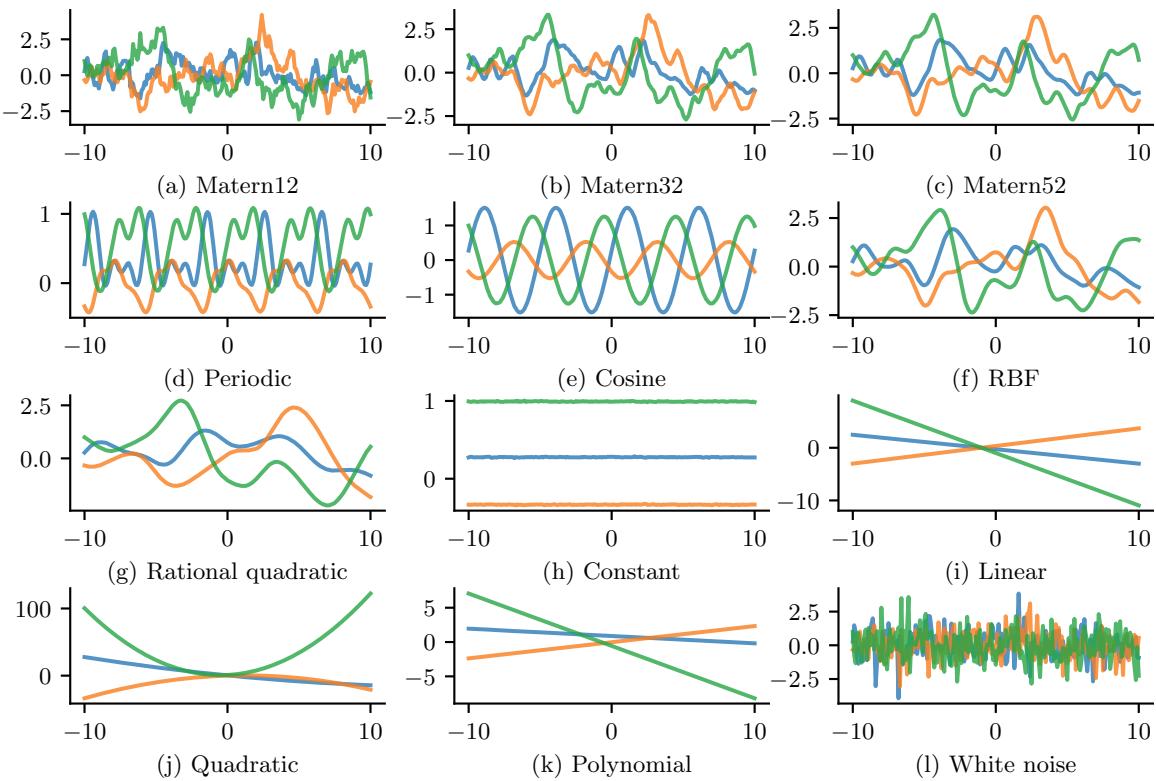


Figure 18.4: GP samples drawn using different kernels. Generated by [gpKernelPlot.ipynb](#).

18.2.1.4 Periodic kernels

One way to create a periodic 1d random function is to map x to the 2d space $\mathbf{u}(x) = (\cos(x), \sin(x))$, and then use an SE kernel in \mathbf{u} -space:

$$\mathcal{K}(x, x') = \exp\left(-\frac{2 \sin^2((x - x')/2)}{\ell^2}\right) \quad (18.17)$$

which follows since $(\cos(x) - \cos(x'))^2 + (\sin(x) - \sin(x'))^2 = 4 \sin^2((x - x')/2)$. We can generalize this by specifying the period p to get the **periodic kernel**, also called the **exp-sine-squared kernel**:

$$\mathcal{K}_{\text{per}}(r; \ell, p) = \exp\left(-\frac{2}{\ell^2} \sin^2\left(\pi \frac{r}{p}\right)\right) \quad (18.18)$$

where p is the period and ℓ is the length scale. See Figure 18.3(d-e) and Figure 18.4(d-e) for a visualization.

A related kernel is the **cosine kernel**:

$$\mathcal{K}(r; p) = \cos\left(2\pi \frac{r}{p}\right) \quad (18.19)$$

18.2.1.5 Rational quadratic kernel

We define the **rational quadratic** kernel to be

$$\mathcal{K}_{RQ}(r; \ell, \alpha) = \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha} \quad (18.20)$$

We recognize this is proportional to a Student t density. Hence it can be interpreted as a scale mixture of SE kernels of different characteristic lengths. In particular, let $\tau = 1/\ell^2$, and assume $\tau \sim \text{Ga}(\alpha, \ell^2)$. Then one can show that

$$\mathcal{K}_{RQ}(r) = \int p(\tau|\alpha, \ell^2) \mathcal{K}_{SE}(r|\tau) d\tau \quad (18.21)$$

As $\alpha \rightarrow \infty$, this reduces to a SE kernel.

See Figure 18.3(g) and Figure 18.4(g) for a visualization.

18.2.1.6 Kernels from spectral densities

Consider the case of a stationary kernel which satisfies $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}(\boldsymbol{\delta})$, 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. In this case, **Bochner's theorem** tells us that we can represent $\mathcal{K}(\boldsymbol{\delta})$ by its Fourier transform:

$$\mathcal{K}(\boldsymbol{\delta}) = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^\top \boldsymbol{\delta}} d\boldsymbol{\omega} \quad (18.22)$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j \sin(\theta)$, $\boldsymbol{\omega}$ is the frequency, and $p(\boldsymbol{\omega})$ is the **spectral density** (see [SS19, p93, p253] for details).

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 $p(\boldsymbol{\omega}) = \sqrt{2\pi\ell^2} \exp(-2\pi^2\omega^2\ell^2)$. 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 Matérn 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 Matérn 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.

18.2.2 Nonstationary kernels

A stationary kernel assumes the measure of similarity between two inputs is independent of their location, i.e., $\mathcal{K}(\mathbf{x}, \mathbf{x}')$ only depends on $\mathbf{r} = \mathbf{x} - \mathbf{x}'$. A **nonstationary kernel** relaxes this assumption. This is useful for a variety of problems, such as environmental modeling (see e.g., [GSR12; Pat+22]), where correlations between locations can change depending on latent factors in the environment.

18.2.2.1 Polynomial kernels

A simple form of non-stationary kernel is the **polynomial kernel** (also called **dot product kernel**) of order M , defined by

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}')^M \quad (18.23)$$

This contains all monomials of order M . For example, if $M = 2$, we get the **quadratic kernel**; in 2d, this becomes

$$(\mathbf{x}^\top \mathbf{x}')^2 = (x_1 x'_1 + x_2 x'_2)^2 = (x_1 x'_1)^2 + (x_2 x_2)^2 + 2(x_1 x'_1)(x_2 x'_2) \quad (18.24)$$

We can generalize this to contain all terms up to degree M by using the **inhomogeneous polynomial kernel**

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + c)^M \quad (18.25)$$

For example, if $M = 2$ and the inputs are 2d, we have

$$\begin{aligned} (\mathbf{x}^\top \mathbf{x}' + 1)^2 &= (x_1 x'_1)^2 + (x_1 x'_1)(x_2 x'_2) + (x_1 x'_1) \\ &\quad + (x_2 x_2)(x_1 x'_1) + (x_2 x'_2)^2 + (x_2 x'_2) \\ &\quad + (x_1 x'_1) + (x_2 x'_2) + 1 \end{aligned} \quad (18.26)$$

18.2.2.2 Gibbs kernel

Consider an RBF kernel where the length scale hyper-parameter, and the signal variance hyper-parameter, are both input dependent; this is called the **Gibbs kernel** [Gib97], and is defined by

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sigma(\mathbf{x})\sigma(\mathbf{x}') \sqrt{\frac{2\ell(\mathbf{x})\ell(\mathbf{x}')}{\ell(\mathbf{x})^2 + \ell(\mathbf{x}')^2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{\ell(\mathbf{x})^2 + \ell(\mathbf{x}')^2}\right) \quad (18.27)$$

If $\ell(\mathbf{x})$ and $\sigma(\mathbf{x})$ are constants, this reduces to the standard RBF kernel. We can model the functional dependency of these kernel parameters on the input by using another GP (see e.g., [Hei+16]).

18.2.2.3 Other non-stationary kernels

Other ways to induce non-stationarity include using a neural network kernel (Section 18.7.1), non-stationary spectral kernels [RHK17], or a deep GP (Section 18.7.3).

18.2.3 Kernels for nonvectorial (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].

18.2.4 Making new kernels from old

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 methods:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = c\mathcal{K}_1(\mathbf{x}, \mathbf{x}'), \text{ for any constant } c > 0 \quad (18.28)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})\mathcal{K}_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}'), \text{ for any function } f \quad (18.29)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = q(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \text{ for any function polynomial } q \text{ with nonneg. coef.} \quad (18.30)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \quad (18.31)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{A} \mathbf{x}', \text{ for any psd matrix } \mathbf{A} \quad (18.32)$$

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 Mercer kernel, since the corresponding Gram matrix is just the (scaled) covariance matrix of the data. From the above rules, we can see that the polynomial kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}')^M$ from Section 18.2.2.1 is a valid Mercer kernel.

We can also use the above rules to establish that the Gaussian kernel is a valid kernel. To see this, note that

$$\|\mathbf{x} - \mathbf{x}'\|^2 = \mathbf{x}^\top \mathbf{x} + (\mathbf{x}')^\top \mathbf{x}' - 2\mathbf{x}^\top \mathbf{x}' \quad (18.33)$$

and hence

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2) = \exp(-\mathbf{x}^\top \mathbf{x} / 2\sigma^2) \exp(\mathbf{x}^\top \mathbf{x}' / \sigma^2) \exp(-(\mathbf{x}')^\top \mathbf{x}' / 2\sigma^2) \quad (18.34)$$

is a valid kernel.

We can also combine kernels using addition or multiplication:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') + \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.35)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') \times \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.36)$$

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 time series data, see Section 18.8.1.

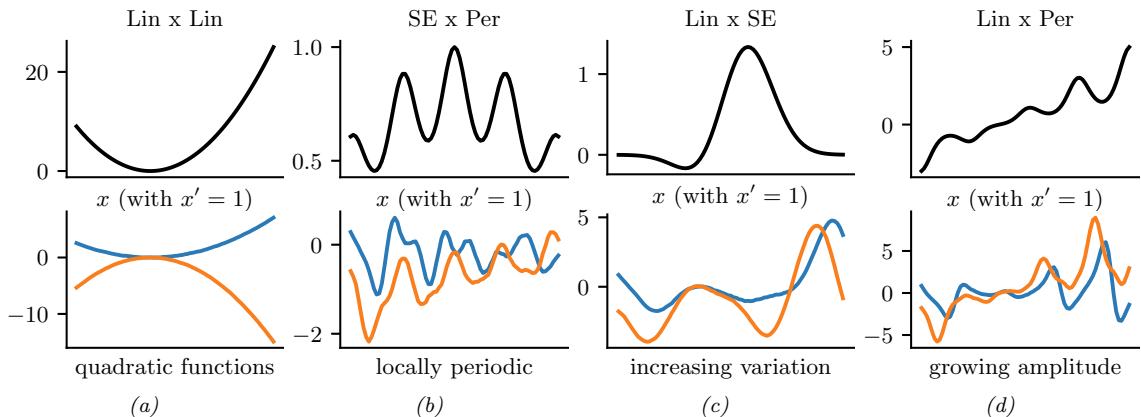


Figure 18.5: Examples of 1d structures obtained by multiplying elementary kernels. Top row shows $K(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, K)$. Adapted from Figure 2.2 of [Duv14]. Generated by [combining_kernels_by_multiplication.ipynb](#).

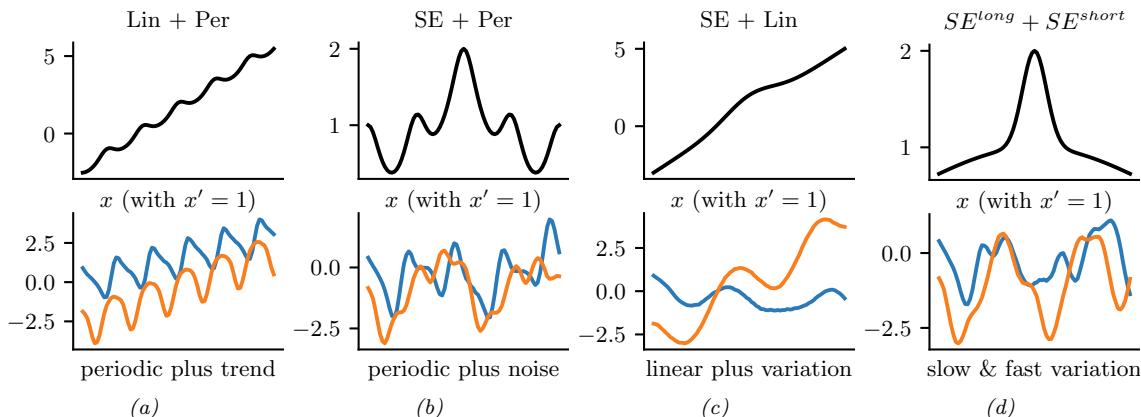


Figure 18.6: Examples of 1d structures obtained by summing elementary kernels. Top row shows $K(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, K)$. Adapted from Figure 2.2 of [Duv14]. Generated by [combining_kernels_by_summation.ipynb](#).

18.2.5 Mercer's theorem

Recall that any positive definite matrix \mathbf{K} can be represented using an eigendecomposition of the form $\mathbf{K} = \mathbf{U}^T \Lambda \mathbf{U}$, where Λ 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} :

$$k_{ij} = (\Lambda^{\frac{1}{2}} \mathbf{U}_{:i})^T (\Lambda^{\frac{1}{2}} \mathbf{U}_{:j}) \quad (18.37)$$

where $\mathbf{U}_{:i}$ is the i 'th column of \mathbf{U} . If we define $\phi(\mathbf{x}_i) = \mathbf{U}_{:i}$, then we can write

$$k_{ij} = \sum_{m=1}^M \lambda_m \phi_m(\mathbf{x}_i) \phi_m(\mathbf{x}_j) \quad (18.38)$$

where M is the rank of the kernel matrix. Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors of the kernel matrix.

This idea can be generalized to apply to kernel functions, not just kernel matrices, as we now show. First, we define an **eigenfunction** $\phi()$ of a kernel \mathcal{K} with eigenvalue λ wrt measure μ as a function that satisfies

$$\int \mathcal{K}(\mathbf{x}, \mathbf{x}') \phi(\mathbf{x}) d\mu(\mathbf{x}) = \lambda \phi(\mathbf{x}') \quad (18.39)$$

We usually sort the eigenfunctions in order of decreasing eigenvalue, $\lambda_1 \geq \lambda_2 \geq \dots$. The eigenfunctions are orthogonal wrt μ :

$$\int \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mu(\mathbf{x}) = \delta_{ij} \quad (18.40)$$

where δ_{ij} is the Kronecker delta. With this definition in hand, we can state **Mercer's theorem**. Informally, it says that any positive definite kernel function can be represented as the following infinite sum:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{m=1}^{\infty} \lambda_m \phi_m(\mathbf{x}) \phi_m(\mathbf{x}') \quad (18.41)$$

where ϕ_m are eigenfunctions of the kernel, and λ_m are the corresponding eigenvalues. This is the functional analog of Equation (18.38).

A **degenerate kernel** has only a finite number of non-zero eigenvalues. In this case, we can rewrite the kernel function as an inner product between two finite-length vectors. For example, consider the quadratic kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle^2$ from Equation (18.24). If we define $\phi(x_1, x_2) = [x_1^2, \sqrt{2}x_1x_2, x_2^2] \in \mathbb{R}^3$, then we can write this as $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \phi(\mathbf{x}')$. Thus we see that this kernel is degenerate.

Now consider the RBF kernel. In this case, the corresponding feature representation is infinite dimensional (see Section 18.2.6 for details). However, by working with kernel functions, we can avoid having to deal with infinite dimensional vectors.

From the above, we see that we can replace inner product operations in an explicit (possibly infinite dimensional) feature space with a call to a kernel function, i.e., we replace $\phi(\mathbf{x})^\top \phi(\mathbf{x})$ with $\mathcal{K}(\mathbf{x}, \mathbf{x}')$. This is called the **kernel trick**.

18.2.6 Approximating kernels with random features

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} , as we will see in Section 18.3. 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. (18.22). 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 the (real-valued) feature vector is given by

$$\phi(\mathbf{x}) = \sqrt{\frac{1}{D}} [\sin(\mathbf{z}_1^\top \mathbf{x}), \dots, \sin(\mathbf{z}_D^\top \mathbf{x}), \cos(\mathbf{z}_1^\top \mathbf{x}), \dots, \cos(\mathbf{z}_D^\top \mathbf{x})] \quad (18.42)$$

$$= \sqrt{\frac{1}{D}} [\sin(\mathbf{Z}^\top \mathbf{x}), \cos(\mathbf{Z}^\top \mathbf{x})] \quad (18.43)$$

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 iid from $\mathcal{N}(0, 1)$. The representation in Equation (18.43) are called **random Fourier features (RFF)** [SS15; RR08] or “weighted sums of random kitchen sinks” [RR09]. (One can obtain an even better approximation by ensuring that the rows of \mathbf{Z} are random but orthogonal; this is called **orthogonal random features** [Yu+16].)

One can create similar random feature representations for other kinds of kernels. We can then 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 Gaussian matrix, and the form of φ depends on the chosen kernel. This is equivalent to a one layer MLP with random input-to-hidden weights; since we only optimize the hidden-to-output weights $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$, the model is equivalent to a linear model with fixed random features. If we use enough random features, we can approximate the performance of a kernelized prediction model, but the computational cost is now $O(N)$ rather than $O(N^2)$.

Unfortunately, random features can result in worse performance than using a non-degenerate kernel, since they don't have enough expressive power. We discuss other ways to scale GPs to large datasets in Section 18.5.

18.3 GPs with Gaussian likelihoods

In this section, we discuss GPs for regression, using a Gaussian likelihood. In this case, all the computations can be performed in closed form, using standard linear algebra methods. We extend this framework to non-Gaussian likelihoods later in the chapter.

18.3.1 Predictions using noise-free observations

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 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 has already seen, we want the GP to return the answer $f(\mathbf{x})$ with no uncertainty. In other words, it should act as an **interpolator** of the training data. Here we assume the observed function values are noiseless. We will consider the case of noisy observations shortly.

Now we consider the case of predicting the outputs for new inputs that may not be in \mathcal{D} . Specifically, given a test set \mathbf{X}_* of size $N_* \times D$, we want to predict the function outputs $\mathbf{f}_* = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_{N_*})]$.

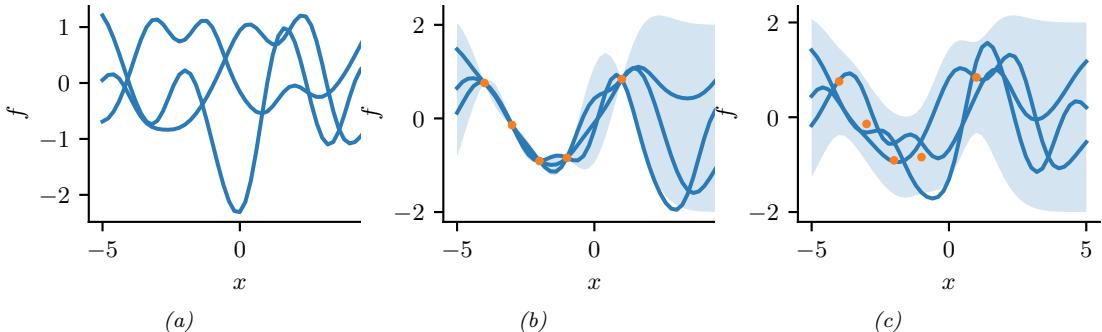


Figure 18.7: Left: some functions sampled from a GP prior with RBF kernel. Middle: some samples from a GP posterior, after conditioning on 5 noise-free observations. Right: some samples from a GP posterior, after conditioning on 5 noisy observations. The shaded area represents $\mathbb{E}[f(\mathbf{x})] \pm 2\sqrt{\text{Var}[f(\mathbf{x})]}$. Adapted from Figure 2.2 of [RW06]. Generated by [gpr_demo_noise.ipynb](#).

By definition of the GP, the joint distribution $p(\mathbf{f}_X, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$ has the following form

$$\begin{pmatrix} \mathbf{f}_X \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^T & \mathbf{K}_{*,*} \end{pmatrix} \right) \quad (18.44)$$

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 $N \times N$, $\mathbf{K}_{X,*} = \mathcal{K}(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$, and $\mathbf{K}_{*,*} = \mathcal{K}(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. See Figure 18.7 for a static illustration, and <http://www.infinitecuriosity.org/vizgp/> for an interactive visualization.

By the standard rules for conditioning Gaussians (Section 2.3.1.4), the posterior has the following form

$$p(\mathbf{f}_* | \mathbf{X}_*, \mathcal{D}) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.45)$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^T \mathbf{K}_{X,X}^{-1} (\mathbf{f}_X - \boldsymbol{\mu}_X) \quad (18.46)$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^T \mathbf{K}_{X,X}^{-1} \mathbf{K}_{X,*} \quad (18.47)$$

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 18.2.1.1) 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

$$\text{Cov}[y_i, y_j] = \text{Cov}[f_i, f_j] + \text{Cov}[\epsilon_i, \epsilon_j] = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) + \sigma_y^2 \delta_{ij} \quad (18.48)$$

where $\delta_{ij} = \mathbb{I}(i = j)$. In other words

$$\text{Cov}[\mathbf{y}|\mathbf{X}] = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I}_N \quad (18.49)$$

The joint density of the observed data and the latent, noise-free function on the test points is given by

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\top & \mathbf{K}_{*,*} \end{pmatrix}\right) \quad (18.50)$$

Hence the posterior predictive density at a set of test points \mathbf{X}_* is

$$p(\mathbf{f}_* | \mathcal{D}, \mathbf{X}_*) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.51)$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) \quad (18.52)$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{K}_{X,*} \quad (18.53)$$

In the case of a single test input, this simplifies as follows

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | m_* + \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X), k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_*) \quad (18.54)$$

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 write the posterior mean as follows:

$$\boldsymbol{\mu}_{*|X} = \mathbf{k}_*^\top \underbrace{\mathbf{K}_\sigma^{-1}}_{\boldsymbol{\alpha}} \mathbf{y} = \sum_{n=1}^N \mathcal{K}(\mathbf{x}_*, \mathbf{x}_n) \alpha_n \quad (18.55)$$

where

$$\mathbf{K}_\sigma = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} \quad (18.56)$$

$$\boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.57)$$

Fitting this model amounts to computing $\boldsymbol{\alpha}$ in Equation (18.57). This is usually done by computing the Cholesky decomposition of \mathbf{K}_σ , as described in Section 18.3.6. Once we have computed $\boldsymbol{\alpha}$, we can compute predictions for each test point in $O(N)$ time for the mean, and $O(N^2)$ time for the variance.

18.3.3 Weight space vs function space

In this section, we show how Bayesian linear regression is a special case of a GP.

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 use a Gaussian prior $p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \boldsymbol{\Sigma}_w)$, then the posterior is as follows (see Section 15.2.2 for the derivation):

$$p(\mathbf{w} | \mathcal{D}) = \mathcal{N}(\mathbf{w} | \frac{1}{\sigma_y^2} \mathbf{A}^{-1} \Phi^T \mathbf{y}, \mathbf{A}^{-1}) \quad (18.58)$$

where Φ is the $N \times D$ design matrix, and

$$\mathbf{A} = \sigma_y^{-2} \Phi^\top \Phi + \Sigma_w^{-1} \quad (18.59)$$

The posterior predictive distribution for $f_* = f(\mathbf{x}_*)$ is therefore

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \frac{1}{\sigma_y^2} \phi_*^\top \mathbf{A}^{-1} \Phi^\top \mathbf{y}, \phi_*^\top \mathbf{A}^{-1} \phi_*) \quad (18.60)$$

where $\phi_* = \phi(\mathbf{x}_*)$. This views the problem of inference and prediction in **weight space**.

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 (18.60) as follows

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.61)$$

$$\boldsymbol{\mu}_{*|X} = \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} = \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.62)$$

$$\boldsymbol{\Sigma}_{*|X} = \phi_*^\top \Sigma_w \phi_* - \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \Phi \Sigma_w \phi_* = k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_* \quad (18.63)$$

which matches the results in Equation (18.54), 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 18.2.5). That is, a GP works in **function space**.

18.3.4 Semiparametric 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:

$$g(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\beta}^\top \phi(\mathbf{x}) \quad (18.64)$$

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]:

$$g(\mathbf{x}) \sim \text{GP}(\phi(\mathbf{x})^\top \mathbf{b}, \mathcal{K}(\mathbf{x}, \mathbf{x}') + \phi(\mathbf{x})^\top \mathbf{B} \phi(\mathbf{x}')) \quad (18.65)$$

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]:

$$\mathbb{E}[g(\mathbf{X}_*) | \mathcal{D}] = \mathbf{H}_*^\top \bar{\boldsymbol{\beta}} + \mathbf{K}_{X,*}^\top \mathbf{K}_\sigma^{-1} (\mathbf{y} - \mathbf{H}_X^\top \bar{\boldsymbol{\beta}}) = \mathbb{E}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top \bar{\boldsymbol{\beta}} \quad (18.66)$$

$$\text{Cov}[g(\mathbf{X}_*) | \mathcal{D}] = \text{Cov}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} \mathbf{R} \quad (18.67)$$

$$\bar{\boldsymbol{\beta}} = (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} + \mathbf{B}^{-1} \mathbf{b}) \quad (18.68)$$

$$\mathbf{R} = \mathbf{H}_* - \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{K}_{X,*} \quad (18.69)$$

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 $\bar{\beta}$; and the covariance is the usual covariance from the GP, plus an additional positive term due to the uncertainty in β .

In the limit of an uninformative prior for the regression parameters, as $\mathbf{B} \rightarrow \infty \mathbf{I}$, this simplifies to

$$\mathbb{E}[g(\mathbf{X}_*)|\mathcal{D}] = \mathbb{E}[f(\mathbf{X}_*)|\mathcal{D}] + \mathbf{R}^T (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^T)^{-1} \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.70)$$

$$\text{Cov}[g(\mathbf{X}_*)|\mathcal{D}] = \text{Cov}[f(\mathbf{X}_*)|\mathcal{D}] + \mathbf{R}^T (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^T)^{-1} \mathbf{R} \quad (18.71)$$

18.3.5 Marginal likelihood

Most kernels have some free parameters. For example, the RBF-ARD kernel (Section 18.2.1.2) has the form

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp \left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} (x_d - x'_d)^2 \right) = \prod_{d=1}^D \mathcal{K}_{\ell_d}(x_d, x'_d) \quad (18.72)$$

where each ℓ_d is a length scale for feature dimension d . Let these (and the observation noise variance σ_y^2 , if present) be denoted by θ . We can compute the likelihood of these parameters as follows:

$$p(\mathbf{y}|\mathbf{X}, \theta) = p(\mathcal{D}|\theta) = \int p(\mathbf{y}|\mathbf{f}_X, \theta) p(\mathbf{f}_X|\mathbf{X}, \theta) d\mathbf{f}_X \quad (18.73)$$

Since we are integrating out the function f , we often call θ hyperparameters, and the quantity $p(\mathcal{D}|\theta)$ the marginal likelihood.

Since f is a GP, we can compute the above integral using the marginal likelihood for the corresponding Gaussian. This gives

$$\log p(\mathcal{D}|\theta) = -\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu}_X)^T \mathbf{K}_\sigma^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) - \frac{1}{2} \log |\mathbf{K}_\sigma| - \frac{N}{2} \log(2\pi) \quad (18.74)$$

The first term is the square of the Mahalanobis distance between the observations and the predicted values: better fits will have smaller distance. The second term is the log determinant of the covariance matrix, which measures model complexity: smoother functions will have smaller determinants, so $-\log |\mathbf{K}_\sigma|$ will be larger (less negative) for simpler functions. The marginal likelihood measures the tradeoff between fit and complexity.

In Section 18.6.1, we discuss how to learn the kernel parameters from data by maximizing the marginal likelihood wrt θ .

18.3.6 Computational and numerical issues

In this section, we discuss computational and numerical issues which arise when implementing the above equations. For notational simplicity, we assume the prior mean is zero, $m(\mathbf{x}) = 0$.

The posterior predictive mean is given by $\mu_* = \mathbf{k}_*^T \mathbf{K}_\sigma^{-1} \mathbf{y}$. For reasons of numerical stability, it is unwise to directly invert \mathbf{K}_σ . A more robust alternative is to compute a Cholesky decomposition, $\mathbf{K}_\sigma = \mathbf{L}\mathbf{L}^T$, which takes $O(N^3)$ time. Given this, we can compute

$$\mu_* = \mathbf{k}_*^T \mathbf{K}_\sigma^{-1} \mathbf{y} = \mathbf{k}_*^T \mathbf{L}^{-T} (\mathbf{L}^{-1} \mathbf{y}) = \mathbf{k}_*^T \boldsymbol{\alpha} \quad (18.75)$$

Here $\boldsymbol{\alpha} = \mathbf{L}^T \setminus (\mathbf{L} \setminus \mathbf{y})$, where we have used the backslash operator to represent backsubstitution.

We can compute the variance in $O(N^2)$ time for each test case using

$$\sigma_*^2 = k_{**} - \mathbf{k}_*^T \mathbf{L}^{-T} \mathbf{L}^{-1} \mathbf{k}_* = k_{**} - \mathbf{v}^T \mathbf{v} \quad (18.76)$$

where $\mathbf{v} = \mathbf{L} \setminus \mathbf{k}_*$.

Finally, the log marginal likelihood (needed for kernel learning, Section 18.6) can be computed using

$$\log p(\mathbf{y} | \mathbf{X}) = -\frac{1}{2} \mathbf{y}^T \boldsymbol{\alpha} - \sum_{n=1}^N \log L_{nn} - \frac{N}{2} \log(2\pi) \quad (18.77)$$

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:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + \lambda \|\mathbf{w}\|_2^2 \quad (18.78)$$

where $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \mathbf{x}$. The solution for this is

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^T + \lambda \mathbf{I} \right)^{-1} \left(\sum_{n=1}^N \mathbf{x}_n y_n \right) \quad (18.79)$$

In this section, we consider a function space version of this:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \lambda \|f\|^2 \quad (18.80)$$

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:

$$\langle \alpha f_1 + \beta f_2, g \rangle = \alpha \langle f_1, g \rangle + \beta \langle f_2, g \rangle \quad (18.81)$$

$$\langle f, g \rangle = \langle g, f \rangle \quad (18.82)$$

$$\langle f, f \rangle \geq 0 \quad (18.83)$$

$$\langle f, f \rangle = 0 \text{ iff } f(x) = 0 \text{ for all } x \in \mathcal{X} \quad (18.84)$$

We define the norm of a function using

$$\|f\| \triangleq \sqrt{\langle f, f \rangle} \quad (18.85)$$

A function space \mathcal{H} with an inner product operator is called a **Hilbert space**. (We also require that the function space be complete, which means that every Cauchy sequence of functions $f_i \in \mathcal{H}$ has a limit that is also in \mathcal{H} .)

The most common Hilbert space is the space known as L^2 . To define this, we need to specify a **measure** μ on the input space \mathcal{X} ; this is a function that assigns any (suitable) subset A of \mathcal{X} to a positive number, such as its volume. This can be defined in terms of the density function $w : \mathcal{X} \rightarrow \mathbb{R}$, as follows:

$$\mu(A) = \int_A w(x) dx \quad (18.86)$$

Thus we have $\mu(dx) = w(x)dx$. We can now define $L^2(\mathcal{X}, \mu)$ to be the space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$ that satisfy

$$\int_{\mathcal{X}} f(x)^2 w(x) dx < \infty \quad (18.87)$$

This is known as the set of **square-integrable functions**. This space has an inner product defined by

$$\langle f, g \rangle = \int_{\mathcal{X}} f(x)g(x)w(x) dx \quad (18.88)$$

We define a **Reproducing Kernel Hilbert Space** or **RKHS** as follows. Let \mathcal{H} be a Hilbert 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 exists a (symmetric) **kernel function** $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ with the following properties:

- For every $\mathbf{x} \in \mathcal{X}$, $\mathcal{K}(\mathbf{x}, \cdot) \in \mathcal{H}$.
- \mathcal{K} satisfies the **reproducing property**:

$$\langle f(\cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = f(\mathbf{x}') \quad (18.89)$$

The reason for the term “reproducing property” is as follows. Let $f(\cdot) = \mathcal{K}(\mathbf{x}, \cdot)$. Then we have that

$$\langle \mathcal{K}(\mathbf{x}, \cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = \mathcal{K}(\mathbf{x}, \mathbf{x}') \quad (18.90)$$

18.3.7.2 Complexity of a function in an RKHS

The main utility of RKHS from the point of view of machine learning is that it allows us to define a notion of a function's "smoothness" or "complexity" in terms of its norm, as we now discuss.

Suppose we have a positive definite kernel function \mathcal{K} . From Mercer's theorem we have $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{\infty} \lambda_i \phi_i(\mathbf{x}) \phi_i(\mathbf{x}')$. Now consider a Hilbert space \mathcal{H} defined by functions of the form $f(\mathbf{x}) = \sum_{i=1}^{\infty} f_i \phi_i(\mathbf{x})$, with $\sum_{i=1}^{\infty} f_i^2 / \lambda_i < \infty$. The inner product of two functions in this space is

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i g_i}{\lambda_i} \quad (18.91)$$

Hence the (squared) norm is given by

$$\|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i^2}{\lambda_i} \quad (18.92)$$

This is analogous to the quadratic form $\mathbf{f}^T \mathbf{K}^{-1} \mathbf{f}$ which occurs in some GP objectives (see Equation (18.101)). Thus the smoothness of the function is controlled by the properties of the corresponding kernel.

18.3.7.3 Representer theorem

In this section, we consider the problem of (regularized) empirical risk minimization in function space. In particular, consider the following problem:

$$f^* = \underset{f \in \mathcal{H}_{\mathcal{K}}}{\operatorname{argmin}} \sum_{n=1}^N \ell(y_n, f(\mathbf{x}_n)) + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2 \quad (18.93)$$

where $\mathcal{H}_{\mathcal{K}}$ is an RKHS with kernel \mathcal{K} and $\ell(y, \hat{y}) \in \mathbb{R}$ is a loss function. Then one can show [KW70; SHS01] the following result:

$$f^*(x) = \sum_{n=1}^N \alpha_n \mathcal{K}(x, \mathbf{x}_n) \quad (18.94)$$

where $\alpha_n \in \mathbb{R}$ are some coefficients that depend on the training data. This is called the **representer theorem**.

Now consider the special case where the loss function is squared loss, and $\lambda = \sigma_y^2$. We want to minimize

$$\mathcal{L}(f) = \frac{1}{2\sigma_y^2} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \frac{1}{2} \|f\|_{\mathcal{H}}^2 \quad (18.95)$$

Substituting in Equation (18.94), and using the fact that $\langle \mathcal{K}(\cdot, \mathbf{x}_i), \mathcal{K}(\cdot, \mathbf{x}_j) \rangle = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$, we obtain

$$\mathcal{L}(f) = \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \|\mathbf{y} - \mathbf{K} \boldsymbol{\alpha}\|^2 \quad (18.96)$$

$$= \frac{1}{2} \boldsymbol{\alpha}^T (\mathbf{K} + \frac{1}{\sigma_y^2} \mathbf{K}^2) \boldsymbol{\alpha} - \frac{1}{\sigma_y^2} \mathbf{y}^T \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \mathbf{y}^T \mathbf{y} \quad (18.97)$$

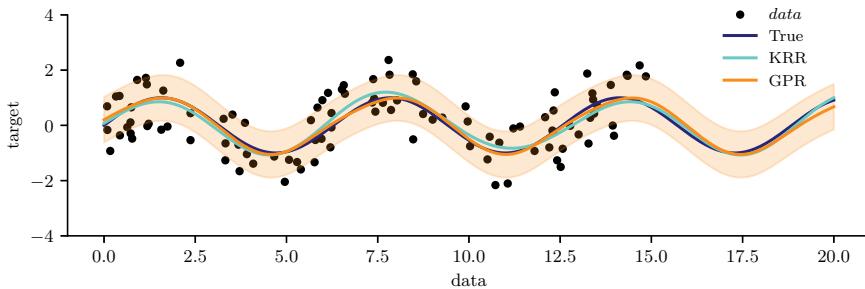


Figure 18.8: Kernel ridge regression (KRR) compared to Gaussian process regression (GPR) using the same kernel. Generated by [krr_vs_gpr.ipynb](#).

Minimizing this wrt α gives $\hat{\alpha} = (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y}$, which is the same as Equation (18.57). Furthermore, the prediction for a test point is

$$\hat{f}(\mathbf{x}_*) = \mathbf{k}_*^\top \hat{\alpha} = \mathbf{k}_*^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.98)$$

This is known as **kernel ridge regression** [Vov13]. We see that the result matches the posterior predictive mean of a GP in Equation (18.55).

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 (18.18). To capture the fact that the observations are noisy, we add to this a **white noise kernel**

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sigma_y^2 \delta(\mathbf{x} - \mathbf{x}') \quad (18.99)$$

as in Equation (18.48). 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 hyperparameters (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. We can create variety of different “classical” models by changing the form of the likelihood, as we show in Table 18.1. In the sections below, we briefly discuss some approximate inference methods. (For more details, see e.g., [WSS21].)

Model	Likelihood	Section
Regression	$\mathcal{N}(f_i, \sigma_y^2)$	Section 18.3.2
Robust regression	$T_\nu(f_i, \sigma_y^2)$	Section 18.4.4
Binary classification	$\text{Ber}(\sigma(f_i))$	Section 18.4.1
Multiclass classification	$\text{Cat}(\text{softmax}(\mathbf{f}_i))$	Section 18.4.2
Poisson regression	$\text{Poi}(\exp(f_i))$	Section 18.4.3

Table 18.1: Summary of GP models with a variety of likelihoods.

$$\begin{array}{c|c|c} \log p(y_i|f_i) & \frac{\partial}{\partial f_i} \log p(y_i|f_i) & \frac{\partial^2}{\partial f_i^2} \log p(y_i|f_i) \\ \hline \log \sigma(y_i f_i) & t_i - \pi_i & -\pi_i(1 - \pi_i) \\ \log \Phi(y_i f_i) & \frac{y_i \phi(f_i)}{\Phi(y_i f_i)} & -\frac{\phi_i^2}{\Phi(y_i f_i)^2} - \frac{y_i f_i \phi(f_i)}{\Phi(y_i f_i)} \end{array}$$

Table 18.2: Likelihood, gradient, and Hessian for binary logistic/probit GP regression. We assume $y_i \in \{-1, +1\}$ 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 regression. Also, ϕ and Φ are the pdf and cdf of $\mathcal{N}(0, 1)$. From [RW06, p43].

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 joint has the form

$$\mathcal{L}(\mathbf{f}_X) = \log p(\mathbf{y}|\mathbf{f}_X) + \log p(\mathbf{f}_X|\mathbf{X}) \quad (18.100)$$

$$= \log p(\mathbf{y}|\mathbf{f}_X) - \frac{1}{2} \mathbf{f}_X^\top \mathbf{K}_{X,X}^{-1} \mathbf{f}_X - \frac{1}{2} \log |\mathbf{K}_{X,X}| - \frac{N}{2} \log 2\pi \quad (18.101)$$

The simplest approach to approximate inference is to use a Laplace approximation (Section 7.4.3). The gradient and Hessian of the log joint are given by

$$\nabla \mathcal{L} = \nabla \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} \mathbf{f}_X \quad (18.102)$$

$$\nabla^2 \mathcal{L} = \nabla^2 \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} = -\boldsymbol{\Lambda} - \mathbf{K}_{X,X}^{-1} \quad (18.103)$$

where $\boldsymbol{\Lambda} \triangleq -\nabla^2 \log p(\mathbf{y}|\mathbf{f}_X)$ is a diagonal matrix, since the likelihood factorizes across examples. Expressions for the gradient and Hessian of the log likelihood for the logit and probit case are shown in Table 18.2. At convergence, the Laplace approximation of the posterior takes the following form:

$$p(\mathbf{f}_X|\mathcal{D}) \approx q(\mathbf{f}_X) = \mathcal{N}(\hat{\mathbf{f}}, (\mathbf{K}_{X,X}^{-1} + \boldsymbol{\Lambda})^{-1}) \quad (18.104)$$

where $\hat{\mathbf{f}}$ is the MAP estimate. See [RW06, Sec 3.4] for further details.

For improved accuracy, we can use variational inference, in which we assume $q(\mathbf{f}_X) = \mathcal{N}(\mathbf{f}_X|\mathbf{m}, \mathbf{S})$; we then optimize \mathbf{m} and \mathbf{S} using (stochastic) gradient descent, rather than assuming \mathbf{S} is the Hessian at the mode. See Section 18.5.4 for the details.

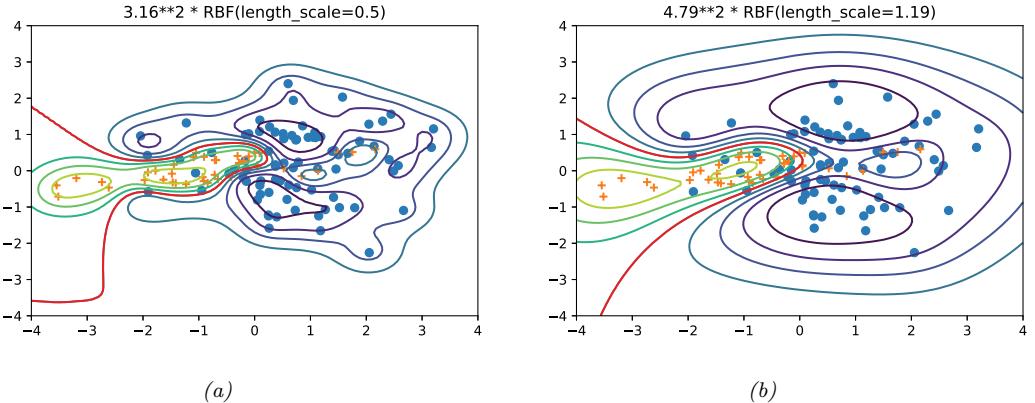


Figure 18.9: Contours of the posterior predictive probability for a binary classifier generated by a GP with an SE kernel. (a) Manual kernel parameters: short length scale, $\ell = 0.5$, variance $3.16^2 \approx 9.98$. (b) Learned kernel parameters: long length scale, $\ell = 1.19$, variance $4.79^2 \approx 22.9$. Generated by [gpc_demo_2d.ipynb](#).

Once we have a Gaussian posterior $q(\mathbf{f}_X | \mathcal{D})$, we can then use standard GP prediction to compute $q(f_* | \mathbf{x}_*, \mathcal{D})$. Finally, we can approximate the posterior predictive distribution over binary labels using

$$\pi_* = p(y_* = 1 | \mathbf{x}_*, \mathcal{D}) = \int p(y_* = 1 | f_*) q(f_* | \mathbf{x}_*, \mathcal{D}) df_* \quad (18.105)$$

This 1d integral can be computed using the probit approximation from Section 15.3.6. In this case we have $\pi_* \approx \sigma(\kappa(v)\mathbb{E}[f_*])$, where $v = \mathbb{V}[f_*]$ and $\kappa^2(v) = (1 + \pi v / 8)^{-1}$.

In Figure 18.9, we show a synthetic binary classification problem in 2d. We use an SE kernel. On the left, we show predictions using hyper-parameters set by hand; we use a short length scale, hence the very sharp turns in the decision boundary. On the right, we show the predictions using the learned hyper-parameters; the model favors more parsimonious explanation of the data.

18.4.2 Multiclass classification

The multi-class case is somewhat harder, since the function now needs to return a vector of C logits to get $p(y_n | \mathbf{x}_n) = \text{Cat}(y_n | \text{softmax}(\mathbf{f}_n))$, 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].

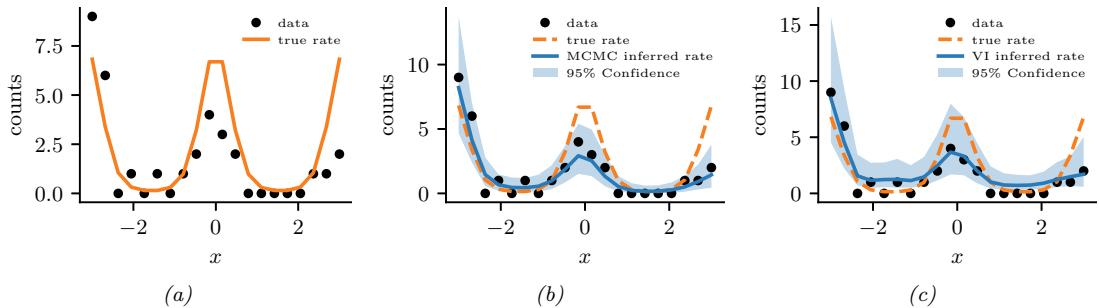


Figure 18.10: Poisson regression with a GP. (a) Observed data (black dots) and true log rate function (yellow line). (b) Posterior predictive distribution (shading shows 1 and 2 σ bands) from MCMC. (c) Posterior predictive distribution from SVI. Generated by [gp_poisson_1d.ipynb](#).

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 Matérn $\frac{5}{2}$ 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 $20\text{km} \times 20\text{km}$ 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 Matérn kernel (Section 18.2.1.3) 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.

18.4.4 Other likelihoods

Many other likelihoods are possible. For example, [VJV09] uses a Student t likelihood in order to perform robust regression. A general method for performing approximate variational inference in GPs with such non-conjugate likelihoods is discussed in [WSS21].

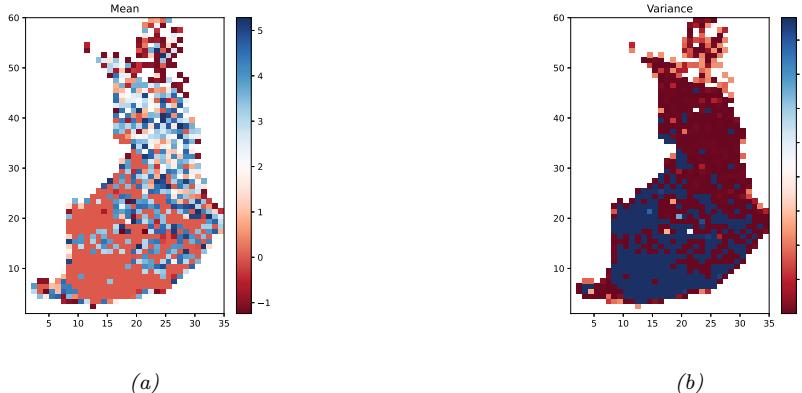


Figure 18.11: We show the relative risk of heart disease in Finland using a Poisson GP fit to 911 data points. Left: posterior mean. Right: posterior variance. Generated by `gp_spatial_demo.ipynb`.

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 18.3: 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, and L is number of Lanczos iterations. Based on Table 2 of [Gar+18a].

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 section, we discuss methods to scale up GPs to handle large N . See Table 18.3 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

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].

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 analyzed 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].)

18.5.1.2 Discussion

The main problem with the subset of data approach is that it ignores some of the data, which can reduce predictive accuracy and increase uncertainty about the true function. Fortunately there are other scalable methods that avoid this problem, essentially by approximately representing (or compressing) the training data, as we discuss below.

18.5.2 Nyström approximation

Suppose we had a rank M approximation to the $N \times N$ matrix gram matrix of the following form:

$$\mathbf{K}_{X,X} \approx \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top \quad (18.106)$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of the M leading eigenvalues, and \mathbf{U} is the matrix of the corresponding M eigenvectors, each of size N . In this case, we can use the matrix inversion lemma to write

$$\mathbf{K}_\sigma^{-1} = (\mathbf{K}_{X,X} + \sigma^2 \mathbf{I}_N)^{-1} \approx \sigma^{-2} \mathbf{I}_N + \sigma^{-2} \mathbf{U}(\sigma^2 \boldsymbol{\Lambda}^{-1} + \mathbf{U}^\top \mathbf{U})^{-1} \mathbf{U}^\top \quad (18.107)$$

which takes $O(NM^2)$ time. Similarly, one can show (using the Sylvester determinant lemma) that

$$|\mathbf{K}_\sigma| \approx |\boldsymbol{\Lambda}| |\sigma^2 \boldsymbol{\Lambda}^{-1} + \mathbf{U}^\top \mathbf{U}| \quad (18.108)$$

which also takes $O(NM^2)$ time.

Unfortunately, directly computing such an eigendecomposition takes $O(N^3)$ time, which does not help. However, suppose we pick a subset Z of $M < N$ points. We can partition the Gram matrix as

follows (where we assume the chosen points come first, and then the remaining points):

$$\mathbf{K}_{X,X} = \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,X-Z} \\ \mathbf{K}_{X-Z,Z} & \mathbf{K}_{X-Z,X-Z} \end{pmatrix} \triangleq \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,\tilde{X}} \\ \mathbf{K}_{\tilde{X},Z} & \mathbf{K}_{\tilde{X},\tilde{X}} \end{pmatrix} \quad (18.109)$$

where $\tilde{X} = X - Z$. We now compute an 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 these to approximate the full matrix as shown below, where the scaling constants are chosen so that $\|\tilde{\mathbf{u}}_i\| \approx 1$:

$$\tilde{\lambda}_i \triangleq \frac{N}{M} \lambda_i \quad (18.110)$$

$$\tilde{\mathbf{u}} \triangleq \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{\tilde{X},Z} \mathbf{u}_i \quad (18.111)$$

$$\mathbf{K}_{X,X} \approx \sum_{i=1}^M \tilde{\lambda}_i \tilde{\mathbf{u}}_i \tilde{\mathbf{u}}_i^\top \quad (18.112)$$

$$= \sum_{i=1}^M \frac{N}{M} \lambda_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{\tilde{X},Z} \mathbf{u}_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{u}_i^\top \mathbf{K}_{\tilde{X},Z}^\top \quad (18.113)$$

$$= \mathbf{K}_{\tilde{X},Z} \left(\sum_{i=1}^M \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^\top \right) \mathbf{K}_{\tilde{X},Z}^\top \quad (18.114)$$

$$= \mathbf{K}_{\tilde{X},Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{\tilde{X},Z}^\top \quad (18.115)$$

This is known as the **Nyström approximation** [WS01]. If we define

$$\mathbf{Q}_{A,B} \triangleq \mathbf{K}_{A,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,B} \quad (18.116)$$

then we can write the approximate Gram matrix as $\mathbf{Q}_{X,X}$. We can then replace \mathbf{K}_σ with $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$. Computing the eigendecomposition takes $O(M^3)$ time, and computing $\hat{\mathbf{Q}}_{X,X}^{-1}$ takes $O(NM^2)$ time. Thus complexity is now linear in N instead of cubic.

If we are approximating *only* $\hat{\mathbf{K}}_{X,X}$ in $\mu_{*,|X}$ in Equation (18.52) and $\Sigma_{*,|X}$ in Equation (18.53), then this is inconsistent with the other un-approximated kernel function evaluations in these formulae, and can result in the predictive variance being negative. One solution to this is to use the same \mathbf{Q} approximation for all terms.

18.5.3 Inducing point methods

In this section, we discuss an approximation method based on **inducing points**, also called **pseudoinputs**, which are like a learned summary of the training data that we can condition on, rather than conditioning on all of it.

Let \mathbf{X} be the observed inputs, and $\mathbf{f}_X = f(\mathbf{X})$ be the unknown vector of function values (for which we have noisy observations \mathbf{y}). Let \mathbf{f}_* be the unknown function values at one or more test points \mathbf{X}_* . Finally, let us assume we have M additional inputs, \mathbf{Z} , with unknown function values \mathbf{f}_Z (often denoted by \mathbf{u}). The exact joint prior has the form

$$p(\mathbf{f}_X, \mathbf{f}_*) = \int p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) d\mathbf{f}_Z = \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) d\mathbf{f}_Z = \mathcal{N} \left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix} \right) \quad (18.117)$$



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 Quiñonero Candela.

(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 into the random function f .)

We will choose \mathbf{f}_Z in such a way that it acts as a sufficient statistic for the data, so that we can predict 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 as follows:

$$p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) = p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z) \approx p(\mathbf{f}_* | \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z) \quad (18.118)$$

See Figure 18.12 for an illustration of this assumption, and Section 18.5.3.4 for details on how to choose the inducing set \mathbf{Z} . (Note that this method is often called a “**sparse GP**”, because it makes predictions for \mathbf{f}_* using a subset of the training data, namely \mathbf{f}_Z , instead of all of it, \mathbf{f}_X .)

From this, we can derive the following train and test conditionals

$$p(\mathbf{f}_X | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{f}_Z, \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.119)$$

$$p(\mathbf{f}_* | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_* | \mathbf{K}_{*,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{f}_Z, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}) \quad (18.120)$$

The above equations can be seen as exact inference on noise-free observations \mathbf{f}_Z . To gain computational speedups, we will make further approximations to the terms $\tilde{\mathbf{Q}}_{X,X} = \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}$ and $\tilde{\mathbf{Q}}_{*,*} = \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$, as we discuss below. We can then derive the approximate prior $q(\mathbf{f}_X, \mathbf{f}_*) = \int q(\mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_* | \mathbf{f}_Z)p(\mathbf{f}_Z)d\mathbf{f}_Z$, which we then condition on the observations in the usual way.

All of the approximations we discuss below result in an initial training cost of $O(M^3 + NM^2)$, and then take $O(M)$ time for the predictive mean for each test case, and $O(M^2)$ time for the predictive variance. (Compare this to $O(N^3)$ training time and $O(N)$ and $O(N^2)$ testing time for exact inference.)

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

$$q_{\text{SOR}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{Q}_{*,*} \end{pmatrix}) \quad (18.121)$$

Let us define $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$, and $\Sigma = (\sigma^{-2} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} + \mathbf{K}_{Z,Z})^{-1}$. Then the predictive distribution is

$$q_{\text{SOR}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{Q}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.122)$$

$$= \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,*} \Sigma \mathbf{K}_{Z,*}) \quad (18.123)$$

This 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

$$\mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) = \mathcal{K}(\mathbf{x}_i, \mathbf{Z}) \mathbf{K}_{Z,Z}^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_j) \quad (18.124)$$

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

$$q_{\text{dtc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.125)$$

Hence the predictive distribution becomes

$$q_{\text{dtc}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.126)$$

$$= \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*} + \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,*}) \quad (18.127)$$

The predictive mean is the same as in SOR, but the variance is larger (since $\mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ is positive definite) due to the uncertainty of \mathbf{f}_* given \mathbf{f}_Z .

18.5.3.3 FITC

A widely used approximation assumes $q(\mathbf{f}_X | \mathbf{f}_Z)$ is fully factorized, i.e,

$$q(\mathbf{f}_X | \mathbf{f}_Z) = \prod_{n=1}^N p(f_n | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{f}_Z, \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X})) \quad (18.128)$$

This is called the **fully independent training conditional** or **FITC** assumption, and was first proposed in [SG06a]. This throws away less uncertainty than the SOR and DTC methods, since it does not make any deterministic assumptions about the relationship between \mathbf{f}_X and \mathbf{f}_Z .

The joint prior has the form

$$q_{\text{fitc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} - \text{diag}(\mathbf{Q}_{X,X} - \mathbf{K}_{X,X}) & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.129)$$

The predictive distribution for a single test case is given by

$$q_{\text{fitc}}(f_* | \mathbf{y}) = \mathcal{N}(f_* | \mathbf{k}_{*,Z} \boldsymbol{\Sigma} \mathbf{K}_{Z,X} \boldsymbol{\Lambda}^{-1} \mathbf{y}, k_{**} - q_{**} + \mathbf{k}_{*,Z} \boldsymbol{\Sigma} \mathbf{k}_{Z,*}) \quad (18.130)$$

where $\boldsymbol{\Lambda} \triangleq \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N)$, and $\boldsymbol{\Sigma} \triangleq (\mathbf{K}_{Z,Z} + \mathbf{K}_{Z,X} \boldsymbol{\Lambda}^{-1} \mathbf{K}_{X,Z})^{-1}$. If we have a batch of test cases, we can assume they are conditionally independent (an approach known as **fully independent conditional** or **FIC**), and multiply the above equation.

The computational cost is the same as for SOR and DTC, but the approach avoids some of the pathologies due to a non-degenerate kernel. In particular, one can show that the FIC method is equivalent to exact GP inference with the following non-degenerate kernel:

$$\mathcal{K}_{\text{fic}}(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i = j \\ \mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i \neq j \end{cases} \quad (18.131)$$

18.5.3.4 Learning the inducing points

So far, we have not specified how to choose the inducing points or pseudoinputs \mathbf{Z} . We can treat these like kernel hyperparameters, and choose them so as to maximize the log marginal likelihood, given by

$$\log q(\mathbf{y} | \mathbf{X}, \mathbf{Z}) = \log \int \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{f}_Z) p(\mathbf{f}_Z | \mathbf{Z}) d\mathbf{f}_Z d\mathbf{f} \quad (18.132)$$

$$= \log \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{Z}) d\mathbf{f}_X \quad (18.133)$$

$$= -\frac{1}{2} \log |\mathbf{Q}_{X,X} + \boldsymbol{\Lambda}| - \frac{1}{2} \mathbf{y}^\top (\mathbf{Q}_{X,X} + \boldsymbol{\Lambda})^{-1} \mathbf{y} - \frac{n}{2} \log(2\pi) \quad (18.134)$$

where the definition of $\boldsymbol{\Lambda}$ depends on the method, namely $\boldsymbol{\Lambda}_{\text{SOR}} = \boldsymbol{\Lambda}_{\text{dtc}} = \sigma^2 \mathbf{I}_N$, and $\boldsymbol{\Lambda}_{\text{fitc}} = \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) + \sigma^2 \mathbf{I}_N$.

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 18.2.3). 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 12.9.1), 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 called the **sparse variational GP** or **SVGP** approximation, also known as the **variational free energy** or **VFE** approach [Tit09; Mat+16]. This is similar to the inducing point methods in Section 18.5.3, except it approximates the posterior, rather than approximating the prior. The variational approach can also easily handle non-conjugate likelihoods, as we will see. For more details, see e.g., [BWR16; Lei+20]. (See also [WKS21] for connections between SVGP and the Nyström method.)

To explain the idea behind SVGP/ VFE, 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

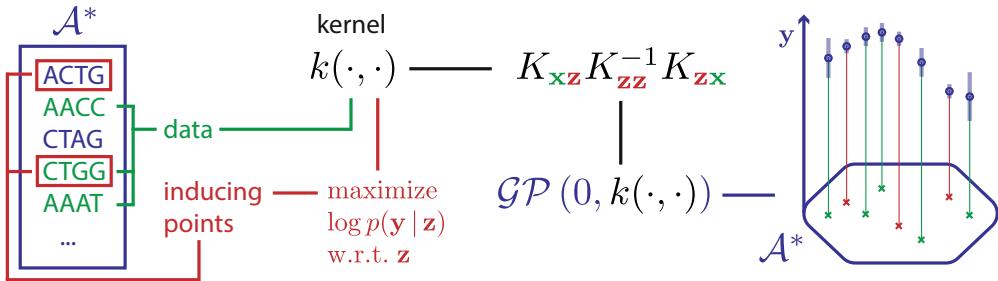


Figure 18.13: Illustration of how to choose inducing points from a discrete input domain (here DNA sequences of length 4) to maximize the log marginal likelihood. From Figure 1 of [For+18a]. Used with kind permission of Vincent Fortuin.

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 $p(\mathbf{f}) = \mathcal{N}(m(\mathcal{X}), \mathcal{K}(\mathcal{X}, \mathcal{X}))$.

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.

In the VFE view, the inducing points \mathbf{Z} and inducing variables \mathbf{f}_Z (often denoted by \mathbf{u}) are variational parameters, rather than model parameters, which avoids the risk of overfitting. Furthermore, one can show that as the number of inducing points m increases, the quality of the posterior consistently improves, eventually recovering exact inference. By contrast, in the classical inducing point method, increasing m does not always result in better performance [BWR16].

In more detail, the VFE approach tries to find an approximate posterior $q(\mathbf{f})$ to minimize $D_{\text{KL}}(q(\mathbf{f}) \| p(\mathbf{f} | \mathbf{y}))$. 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 $p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z)$ is computed exactly using the GP prior, and $q(\mathbf{f}_Z)$ is learned, by minimizing $\mathcal{K}(q) = D_{\text{KL}}(q(\mathbf{f}) \| p(\mathbf{f} | \mathbf{y}))$.² Intuitively, $q(\mathbf{f}_Z)$ acts as a “bottleneck” which “absorbs” all the observations from \mathbf{y} ; posterior predictions for elements of \mathbf{f}_X or \mathbf{f}_* are then made via their dependence on \mathbf{f}_Z , rather than their dependence on each other.

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].

We can derive the form of the loss, which is used to compute the posterior $q(\mathbf{f}_Z)$, as follows:

$$\mathcal{K}(q) = D_{\text{KL}}(q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \| p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z | \mathbf{y})) \quad (18.135)$$

$$= \int q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \log \frac{q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z)}{p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z | \mathbf{y})} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.136)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log \frac{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) p(\mathbf{y})}{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) p(\mathbf{y} | \mathbf{f}_X)} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.137)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z) p(\mathbf{y})}{p(\mathbf{f}_Z) p(\mathbf{y} | \mathbf{f}_X)} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.138)$$

$$= \int q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z)}{p(\mathbf{f}_Z)} d\mathbf{f}_Z - \int p(\mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log p(\mathbf{y} | \mathbf{f}_X) d\mathbf{f}_X d\mathbf{f}_Z + C \quad (18.139)$$

$$= D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) - \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] + C \quad (18.140)$$

where $C = \log p(\mathbf{y})$ is an irrelevant constant.

We can alternatively write the objective as an evidence lower bound that we want to maximize:

$$\log p(\mathbf{y}) = \mathcal{K}(q) + \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.141)$$

$$\geq \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \triangleq \mathcal{L}(q) \quad (18.142)$$

Now suppose we choose a Gaussian posterior approximation, $q(\mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_Z | \mathbf{m}, \mathbf{S})$. Since $p(\mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_Z | \mathbf{0}, \mathcal{K}(\mathbf{Z}, \mathbf{Z}))$, we can compute the KL term in closed form using the formula for KL divergence between Gaussians (Equation (5.77)). To compute the expected log-likelihood term, we first need to compute the induced posterior over the latent function values at the training points:

$$q(\mathbf{f}_X | \mathbf{m}, \mathbf{S}) = \int p(\mathbf{f}_X | \mathbf{f}_Z, \mathbf{X}, \mathbf{Z}) q(\mathbf{f}_Z | \mathbf{m}, \mathbf{S}) d\mathbf{f}_Z = \mathcal{N}(\mathbf{f}_X | \tilde{\mathbf{\mu}}, \tilde{\mathbf{\Sigma}}) \quad (18.143)$$

$$\tilde{\mu}_i = m(\mathbf{x}_i) + \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathbf{m} - m(\mathbf{Z})) \quad (18.144)$$

$$\tilde{\Sigma}_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) - \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathcal{K}(\mathbf{Z}, \mathbf{Z}) - \mathbf{S}) \boldsymbol{\alpha}(\mathbf{x}_j) \quad (18.145)$$

$$\boldsymbol{\alpha}(\mathbf{x}_i) = \mathcal{K}(\mathbf{Z}, \mathbf{Z})^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_i) \quad (18.146)$$

Hence the marginal at a single point is $q(f_n) = \mathcal{N}(f_n | \tilde{\mu}_n, \tilde{\Sigma}_{nn})$, which we can use to compute the expected log likelihood:

$$\mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] = \sum_{n=1}^N \mathbb{E}_{q(f_n)} [\log p(y_n | f_n)] \quad (18.147)$$

We discuss how to compute these expectations 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

$$\mathbb{E}_{q(f_n)} [\log \mathcal{N}(y_n | f_n, \beta^{-1})] = \log \mathcal{N}(y_n | \mathbf{k}_n^T \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1}) - \frac{1}{2} \beta \tilde{k}_{nn} - \frac{1}{2} \text{tr}(\mathbf{S} \mathbf{\Lambda}_n) \quad (18.148)$$

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 $\Lambda_n = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{k}_n \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1}$.

Hence the overall ELBO has the form

$$\mathcal{L}(q) = \log \mathcal{N}(\mathbf{y} | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1} \mathbf{I}_N) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{S} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}) \quad (18.149)$$

$$- \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.150)$$

where $\mathbf{Q}_{X,X} = \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}$.

To compute the gradients of this, we leverage the following result [OA09]:

$$\frac{\partial}{\partial \mu} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)}[h(x)] = \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)} \left[\frac{\partial}{\partial x} h(x) \right] \quad (18.151)$$

$$\frac{\partial}{\partial \sigma^2} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)}[h(x)] = \frac{1}{2} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)} \left[\frac{\partial^2}{\partial x^2} h(x) \right] \quad (18.152)$$

We then substitute $h(x)$ with $\log p(y_n|f_n)$. Using this, one can show

$$\nabla_{\mathbf{m}} \mathcal{L}(q) = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} - \Lambda \mathbf{m} \quad (18.153)$$

$$\nabla_{\mathbf{S}} \mathcal{L}(q) = \frac{1}{2} \mathbf{S}^{-1} - \frac{1}{2} \Lambda \quad (18.154)$$

Setting the derivatives to zero gives the optimal solution:

$$\mathbf{S} = \Lambda^{-1} \quad (18.155)$$

$$\Lambda = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} + \mathbf{K}_{Z,Z}^{-1} \quad (18.156)$$

$$\mathbf{m} = \beta \Lambda^{-1} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} \quad (18.157)$$

This is called **sparse GP regression** or **SGPR** [Tit09].

With these parameters, the lower bound on the log marginal likelihood is given by

$$\log p(\mathbf{y}) \geq \log \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} + \beta^{-1} \mathbf{I}) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.158)$$

(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 (18.158) is almost the same as the log marginal likelihood for the DTC model in Equation (18.134), 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 (18.151) and Equation (18.152). In the case of a binary classifier, we can use the results in Table 18.2 to compute the inner $\frac{\partial}{\partial f_n} h(f_n)$ and $\frac{\partial^2}{\partial f_n^2} h(f_n)$ terms. Alternatively, we can use numerical integration techniques, such as those discussed in Section 8.5.1.4. (See also [WSS21].)

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 10.1.4). The key observation is that the log likelihood in Equation (18.147) 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

$$\mathcal{L}(q) = \left[\frac{N}{B} \sum_{b=1}^B \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \mathbb{E}_{q(f_n)} [\log p(y_n | f_n)] \right] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.159)$$

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 (18.159) 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.

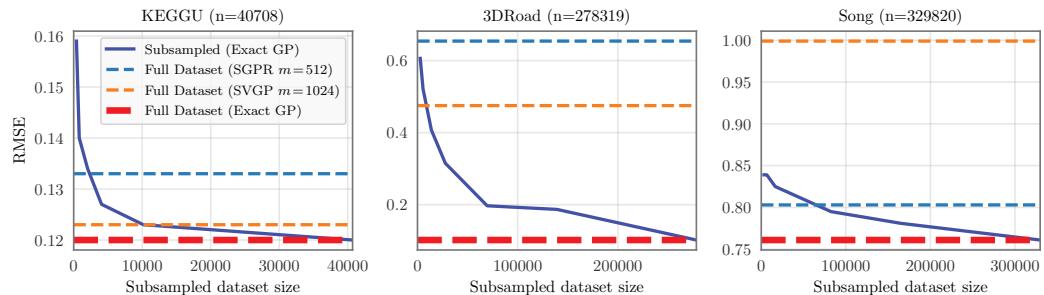


Figure 18.14: RMSE on test set as a function of training set size using a GP with Matern 3/2 kernel with shared lengthscale across all dimensions. Solid lines: exact inference. Dashed blue: SGPR method (closed-form batch solution to the Gaussian variational approximation) of Section 18.5.4.1 with $M = 512$ inducing points. Dashed orange: SVGP method (SGD on Gaussian variational approxiation) 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.

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

$$\mathbf{K}_{X,X} \approx \mathbf{W}_X \mathbf{K}_{Z,Z} \mathbf{W}_X^\top \quad (18.160)$$

where $\mathbf{W}_\mathbf{X}$ is a sparse matrix containing interpolation weights. If we use cubic interpolation, each row only has 4 nonzeros. Thus we can compute $(\mathbf{W}_\mathbf{X} \mathbf{K}_{Z,Z} \mathbf{W}_\mathbf{X}^\top) \mathbf{v}$ 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 $C \sim 10^2$.

18.5.5.4 Tensor train methods

Consider the Gaussian VFE approach in Section 18.5.4. We have to estimate the covariance \mathbf{S} and the mean \mathbf{m} . We can represent \mathbf{S} efficiently using Kronecker structure, as used by KISS. Additionally, we can represent \mathbf{m} efficiently using the **tensor train decomposition** [Ose11] in combination with **SKI** [WN15]. The resulting **TT-GP** method can scale efficiently to billions of inducing points, as explained in [INK18].

18.5.6 Converting a GP to an SSM

Consider a function defined on a 1d scalar input, such as a time index. For many stationary 1d kernels, the corresponding GP can be modeled using a linear time invariant (LTI) stochastic differential equation (SDE)³; this SDE can then be converted to a linear-Gaussian state space model (Section 29.1) as first proposed in [HS10]. For example, consider the exponential kernel in Equation (18.14), $\mathcal{K}(t, t') = \frac{q}{2\lambda} \exp(-\lambda|t - t'|)$, which corresponds to a Matérn kernel with $\nu = 1/2$. The corresponding SDE is the Orstein-Uhlenbeck process which has the form $\frac{dx(t)}{dt} = -\lambda x(t) + w(t)$, where $w(t)$ is a white noise process with spectral density q [SS19, p258].⁴ For other kernels (such as Matérn with $\nu = 3/2$), we need to use multiple latent states in order to capture higher order

3. The condition is that the spectral density of the covariance function has to be a rational function. This includes many kernels, such as the Matérn kernel, but excludes the squared exponential (RBF) kernel. However the latter can be approximated by an SDE, as explained in [SS19, p261].

4. This is sometimes written as $dx = -\lambda x dt + d\beta$, where $\beta(t)$ is a Brownian noise process, and $w(t) = \frac{d\beta(t)}{dt}$, as explained in [SS19, p45].

derivative terms (see [Supplementary](#) Section 18.2 for details). Furthermore, for higher dimensional inputs, we need to use even more latent states, to enforce the Markov property [DSP21].

Once we have converted the GP to LG-SSM form, we can perform exact inference in $O(N)$ time using Kalman smoothing, as explained in Section 8.2.3. Furthermore, if we have access to a highly parallel processor, such as a GPU, we can reduce the time to $\log(N)$ [CZS22], as explained in Section 8.2.3.4.

18.6 Learning the kernel

In [Mac98], David MacKay asked: “How can Gaussian processes replace neural networks? Have we thrown the baby out with the bathwater?” This remark was made in the late 1990s, at the end of the second wave of neural networks. Researchers and practitioners had grown weary of the design decisions associated with neural networks — such as activation functions, optimization procedures, architecture design — and the lack of a principled framework to make these decisions. Gaussian processes, by contrast, were perceived as flexible and principled probabilistic models, which naturally followed from Radford Neal’s results on infinite neural networks [Nea96], which we discuss in more depth in Section 18.7.

However, MacKay [Mac98] noted that neural networks could discover rich representations of data through adaptive hidden basis functions, while Gaussian processes with standard kernel functions, such as the RBF kernel, are essentially just smoothing devices. Indeed, the generalization properties of Gaussian processes hinge on the suitability of the kernel function. *Learning* the kernel is how we do representation learning with Gaussian processes, and in many cases will be crucial for good performance — especially when we wish to perform extrapolation, making predictions far away from the data [WA13; Wil+14].

As we will see, learning a kernel is in many ways analogous to training a neural network. Moreover, neural networks and Gaussian processes can be synergistically combined through approaches such as deep kernel learning (see Section 18.6.6) and NN-GPs (Section 18.7.2).

18.6.1 Empirical Bayes for the kernel parameters

Suppose, as in Section 18.3.2, we are performing 1d regression using a GP with an RBF kernel. Since the data has observation noise, the kernel has the following form:

$$\mathcal{K}_y(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x_p - x_q)^2\right) + \sigma_y^2 \delta_{pq} \quad (18.161)$$

Here ℓ is the horizontal scale over which the function changes, σ_f^2 controls the vertical scale of the function, and σ_y^2 is the noise variance. Figure 18.15 illustrates the effects of changing these parameters. We sampled 20 noisy datapoints from the SE kernel using $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and then made predictions various parameters, conditional on the data. In Figure 18.15(a), we use $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and the result is a good fit. In Figure 18.15(b), we increase the length scale to $\ell = 3$; now the function looks smoother, but we are arguably underfitting.

To estimate the kernel parameters θ (sometimes called hyperparameters), we could use exhaustive search over a discrete grid of values, with validation loss as an objective, but this can be quite slow. (This is the approach used by nonprobabilistic methods, such as SVMs, to tune kernels.) Here we

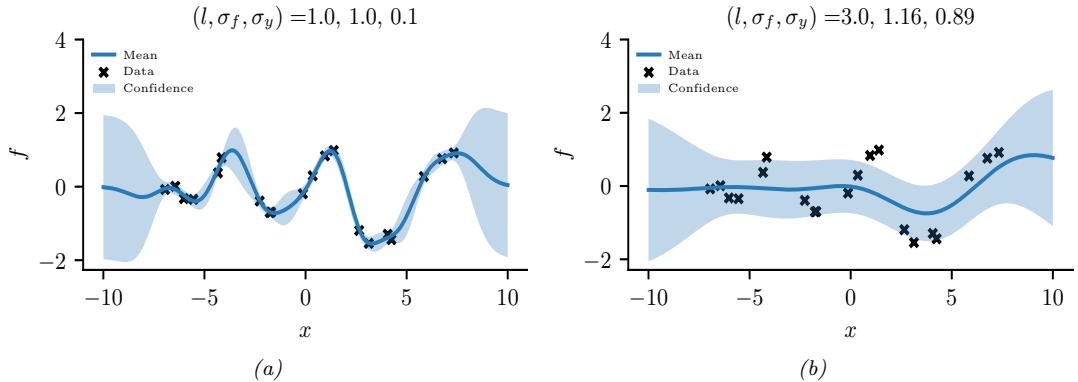


Figure 18.15: Some 1d GPs with RBF kernels but different hyper-parameters fit to 20 noisy observations. The hyper-parameters $(\ell, \sigma_f, \sigma_y)$ are as follows: (a) $(1, 1, 0.1)$ (b) $(3.0, 1.16, 0.89)$. Adapted from Figure 2.5 of [RW06]. Generated by [gpr_demo_change_hparams.ipynb](#).

consider an empirical Bayes approach, which will allow us to use continuous optimization methods, which are much faster. In particular, we will maximize the marginal likelihood

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X})p(\mathbf{f}|\mathbf{X}, \boldsymbol{\theta})d\mathbf{f} \quad (18.162)$$

(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

$$\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_\sigma) = -\frac{1}{2}\mathbf{y}^\top \mathbf{K}_\sigma^{-1} \mathbf{y} - \frac{1}{2} \log |\mathbf{K}_\sigma| - \frac{N}{2} \log(2\pi) \quad (18.163)$$

where the dependence of \mathbf{K}_σ on $\boldsymbol{\theta}$ 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.

We now discuss how to maximize the marginal likelihood. One can show that

$$\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \frac{1}{2} \mathbf{y}^\top \mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j} \mathbf{K}_\sigma^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j}) \quad (18.164)$$

$$= \frac{1}{2} \text{tr} \left((\boldsymbol{\alpha} \boldsymbol{\alpha}^\top - \mathbf{K}_\sigma^{-1}) \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j} \right) \quad (18.165)$$

where $\boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1} \mathbf{y}$. 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.

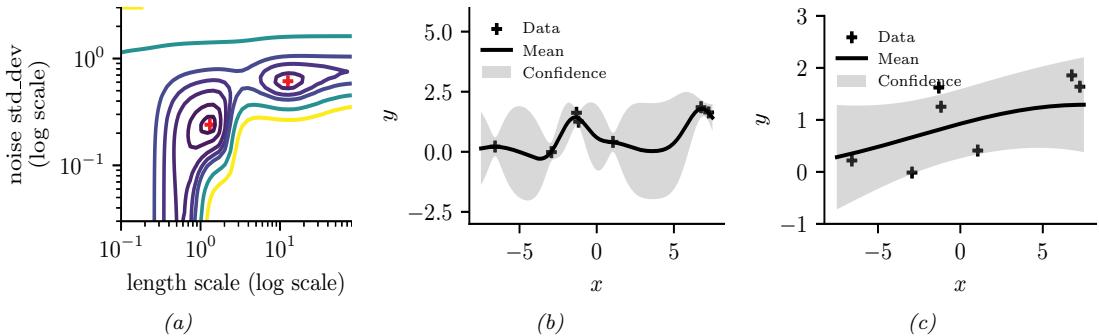


Figure 18.16: Illustration of local minima in the marginal likelihood surface. (a) We plot the log marginal likelihood vs σ_y^2 and ℓ , for fixed $\sigma_f^2 = 1$, using the 7 datapoints shown in panels b and c. (b) The function corresponding to the lower left local minimum, $(\ell, \sigma_n^2) \approx (1, 0.2)$. 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 [gpr_demo_marglik.ipynb](#).

The form of $\frac{\partial \mathbf{K}_\alpha}{\partial \theta_j}$ 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 (18.161) 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 datapoints 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 datapoints, 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 18.16 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.

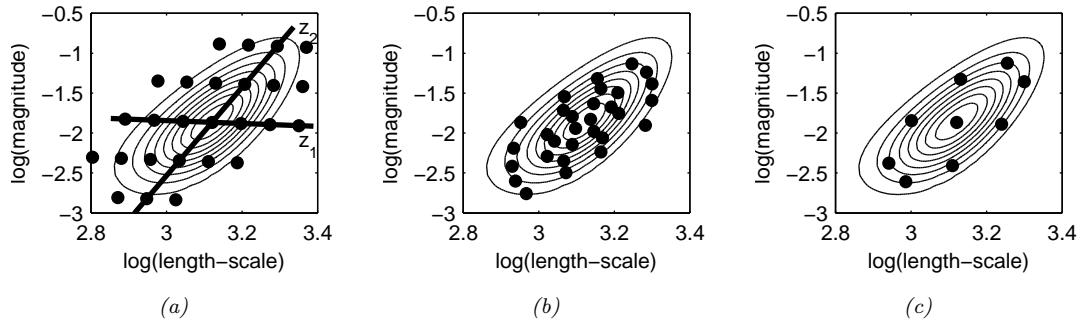


Figure 18.17: Three different approximations to the posterior over hyper-parameters: grid-based, Monte Carlo, and central composite design. From Figure 3.2 of [Van10]. Used with kind permission of Jarno Vanhatalo.

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 6.6), 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.⁵

To overcome such overconfidence, we can compute a posterior over the kernel parameters. If the dimensionality of θ is small, we can compute a discrete grid of possible values, centered on the MAP estimate $\hat{\theta}$ (computed as above). We can then approximate the posterior using

$$p(\mathbf{f}|\mathcal{D}) = \sum_{s=1}^S p(\mathbf{f}|\mathcal{D}, \theta_s) p(\theta_s|\mathcal{D}) w_s \quad (18.166)$$

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 place grid points at the mode, and at a distance $\pm 1\text{sd}$ from the mode along each dimension, for a total of $2|\theta| + 1$ 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 Equation (18.166). 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

$$\mathcal{K}(r) = \sigma_1^2 \mathcal{K}_{\text{SE}}(r; \tau) \mathcal{K}_{\text{cos}}(r; \rho_1) + \sigma_2^2 \mathcal{K}_{32}(r; \rho_2) \quad (18.167)$$

where $\mathcal{K}_{\text{SE}}(r; \ell)$ is the squared exponential kernel (Equation (18.12)), $\mathcal{K}_{\text{cos}}(r; \rho_1)$ is the cosine kernel (Equation (18.19)), and $\mathcal{K}_{32}(r; \rho_2)$ is the Matérn $\frac{3}{2}$ kernel (Equation (18.15)). We then compute a

5. 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.

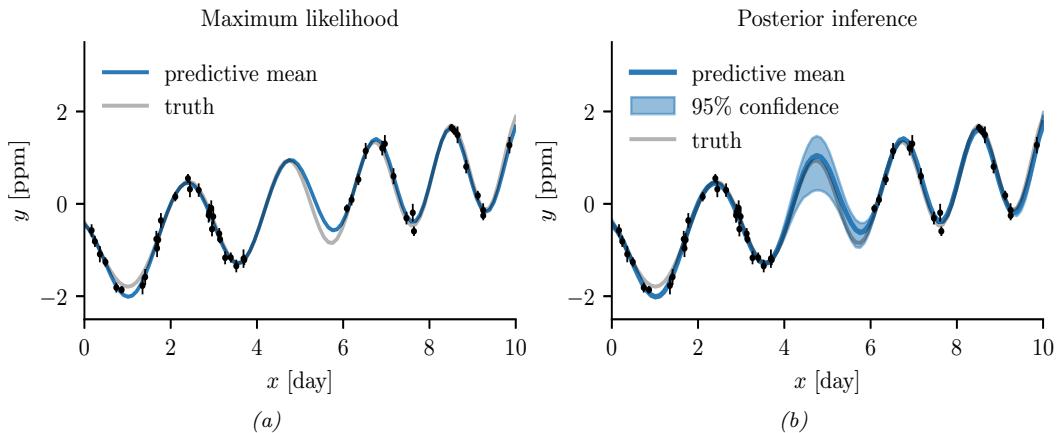


Figure 18.18: Difference between estimation and inference for kernel hyper-parameters. (a) Empirical Bayes approach based on optimization. We plot the posterior predicted mean given a plug-in estimate, $\mathbb{E}[f(x)|\mathcal{D}, \hat{\theta}]$. (b) Bayesian approach based on HMC. We plot the posterior predicted mean, marginalizing over hyper-parameters, $\mathbb{E}[f(x)|\mathcal{D}]$. Generated by [gp_kernel_opt.ipynb](#).

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

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{x}, \mathbf{x}') \quad (18.168)$$

Optimizing the weights $w_b > 0$ using structural risk minimization is known as **multiple kernel learning**; see e.g., [Rak+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 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:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^D \mathcal{K}_d(x_d, x'_d) \quad (18.169)$$

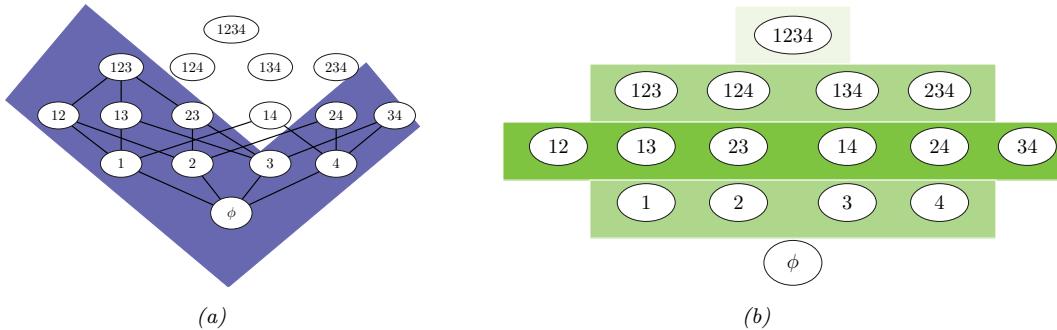


Figure 18.19: Comparison of different additive model classes for a 4d function. Circles represent different interaction terms, ranging from first-order to fourth-order. Left: hierarchical kernel learning uses a nested hierarchy of terms. Right: additive GPs use a weighted sum of additive kernels of different orders. Color shades represent different weighting terms. Adapted from Figure 6.2 of [Duv14].

The resulting function then has the form

$$f(\mathbf{x}) = f_1(x_1) + \dots + f_D(x_D) \quad (18.170)$$

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:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{P}_b \mathbf{x}, \mathbf{P}_b \mathbf{x}') \quad (18.171)$$

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.

More recently, [LBH22] has proposed the **orthogonal additive kernel** (OAK), which imposes an orthogonality constraint on the additive functions. This ensures an identifiable, low-dimensional representation of the functional relationship, and results in improved performance.

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. (See also [BHB22] for a review.)

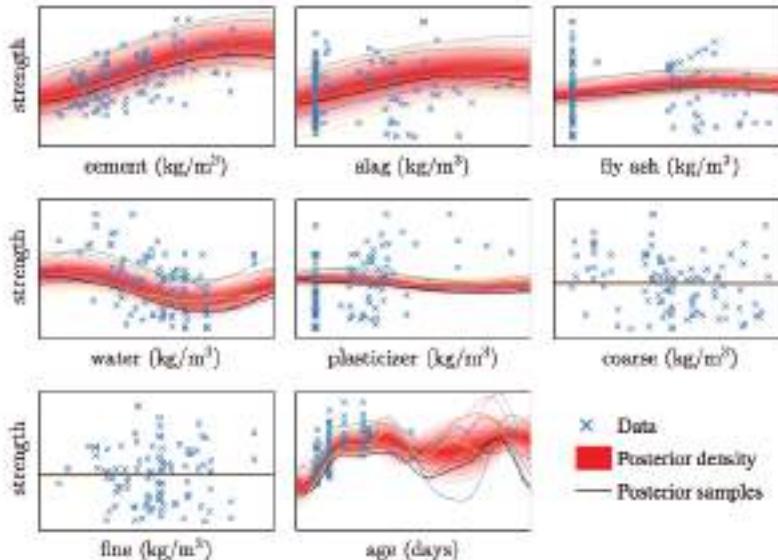


Figure 18.20: Predictive distribution of each term in a GP-GAM model applied to a dataset with 8 continuous inputs and 1 continuous output, representing the strength of some concrete. From Figure 2.7 of [Duv14]. Used with kind permission of David Duvenaud.

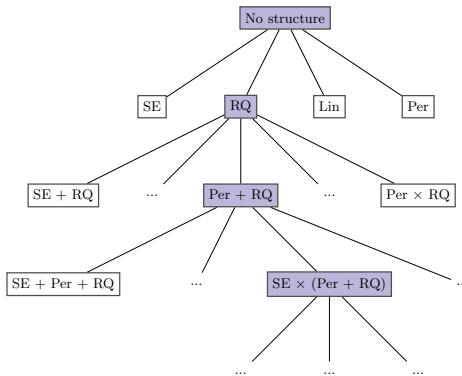


Figure 18.21: Example of a search tree over kernel expressions. Adapted from Figure 3.2 of [Duv14].

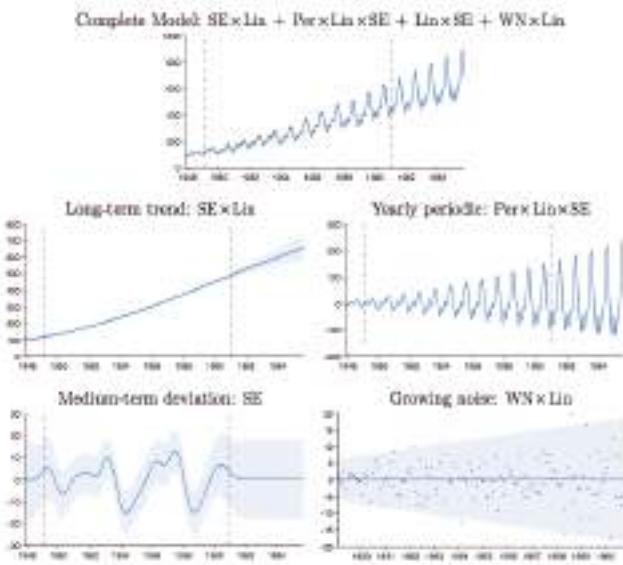


Figure 18.22: Top row: airline dataset and posterior distribution of the model discovered after a search of depth 10. Subsequent rows: predictions of the individual components. From Figure 3.5 of [Duv14], based on [Llo+14]. Used with kind permission of David Duvenaud.

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 3.8.7.2) 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.

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

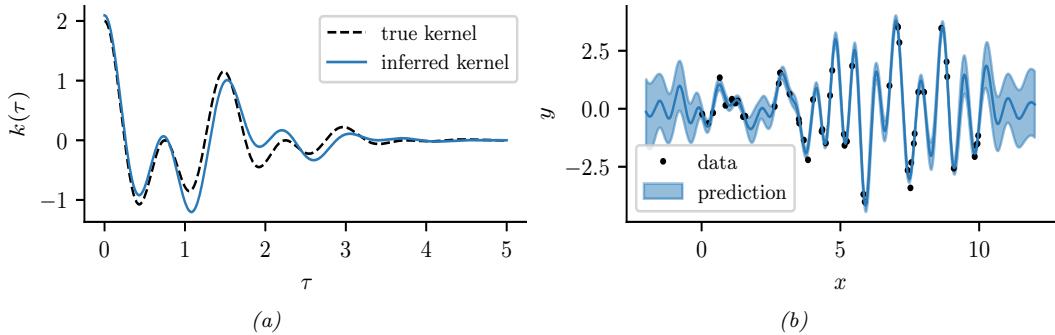


Figure 18.23: Illustration of a GP with a spectral mixture kernel in 1d. (a) Learned vs true kernel. (b) Predictions using learned kernel. Generated by [gp_spectral_mixture.ipynb](#).

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 Matérn kernel, we get a Student spectral density centred at the origin. Thus standard approaches to multiple kernel learning, which typically involve additive compositions of RBF and Matérn kernels with different length-scale parameters, amount to density estimation with a scale mixture of Gaussian or Student 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:

$$\mathcal{K}(x, x') = \sum_i w_i \cos((x - x')(2\pi\mu_i)) \exp(-2\pi^2(x - x')^2 v_i) \quad (18.172)$$

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 Sec-

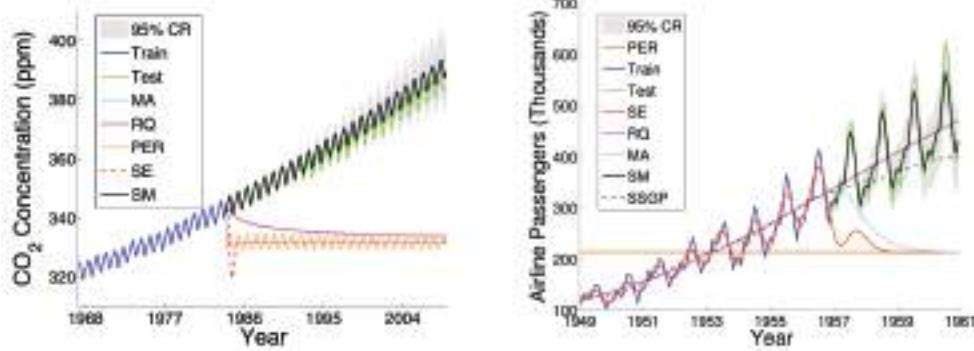


Figure 18.24: Extrapolations (point predictions and 95% credible set) on CO₂ and airline datasets using Gaussian processes with Matérn, rational quadratic, periodic, RBF (SE), and spectral mixture kernels, each with hyperparameters learned using empirical Bayes. From [Wil14].

tion 18.8.1, 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. 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.

18.6.6 Deep kernel learning

Deep kernel learning [SH07; Wil+16] 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:

$$\mathcal{K}_{\theta}(\mathbf{x}, \mathbf{x}') = \exp \left[-\frac{1}{2\sigma^2} \|\mathbf{h}_{\theta}^L(\mathbf{x}) - \mathbf{h}_{\theta}^L(\mathbf{x}')\|^2 \right] \quad (18.173)$$

18.6. Learning the kernel

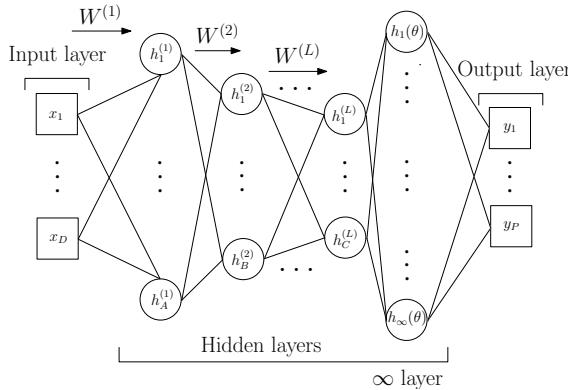


Figure 18.25: Deep kernel learning: a Gaussian process with a deep kernel maps D dimensional inputs \mathbf{x} through L parametric hidden layers followed by a hidden layer with an infinite number of basis functions, with base kernel hyperparameters θ . Overall, a Gaussian process with a deep kernel produces a probabilistic mapping with an infinite number of adaptive basis functions parameterized by $\gamma = \{\mathbf{w}, \theta\}$. All parameters γ are learned through the marginal likelihood of the Gaussian process. From Figure 1 of [Wil+16].

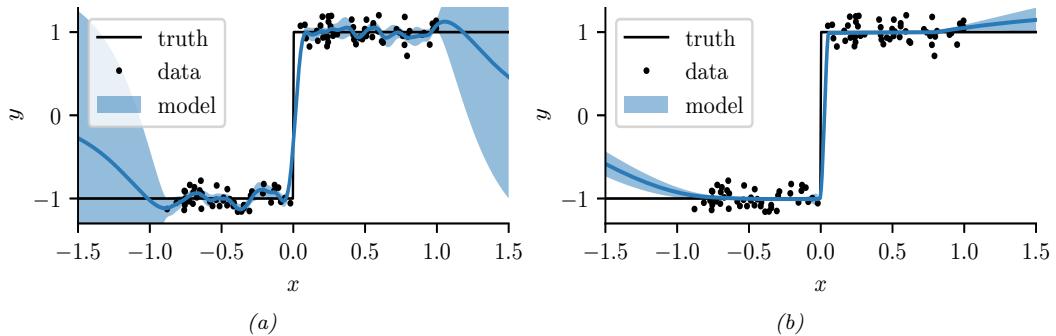


Figure 18.26: Modeling a discontinuous function with (a) a GP with a “shallow” Matérn $\frac{3}{2}$ kernel, and (b) a GP with a “deep” MLP + Matérn kernel. Generated by [gp_deep_kernel_learning.ipynb](#).

where $\mathbf{h}_{\theta}^L(\mathbf{x})$ are the outputs of layer L from a DNN. We can then learn the parameters θ 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 (1) pre-trained, (2) learned jointly with the base kernel parameters, or (3) 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 Matérn $\frac{3}{2}$ kernel. It is clear

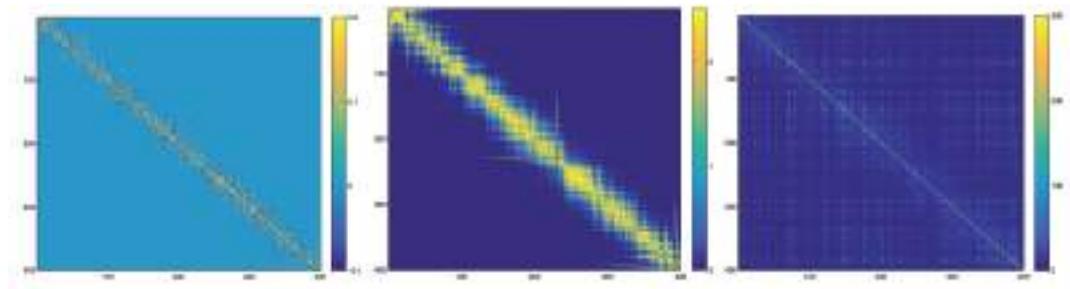


Figure 18.27: Left: the learned covariance matrix of a deep kernel with spectral mixture base kernel on a set of test cases for the Olivetti faces dataset, where the test samples are ordered according to the orientations of the input faces. Middle: the respective covariance matrix using a deep kernel with RBF base kernel. Right: the respective covariance matrix using a standard RBF kernel. From Figure 5 of [Wil+16].

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 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.

However, [ORW21] show that the approach to DKL based on maximizing the marginal likelihood can result in overfitting that is worse than standard DNN learning. They propose a fully Bayesian approach, in which they use SGLD (Section 12.7.1) to sample the DNN weights as well as the GP hyperparameters.

18.7 GPs and DNNs

In Section 18.6.6, we showed how we can combine the structural properties of neural networks with GPs. In Section 18.7.1 we show that, in the limit of infinitely wide networks, a neural network defines a GP with a certain kernel. These kernels are fixed, so the method is not performing representation learning, as a standard neural network would (see e.g., [COB18; Woo+19]). Nonetheless, these kernels are interesting in their own right, for example in modelling non-stationary covariance structure. In Section 18.7.2, we discuss the connection between SGD training of DNNs and GPs. And in Section 18.7.3, we discuss deep GPs, which are similar to DNNs in that they consist of many layers of functions which are composed together, but each layer is a nonparametric function.

18.7.1 Kernels derived from infinitely wide DNNs (NN-GP)

In this section, we show that an MLP with one hidden layer, whose width goes to infinity, and which has a Gaussian prior on all the parameters, converges to a Gaussian process with a well-defined kernel.⁶ This result was first shown for in [Nea96; Wil98], and was later extended to deep MLPs in [DFS16; Lee+18], to CNNs in [Nov+19], and to general DNNs in [Yan19]. The resulting kernel is called the **NN-GP** kernel [Lee+18].

We will consider the following model:

$$f_k(\mathbf{x}) = b_k + \sum_{j=1}^H v_{jk} h_j(\mathbf{x}), \quad h_j(\mathbf{x}) = \varphi(u_{0j} + \mathbf{x}^\top \mathbf{u}_j) \quad (18.174)$$

where H is the number of hidden units, and $\varphi()$ is some nonlinear activation function, such as ReLU. We will assume Gaussian priors on the parameters:

$$b_k \sim \mathcal{N}(0, \sigma_b), \quad v_{jk} \sim \mathcal{N}(0, \sigma_v), \quad u_{0j} \sim \mathcal{N}(0, \sigma_0), \quad \mathbf{u}_j \sim \mathcal{N}(0, \Sigma) \quad (18.175)$$

Let $\boldsymbol{\theta} = \{b_k, v_{jk}, u_{0j}, \mathbf{u}_j\}$ be all the parameters. The expected output from unit k when applied to one input vector is given by

$$\mathbb{E}_{\boldsymbol{\theta}} [f_k(\mathbf{x})] = \mathbb{E}_{\boldsymbol{\theta}} \left[b_k + \sum_{j=1}^H v_{jk} h_j(\mathbf{x}) \right] = \underbrace{\mathbb{E}_{\boldsymbol{\theta}} [b_k]}_{=0} + \sum_{j=1}^H \underbrace{\mathbb{E}_{\boldsymbol{\theta}} [v_{jk}]}_{=0} \mathbb{E}_{\mathbf{u}} [h_j(\mathbf{x})] = 0 \quad (18.176)$$

The covariance in the output for unit k when the function is applied to two different inputs is given by the following:⁷

$$\mathbb{E}_{\boldsymbol{\theta}} [f_k(\mathbf{x}) f_k(\mathbf{x}')] = \mathbb{E}_{\boldsymbol{\theta}} \left[\left(b_k + \sum_{j=1}^H v_{jk} h_j(\mathbf{x}) \right) \left(b_k + \sum_{j=1}^H v_{jk} h_j(\mathbf{x}') \right) \right] \quad (18.177)$$

$$= \sigma_b^2 + \sum_{j=1}^H \mathbb{E}_{\boldsymbol{\theta}} [v_{jk}^2] \mathbb{E}_{\mathbf{u}} [h_j(\mathbf{x}) h_j(\mathbf{x}')] = \sigma_b^2 + \sigma_v^2 H \mathbb{E}_{\mathbf{u}} [h_j(\mathbf{x}) h_j(\mathbf{x}')] \quad (18.178)$$

Now consider the limit $H \rightarrow \infty$. We scale the magnitude of the output by defining $\sigma_v^2 = \omega/H$. Since the input to k 'th output unit is an infinite sum of random variables (from the hidden units $h_j(\mathbf{x})$), we can use the **central limit theorem** to conclude that the output converges to a Gaussian with mean and variance given by

$$\mathbb{E}[f_k(\mathbf{x})] = 0, \quad \mathbb{V}[f_k(\mathbf{x})] = \sigma_b^2 + \omega \mathbb{E}_{\mathbf{u}} [h(\mathbf{x})^2] \quad (18.179)$$

Furthermore, the joint distribution over $\{f_k(\mathbf{x}_n) : n = 1 : N\}$ for any $N \geq 2$ converges to a multivariate Gaussian with covariance given by

$$\mathbb{E}[f_k(\mathbf{x}) f_k(\mathbf{x}')] = \sigma_b^2 + \omega \mathbb{E}_{\mathbf{u}} [h(\mathbf{x}) h(\mathbf{x}')] \triangleq \mathcal{K}(\mathbf{x}, \mathbf{x}') \quad (18.180)$$

6. Our presentation is based on http://cbl.eng.cam.ac.uk/pub/Intranet/MLG/ReadingGroup/presentation_matthias.pdf.

7. We are using the fact that $u \sim \mathcal{N}(0, \sigma^2)$ implies $\mathbb{E}[u^2] = \mathbb{V}[u] = \sigma^2$.

$\frac{h(\tilde{\mathbf{x}})}{\text{erf}(\tilde{\mathbf{x}}^\top \tilde{\mathbf{u}})}$	$\frac{C(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')}{\frac{2}{\pi} \arcsin(f_1(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}'))}$
$\mathbb{I}(\tilde{\mathbf{x}}^\top \tilde{\mathbf{u}} \geq 0)$	$\pi - \theta(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')$
$\text{ReLU}(\tilde{\mathbf{x}}^\top \tilde{\mathbf{u}})$	$\frac{f_2(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')}{\pi} \sin(\theta(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')) + \frac{\pi - \theta(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')}{\pi} \tilde{\mathbf{x}}^\top \tilde{\Sigma} \tilde{\mathbf{x}}'$

Table 18.4: Some neural net GP kernels. Here we define $f_1(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') = \frac{2\tilde{\mathbf{x}}^\top \tilde{\Sigma} \tilde{\mathbf{x}}'}{\sqrt{(1+2\tilde{\mathbf{x}}^\top \tilde{\Sigma} \tilde{\mathbf{x}})(1+(\tilde{\mathbf{x}}')^\top \tilde{\Sigma} \tilde{\mathbf{x}}')}}$, $f_2(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') = \|\tilde{\Sigma}^{\frac{1}{2}} \tilde{\mathbf{x}}\| \|\tilde{\Sigma}^{\frac{1}{2}} \tilde{\mathbf{x}}'\|$, $f_3(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') = \sqrt{(\tilde{\mathbf{x}}^\top \tilde{\Sigma} \tilde{\mathbf{x}})((\tilde{\mathbf{x}}')^\top \tilde{\Sigma} \tilde{\mathbf{x}}')}$, and $\theta(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') = \arccos(f_3(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}'))$. Results are derived in [Wil98; CS09].

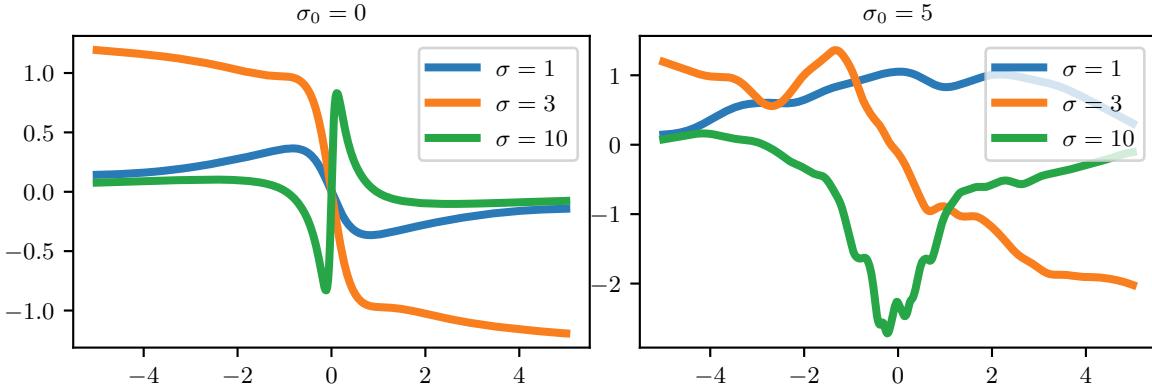


Figure 18.28: Sample output from a GP with an NNGP kernel derived from an infinitely wide one layer MLP with activation function of the form $h(x) = \text{erf}(x \cdot u + u_0)$ where $u \sim \mathcal{N}(0, \sigma)$ and $u_0 \sim \mathcal{N}(0, \sigma_0)$. Generated by [nngp_1d.ipynb](#). Used with kind permission of Matthias Bauer.

Thus the MLP converges to a GP. To compute the kernel function, we need to evaluate

$$C(\mathbf{x}, \mathbf{x}') = \mathbb{E}_{\mathbf{u}} [h(u_0 + \mathbf{u}^\top \mathbf{x}) h(u_0 + \mathbf{u}^\top \mathbf{x}')] = \mathbb{E}_{\mathbf{u}} [h(\tilde{\mathbf{u}}^\top \tilde{\mathbf{x}}) h(\tilde{\mathbf{u}}^\top \tilde{\mathbf{x}}')] \quad (18.181)$$

where we have defined $\tilde{\mathbf{x}} = (1, \mathbf{x})$ and $\tilde{\mathbf{u}} = (u_0, \mathbf{u})$. Let us define

$$\tilde{\Sigma} = \begin{pmatrix} \sigma_0^2 & 0 \\ 0 & \Sigma \end{pmatrix} \quad (18.182)$$

Then we have

$$C(\mathbf{x}, \mathbf{x}') = \int h(\tilde{\mathbf{u}}^\top \tilde{\mathbf{x}}) h(\tilde{\mathbf{u}}^\top \tilde{\mathbf{x}}') \mathcal{N}(\tilde{\mathbf{u}} | \mathbf{0}, \tilde{\Sigma}) d\tilde{\mathbf{u}} \quad (18.183)$$

This can be computed in closed form for certain activation functions, as shown in Table 18.4.

This is sometimes called the **neural net kernel**. Note that this is a non-stationary kernel, and sample paths from it are nearly discontinuous and tend to constant values for large positive or negative inputs, as illustrated in Figure 18.28.

18.7.2 Neural tangent kernel (NTK)

In Section 18.7.1 we derived the NN-GP kernel, under the assumption that all the weights are random. A natural question is: can we derive a kernel from a DNN after it has been trained, or more generally, while it is being trained. It turns out that this can be done, as we show below.

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:

$$\partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{f}_t) = -\eta \nabla_{\boldsymbol{\theta}} \mathbf{f}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) \quad (18.184)$$

Thus the function evolves over time as follows:

$$\partial_t \mathbf{f}_t = \nabla_{\boldsymbol{\theta}} \mathbf{f}_t^\top \partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \mathbf{f}_t^\top \nabla_{\boldsymbol{\theta}} \mathbf{f}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) = -\eta \mathcal{T}_t \cdot \nabla_f \mathcal{L}(\mathbf{f}_t) \quad (18.185)$$

where \mathcal{T}_t is the $N \times N$ kernel matrix

$$\mathcal{T}_t(\mathbf{x}, \mathbf{x}') \triangleq \nabla_{\boldsymbol{\theta}} f_t(\mathbf{x}) \cdot \nabla_{\boldsymbol{\theta}} f_t(\mathbf{x}') = \sum_{p=1}^P \frac{\partial f(\mathbf{x}; \boldsymbol{\theta})}{\partial \theta_p} \Big|_{\boldsymbol{\theta}_t} \frac{\partial f(\mathbf{x}'; \boldsymbol{\theta})}{\partial \theta_p} \Big|_{\boldsymbol{\theta}_t} \quad (18.186)$$

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]:

$$\mathcal{T}(\mathbf{x}, \mathbf{x}') \triangleq \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_\infty) \cdot \nabla_{\boldsymbol{\theta}} f(\mathbf{x}'; \boldsymbol{\theta}_\infty) \quad (18.187)$$

Details on how to compute this kernel for various models, such as CNNs, graph neural nets, and general neural nets, can be found in [Aro+19; Du+19; Yan19]. A software library to compute the NN-GP kernel and NTK is available in [Ano19].

The assumptions behind the NTK results in the parameters barely changing from their initial values (which is why a linear approximation around the starting parameters is valid). This can still lead to a change in the final predictions (and zero final training error), because the final layer weights can learn to use the random features just like in kernel regression. However, this phenomenon — which has been called “**lazy training**” [COB18] — is not representative of DNN behavior in practice [Woo+19], where parameters often change a lot. Fortunately it is possible to use a different parameterization which does result in feature learning in the infinite width limit [YH21].

18.7.3 Deep GPs

A **deep Gaussian process** or **DGP** is a composition of GPs [DL13]. More formally, a DGP of L layers is a hierarchical model of the form

$$\text{DGP}(\mathbf{x}) = f_L \circ \cdots \circ f_1(\mathbf{x}), \quad \mathbf{f}_i(\cdot) = [f_i^{(1)}(\cdot), \dots, f_i^{(H_i)}(\cdot)], \quad f_i^{(j)} \sim \text{GP}(0, \mathcal{K}_i(\cdot, \cdot)) \quad (18.188)$$

This is similar to a deep neural network, except the hidden nodes are now hidden functions.

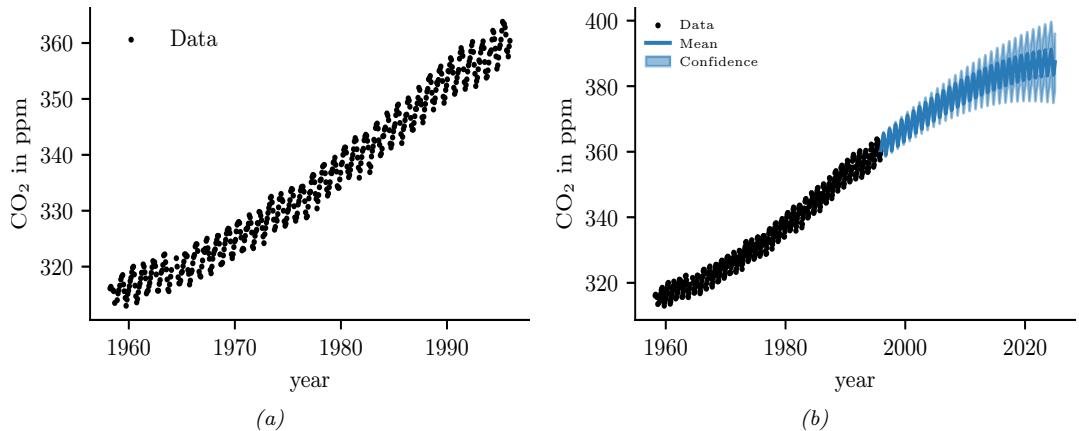


Figure 18.29: (a) The observed Mauna Loa CO₂ time series. (b) Forecasts from a GP. Generated by [gp_mauna_loa.ipynb](#).

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. It is tempting to think that deep kernel learning (Section 18.6.6) can solve this problem, but in theory a GP on top of a DNN is still just a GP. However, one can show that a composition of GPs is strictly more general. Unfortunately, inference in deep GPs is rather complicated, so we leave the details to [Supplementary](#) Section 18.1. See also [Jak21] for a recent survey on this topic.

18.8 Gaussian processes for time series forecasting

It is possible to use Gaussian processes to perform time series 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 Matérn 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 (see Section 8.2.3.4).

18.8.1 Example: Mauna Loa

In this section, we use the Mauna Loa CO₂ dataset from Section 29.12.5.1. We show the raw data in Figure 18.29(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:

$$\mathcal{K}(r) = \mathcal{K}_1(r) + \mathcal{K}_2(r) + \mathcal{K}_3(r) + \mathcal{K}_4(r) \quad (18.189)$$

where $\mathcal{K}_i(t, t') = \mathcal{K}_i(t - t')$ for the i 'th kernel.

To capture the long term smooth rising trend, we let \mathcal{K}_1 be a squared exponential (SE) kernel, where θ_0 is the amplitude and θ_1 is the length scale:

$$\mathcal{K}_1(r) = \theta_0^2 \exp\left(-\frac{r^2}{2\theta_1^2}\right) \quad (18.190)$$

To model the periodicity, we can use a periodic or exp-sine-squared kernel from Equation (18.18) 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 \mathcal{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.

$$\mathcal{K}_2(r) = \theta_2^2 \exp\left(-\frac{r^2}{2\theta_3^2} - \theta_5 \sin^2\left(\frac{\pi r}{\theta_4}\right)\right) \quad (18.191)$$

To model the (small) medium term irregularities, we use a rational quadratic kernel (Equation (18.20)):

$$\mathcal{K}_3(r) = \theta_6^2 \left[1 + \frac{r^2}{2\theta_7^2\theta_8}\right]^{-\theta_8} \quad (18.192)$$

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:

$$\mathcal{K}_4(r) = \theta_9^2 \exp\left(-\frac{r^2}{2\theta_{10}^2}\right) \quad (18.193)$$

where θ_9 is the magnitude of the correlated noise, and θ_{10} is the length scale. (Note that the combination of \mathcal{K}_1 and \mathcal{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 18.29(b).

19 Beyond the iid assumption

19.1 Introduction

The standard approach to supervised ML assumes the training and test sets both contain independent and identically distributed (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 19.2.

In some cases, we may have data from multiple related distributions, not just train and test, as we discuss in Section 19.6. We may also encounter data in a streaming setting, where the data distribution may be changing continuously, or in a piecewise constant fashion, as we discuss in Section 19.7. Finally, in Section 19.8, we discuss settings in which the test distribution is chosen by an adversary to minimize performance of a prediction system.

19.2 Distribution shift

Suppose we have a labeled training set from a **source distribution** $p(\mathbf{x}, \mathbf{y})$ which we use to fit a predictive model $p(\mathbf{y}|\mathbf{x})$. At test time we encounter data from the **target distribution** $q(\mathbf{x}, \mathbf{y})$. If $p \neq q$, we say that there has been a **distribution shift** or **dataset shift** [QC+08; BD+10]. This can adversely affect the performance of predictive models, as we illustrate in Section 19.2.1. In Section 19.2.2 we give a taxonomy of some kinds of distribution shift using the language of causal graphical models. We then proceed to discuss a variety of strategies that can be adopted to ameliorate the harm caused by distribution shift. In particular, in Section 19.3, we discuss techniques for detecting shifts, so that we can abstain from giving an incorrect prediction if the model is not confident. In Section 19.4, we discuss techniques to improve robustness to shifts; in particular, given labeled data from $p(\mathbf{x}, \mathbf{y})$, we aim to create a model that approximates $q(\mathbf{y}|\mathbf{x})$. In Section 19.5, we discuss techniques to adapt the model to the target distribution given some labeled or unlabeled data from the target.

19.2.1 Motivating examples

Figure 19.1 shows how shifting the test distribution slightly, by 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 corruptions**, such as image blurring [HD19]. Analogous problems can also occur in the text domain [Ryc+19], and the speech domain (see e.g., male vs female speakers in

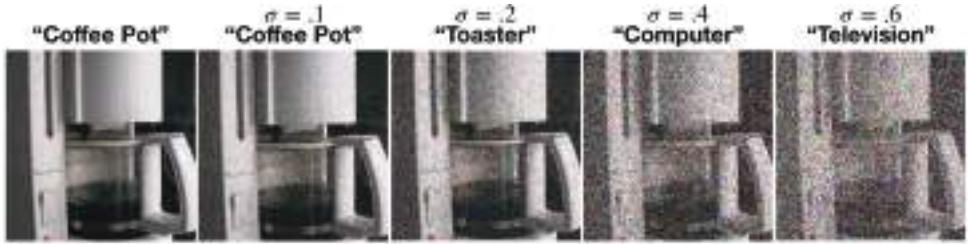
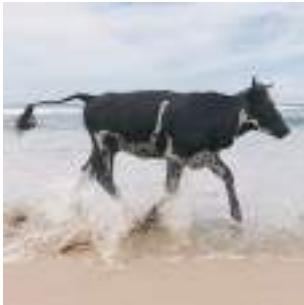


Figure 19.1: Effect of Gaussian noise of increasing magnitude on an image classifier. The model is a ResNet-50 CNN trained on ImageNet. From Figure 23 of [For+19]. Used with kind permission of Justin Gilmer.



(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:
0.94

Figure 19.2: 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 images, the cow occurs “out of context”, namely on a beach. The background is considered a “spurious correlation”. 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.

Figure 34.3). These examples illustrate that high performing predictive models can be very sensitive to small changes in the input distribution.

Performance can also drop on “clean” images, but which exhibit other kinds of shift. Figure 19.2 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** data, such as images with a different background, or taken at a different time or location (see e.g., [Koh+20b]) or from a novel viewing angle (see e.g., [KH22])).

The root cause of many of these problems is the fact that discriminative models often leverage features that are predictive of the output *in the training set*, but which are not reliable in general. For example, in an image classification dataset, we may find that green grass in the background is very predictive of the class label “cow”, but this is not a feature that is stable across different distributions; these are called **spurious correlations** or **shortcut features**. Unfortunately, such features are often easier for models to learn, for reasons explained in [Gei+20a; Xia+21b; Sha+20;

19.2. Distribution shift

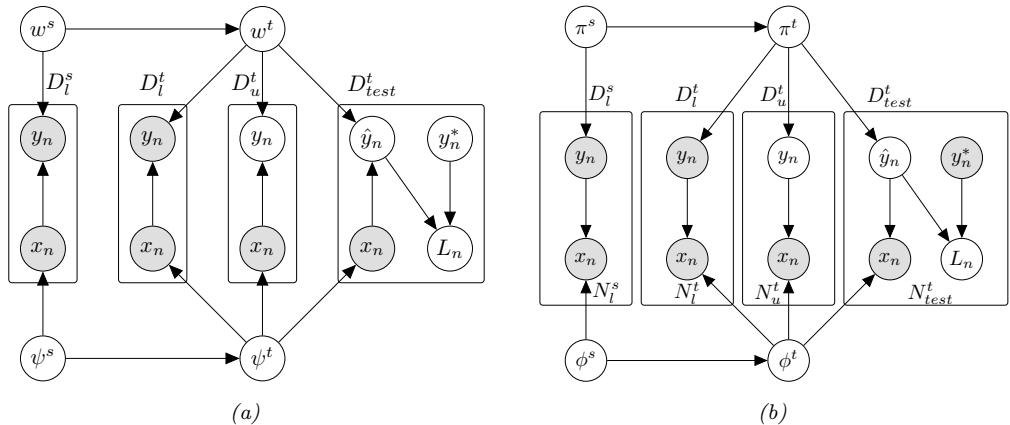


Figure 19.3: Models for distribution shift from source s to target t . Here \mathcal{D}_L^s is the labeled training set from the source, \mathcal{D}_L^t is an optional labeled training set from the target, \mathcal{D}_U^t is an optional unlabeled training set from the target, and \mathcal{D}_{test}^t is a labeled test set from the target. In the latter case, \hat{y}_n is the prediction on the n 'th test case (generated by the model), y_n^* is the true value, and $\ell_n = \ell(y_n^*, \hat{y}_n)$ is the corresponding loss. (Note that we don't evaluate the loss on the source distribution.) (a) Discriminative (causal) model. (b) Generative (anticausal).

Pez+21].

Relying on these shortcuts can have serious real-world consequences. For example, [Zec+18a] found that a CNN trained to recognize pneumonia was relying on hospital-specific metal tokens in the chest X-ray scans, rather than focusing on the lungs themselves, and thus the model did not generalize to new hospitals.

Analogous problems arise with other kinds of ML models, as well as other data types, 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 records [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 19.8).

19.2.2 A causal view of distribution shift

In the sections below, we briefly summarize some canonical kinds of distribution shift. We adopt a causal view of the problem, following [Sch+12a; Zha+13b; BP16; Mei18a; CWG20; Bud+21; SCS22]).¹ (See Section 4.7 for a brief discussion of causal DAGs, and Chapter 36 for more details.)

We assume the inputs to the model (the covariates) are X and the outputs to be predicted (the labels) are Y . If we believe that X causes Y , denoted $X \rightarrow Y$, we call it **causal prediction** or **discriminative prediction**. If we believe that Y causes X , denoted $Y \rightarrow X$, we call it **anticausal prediction** or **generative prediction**. [Sch+12a].

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].

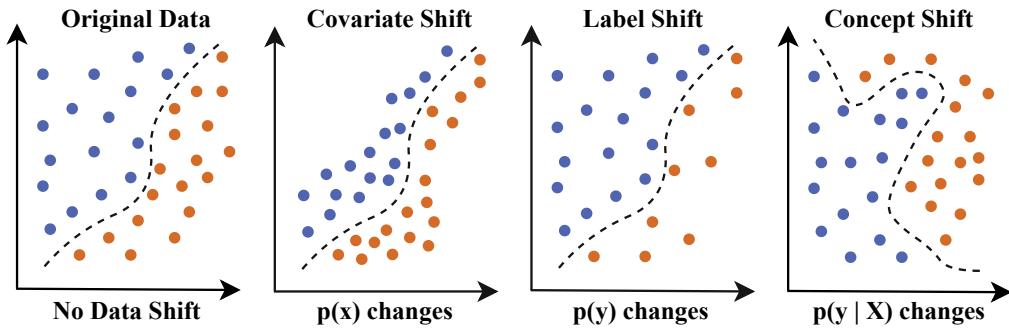


Figure 19.4: Illustration of the 4 main kinds of distribution shift for a 2d binary classification problem. Adapted from Figure 1 of [al21].

The decision about which model to use depends on our assumptions about the underlying **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 factor the joint distribution in two possible ways. One way is to define a discriminative model:

$$p_{\theta}(x, y) = p_{\psi}(x)p_w(y|x) \quad (19.1)$$

See Figure 19.3a. Alternatively we can define a generative model:

$$p_{\theta}(x, y) = p_{\pi}(y)p_{\phi}(x|y) \quad (19.2)$$

See Figure 19.3b. For each of these 2 models model types, different parts of the distribution may change from source to target. This gives rise to 4 canonical type of shift, as we discuss in Section 19.2.3.

19.2.3 The four main types of distribution shift

The four main types of distribution shift are summarized in Section 19.2 and are illustrated in Figure 19.4. We give more details below (see also [LP20]).

19.2.3.1 Covariate shift

In a causal (discriminative) model, if $p_{\psi}(x)$ changes (so $\psi^s \neq \psi^t$), we call it **covariate shift**, also called **domain shift**. For example, the training distribution may be clean images of coffee pots, and

19.2. Distribution shift

Name	Source	Target	Joint
Covariate/domain shift	$p(X)p(Y X)$	$q(X)p(Y X)$	Discriminative
Concept shift	$p(X)p(Y X)$	$p(X)q(Y X)$	Discriminative
Label (prior) shift	$p(Y)p(X Y)$	$q(Y)p(X Y)$	Generative
Manifestation shift	$p(Y)p(X Y)$	$p(Y)q(X Y)$	Generative

Table 19.1: The 4 main types of distribution shift.

the test distribution may be images of coffee pots with Gaussian noise, as shown in Figure 19.1; 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. 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.

Some standard strategies to combat covariate shift include importance weighting (Section 19.5.2) and domain adaptation (Section 19.5.3).

19.2.3.2 Concept shift

In a causal (discriminative) model, if $p_{\mathbf{w}}(\mathbf{y}|\mathbf{x})$ changes (so $\mathbf{w}^s \neq \mathbf{w}^t$), we call it **concept shift**, also called **annotation shift**. For example, consider the medical imaging context: the conventions for annotating images might be different between the training distribution and test distribution. 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 19.3.4.

Since concept shift is a change in what we “mean” by a label, it is impossible to fix this problem without seeing labeled examples from the target distribution, which defines each label by means of examples.

19.2.3.3 Label/prior shift

In a generative model, if $p_{\boldsymbol{\pi}}(\mathbf{y})$ changes (i.e., $\boldsymbol{\pi}^s \neq \boldsymbol{\pi}^t$), we call it **label shift**, also called **prior shift** or **prevalence shift**. 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.

Some standard strategies to combat label shift are to reweight the output of a discriminative classifier using an estimate of the new label distribution, as we discuss in Section 19.5.4.

19.2.3.4 Manifestation shift

In a generative model, if $p_{\phi}(\mathbf{x}|\mathbf{y})$ changes (i.e., $\phi^s \neq \phi^t$), we call **manifestation shift** [CWG20], or **conditional shift** [Zha+13b]. This is, in some sense, the inverse of concept 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. This is usually due to the presence of a hidden confounding factor that has changed between source and target (e.g., different age of the patients).

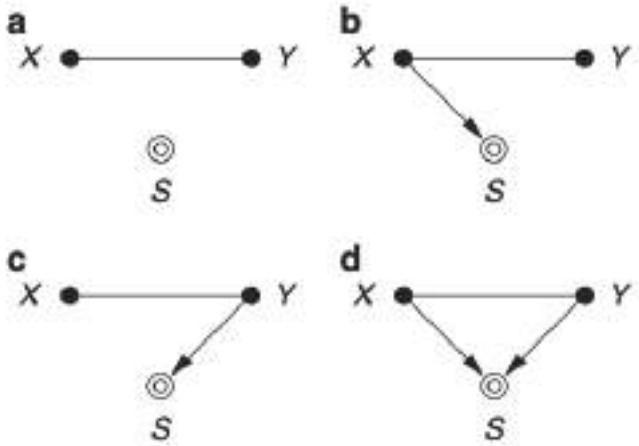


Figure 19.5: Causal diagrams for different sample selection strategies. Undirected edges can be oriented in either direction. The selection variable S is set to 1 if its parent nodes match the desired criterion; only these samples are included in the dataset. (a) No selection. (b) Selection on X . (c) Selection on Y . (d) Selection on X and Y . Adapted from Figure 4 of [CWG20].

19.2.4 Selection bias

In some cases, we may induce a shift in the distribution just due to the way the data is collected, without any changes to the underlying distributions. In particular, let $S = 1$ if a sample from the population is included in the training set, and $S = 0$ otherwise. Thus the source distribution is $p(X, Y) = p(X, Y|S = 1)$ but the target distribution is $q(X, Y) = p(X, Y|S \in \{0, 1\}) = p(X, Y)$, so there is no selection.

In Figure 19.5 we visualize the four kinds of selection. For example, suppose we select based on X meeting certain criteria, e.g., images of a certain quality, or exhibiting a certain pattern; this can induce domain shift or covariate shift. Now suppose we select based on Y meeting certain criteria, e.g., we are more likely to select rare examples where $Y = 1$, in order to **balance the dataset** (for reasons of computational efficiency); this can induce label shift. Finally, suppose we select based on both X and Y ; this can induce non-causal dependencies between X and Y , a phenomenon known as **selection bias** (see Section 4.2.4.2 for details).

19.3 Detecting distribution shifts

In general it will not be possible to make a model robust to all of the ways a distribution can shift at test time, nor will we always have access to test samples at training time. As an alternative, it may be sufficient for the model to *detect* that a shift has happened, and then to respond in the appropriate way. There are several ways of detecting distribution shift, some of which we summarize below. (See also Section 29.5.6, where we discuss changepoint detection in time series data.) The main distinction between methods is based on whether we have a set of samples from the target

distribution, or just a single sample, and whether the test samples are labeled or unlabeled. We discuss these different scenarios below.

19.3.1 Detecting shifts using two-sample testing

Suppose we collect a set of samples from the source and target distribution. We can then use standard techniques for **two-sample testing** to estimate if the null hypothesis, $p(\mathbf{x}, y) = q(\mathbf{x}, y)$, is true or not. (If we have unlabeled samples, we just test if $p(\mathbf{x}) = q(\mathbf{x})$.) For example, we can use MMD (Section 2.7.3) to measure the distance between the set of input samples (see e.g., [Liu+20a]). Or we can measure (Euclidean) distances in the embedding space of a classifier trained on the source (see e.g., [KM22]).

In some cases it may be possible to just test if the distribution of the labels $p(y)$ has changed, which is an easier problem than testing for changes in the distribution of inputs $p(\mathbf{x})$. In particular, if the label shift assumption (Section 19.2.3.3) holds (i.e., $q(\mathbf{x}|y) = p(\mathbf{x}|y)$), plus some other assumptions, then we can use the blackbox shift estimation technique from Section 19.5.4 to estimate $q(y)$. If we find that $q(y) = p(y)$, then we can conclude that $q(\mathbf{x}, y) = p(\mathbf{x}, y)$. In [RGL19], they showed experimentally that this method worked well for detecting distribution shifts even when the label shift assumption does not hold.

It is also possible to use conformal prediction (Section 14.3) to develop “distribution free” methods for detecting covariate shift, given only access to a calibration set and some conformity scoring function [HL20].

19.3.2 Detecting single out-of-distribution (OOD) inputs

Now suppose we just have *one* unlabeled sample from the target distribution, $\mathbf{x} \sim q$, and we want to know if \mathbf{x} is in-distribution (**ID**) or out-of-distribution (**OOD**). We will call this problem **out-of-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 19.6. 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].

19.3.2.1 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 from the OOD distribution, since the OOD set is basically the set of all possible inputs except for the ones of interest.

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.)

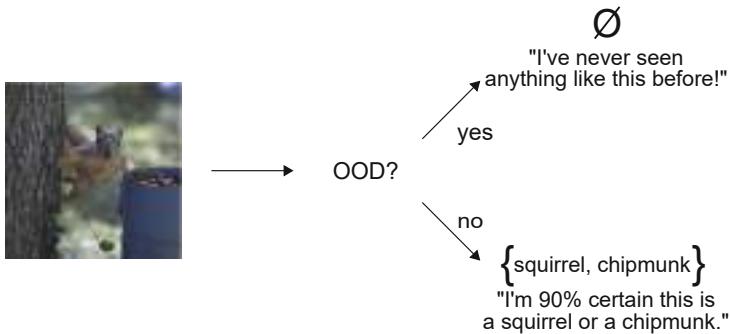


Figure 19.6: Illustration of a two-stage decision problem. First we must decide if the input image is out-of-distribution (OOD) or not. If it is not, we must return the set of class labels that have high probability. From [AB21]. Used with kind permission of Anastasios Angelopoulos.

19.3.2.2 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**” $s = \sum_c \ell_c$ [Liu+21b], etc. In [Mil+21; Vaz+22] they show that the simple max probability baseline performs very well in practice.

19.3.2.3 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 19.6. We define the prediction set as follows:

$$\mathcal{T}_\lambda(\mathbf{x}) = \begin{cases} \emptyset & \text{if } \text{OOD}(\mathbf{x}) > \lambda_1 \\ \text{APS}(\mathbf{x}) & \text{otherwise} \end{cases} \quad (19.3)$$

where $\text{OOD}(\mathbf{x})$ is some heuristic OOD score (such as max class probability), 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

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.

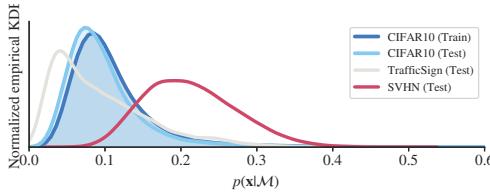


Figure 19.7: Likelihoods from a Glow normalizing flow model (Section 23.2.1) trained on CIFAR10 and evaluated on different test sets. The SVHN street sign dataset has lower visual complexity, and hence higher likelihood. Qualitatively similar results are obtained for other generative models and datasets. From Figure 1 of [Ser+20]. Used with kind permission of Joan Serrà.

method (see [Ang+21]). The resulting thresholds will jointly minimize the following risks:

$$R_1(\boldsymbol{\lambda}) = p(\mathcal{T}_{\boldsymbol{\lambda}}(\mathbf{x}) = \emptyset) \quad (19.4)$$

$$R_2(\boldsymbol{\lambda}) = p(\mathbf{y} \notin \mathcal{T}_{\boldsymbol{\lambda}}(\mathbf{x}) | \mathcal{T}_{\boldsymbol{\lambda}}(\mathbf{x}) \neq \emptyset) \quad (19.5)$$

where $p(\mathbf{x}, y)$ is the true but unknown source 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.

19.3.2.4 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) to the ID samples, and then to evaluate the likelihood $p(\mathbf{x})$ and compare this to some threshold value. Unfortunately for many kinds of deep model and datasets, we sometimes find that $p(\mathbf{x})$ is lower for samples that are from the source distribution than from a novel target distribution. For example, if we train a pixel-CNN model (Section 22.3.2) or a normalizing-flow model (Chapter 23) 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 19.7).

One solution to this is to use $\log R(\mathbf{x})$ 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 $p(\mathbf{x}') = p(\mathbf{x}) |\det \text{Jac}(\phi^{-1})(\mathbf{x})|$. Thus $L(\mathbf{x}')$ will 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

$$R(\mathbf{x}') = \log p(\mathbf{x}') - \log q(\mathbf{x}') = \log p(\mathbf{x}) + \log |\det \text{Jac}(\phi^{-1})(\mathbf{x})| - \log q(\mathbf{x}) - \log |\det \text{Jac}(\phi^{-1})(\mathbf{x})| \quad (19.6)$$

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 instead of inputs [Mor+21a], etc.

A closely related technique relies on **reconstruction error**. The idea is to fit an autoencoder or VAE (Section 21.2) 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 26) 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].

19.3.3 Selective prediction

Suppose the system has a confidence level of p that an input is OOD (see Section 19.3.4 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** (see e.g., [EW10; GEY19; Ziy+19; JKG18]), and is useful for applications where an error can be more costly than asking a human expert for help (e.g., medical image classification).

19.3.3.1 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.

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 stochastic gradient Langevin dynamics (see Section 12.7.1), which is a form of MCMC inference. We use 6,000 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 19.8: the green curve is the fraction of the test set for which we make a prediction, and the blue curve is the accuracy. On the left we show SGD, and on the right we show SGLD. In this case, performance is quite similar, although SGD has slightly higher accuracy. However, the story changes somewhat when there is distribution shift.

To study the effects under distribution shift, we apply both models to FashionMNIST data. We show the results in Figure 19.9. The accuracy of both models is very low (less than the chance level of 10%), but SGD remains quite confident in many more of its predictions than SGLD, which is more conservative. To see this, consider a confidence threshold of 0.5: the SGD approach predicts on about 97% of the examples (recall that the green curve corresponds to the right hand axis), whereas the SGLD only predicts on about 70% of the examples.

More details on the behavior of Bayesian neural networks under distribution shift can be found in Section 17.4.6.2.

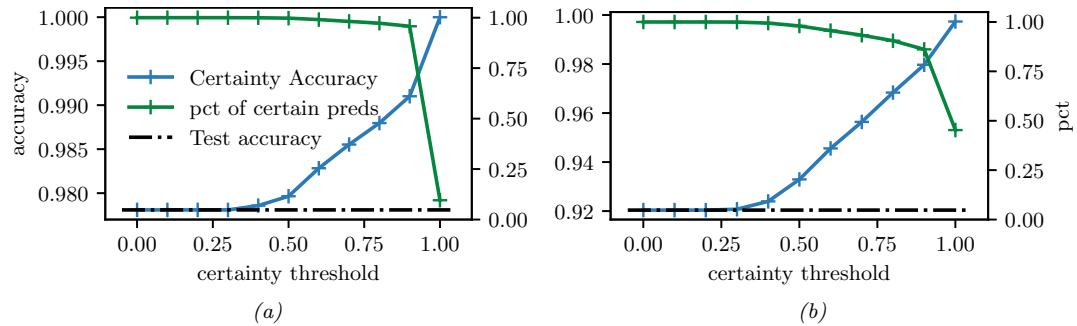


Figure 19.8: Accuracy vs confidence plots for an MLP fit to the MNIST training set, and then evaluated on one batch from the MNIST test set. Scale for blue accuracy curve is on the left, scale for green percentage predicted curve is on the right. (a) Plugin approach, computed using SGD. (b) Bayesian approach, computed using 10 samples from SGLD. Generated by [bnn_mnist_sgld.ipynb](#).

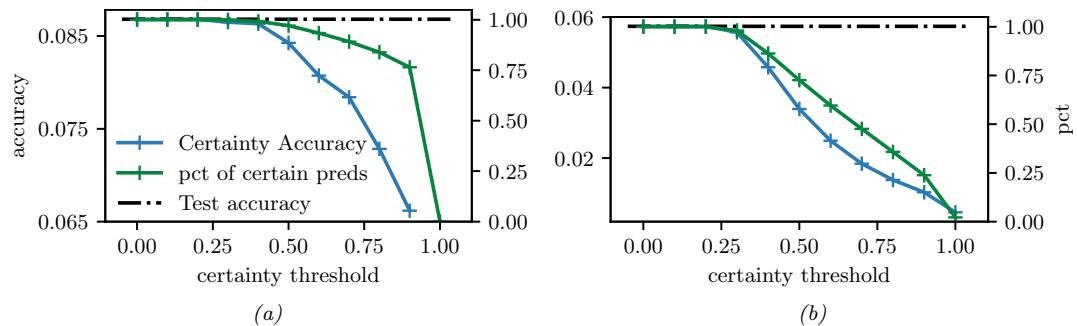


Figure 19.9: Similar to Figure 19.8, except that performance is evaluated on the Fashion MNIST dataset. (a) SGD. (b) SGLD. Generated by [bnn_mnist_sgld.ipynb](#).

19.3.4 Open set and open world recognition

In Section 19.3.3, we discussed methods that “refuse to classify” if the system is not confident enough about its predicted output. If the system detects that this lack of confidence is due to the input coming from a novel class, rather than just being a novel instance of an existing class, we call the problem **open set recognition** (see e.g., [GHC20] for a review).

Rather than “flagging” novel classes as OOD, we can instead allow the set of classes to grow over time; this is called **open world classification** [BB15a]. Note that open world classification is most naturally tackled in the context of a continual learning system, which we discuss in Section 19.7.3.

For a survey article that connects open set learning with OOD detection, see [Sal+22].

19.4 Robustness to distribution shifts

In this section, we discuss techniques to improve the **robustness** of a model to distribution shifts. In particular, given labeled data from $p(\mathbf{x}, \mathbf{y})$, we aim to create a model that approximates $q(\mathbf{y}|\mathbf{x})$.

19.4.1 Data augmentation

A simple approach to potentially increasing the robustness of a predictive model to distribution shifts is to simulate samples from the target distribution by modifying the source data. This is called **data augmentation**, and is widely used in the deep learning community. For example, it is standard to apply small perturbations to images (e.g., shifting them or rotating them), while keeping the label the same (assuming that the label should be invariant to such changes); see e.g., [SK19; Hen+20] for details. Similarly, in NLP (natural language processing), it is standard to change words that should not affect the label (e.g., replacing “he” with “she” in a sentiment analysis system), or to use **back translation** (from a source language to a target language and back) to generate paraphrases; see e.g., [Fen+21] for a review of such techniques. For a causal perspective on data augmentation, see e.g., [Kau+21].

19.4.2 Distributionally robust optimization

We can make a discriminative model that is robust to (some forms of) covariate shift by solving the following **distributionally robust optimization** (DRO) problem:

$$\min_{f \in \mathcal{F}} \max_{\mathbf{w} \in \mathcal{W}} \frac{1}{N} \sum_{n=1}^N w_n \ell(f(\mathbf{x}_n), \mathbf{y}_n) \quad (19.7)$$

where the samples are from the source distribution, $(\mathbf{x}_n, \mathbf{y}_n) \sim p$. This is an example of a **min-max optimization problem**, in which we want to minimize the worst case risk. The specification of the robustness set, \mathcal{W} , is a key factor that determines how well the method works, and how difficult the optimization problem is. Typically it is specified in terms of an ℓ_2 ball around the inputs, but this could also be defined in a feature (embedding space). It is also possible to define the robustness set in terms of local changes to a structural causal model [Mei18a]. For more details on DRO, see e.g., [CP20a; LFG21; Sag+20; RM22].

19.5 Adapting to distribution shifts

In this section, we discuss techniques to **adapt** the model to the target distribution. If we have some labeled data from the target distribution, we can use transfer learning, as we discuss in Section 19.5.1. However, getting labeled data from the target distribution is often not an option. Therefore, in the other sections, we discuss techniques that just rely on *unlabeled* data from the target distribution.

19.5.1 Supervised adaptation using transfer learning

Suppose we have labeled training data from a source distribution, $\mathcal{D}^s = \{(\mathbf{x}_n, \mathbf{y}_n) \sim p : n = 1 : N_s\}$, and also some labeled data from the target distribution, $\mathcal{D}^t = \{(\mathbf{x}_n, \mathbf{y}_n) \sim q : n = 1 : N_t\}$. Our goal is to minimize the risk on the target distribution q , which can be computed using

$$R(f, q) = \mathbb{E}_{q(\mathbf{x}, \mathbf{y})} [\ell(\mathbf{y}, f(\mathbf{x}))] \quad (19.8)$$

We can approximate the risk empirically using

$$\hat{R}(f, \mathcal{D}^t) = \frac{1}{|\mathcal{D}^t|} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{D}^t} \ell(\mathbf{y}_n, f(\mathbf{x}_n)) \quad (19.9)$$

If \mathcal{D}^t is large enough, we can directly optimize this using standard empirical risk minimization (ERM). However, if \mathcal{D}^t is small, we might want to use \mathcal{D}^s somehow as a regularizer. This is called **transfer learning**, since we hope to “transfer knowledge” from p to q . There are many approaches to transfer learning (see e.g., [Zhu+21] for a review). We briefly mention a few below.

19.5.1.1 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 $f^s = \operatorname{argmin}_f \hat{R}(f, \mathcal{D}^s)$. (Note that the source data may be unlabeled, in which case we can use self-supervised learning methods.) We then adapt the model to work on the target distribution by computing

$$f^t = \operatorname{argmin}_f \hat{R}(f, \mathcal{D}^t) + \lambda \|f - f^s\| \quad (19.10)$$

where $\|f - f^s\|$ is some distance between the functions, and $\lambda \geq 0$ controls the degree of regularization.

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.) We can then solve Equation (19.10) by “chopping off the head” from f^s and replacing it with a new linear layer, to map to the new set of labels for the target distribution, and then compute a new MAP estimate for the parameters on the target distribution. (We can also compute a prior for the parameters of the source model, and use it to compute a posterior for the parameters of the target model, as discussed in Section 17.2.3.)

This approach is very widely used in practice, since it is simple and effective. In particular, it is common to take a large pre-trained model, such as a transformer, that has been trained (often using self supervised learning, Section 32.3.3) on a lot of data, such as the entire web, and then to use this model as a feature extractor (see e.g., [Kol+20]). The features are fed to the downstream model, which may be a linear classifier or a shallow MLP, which is trained on the target distribution.

19.5.1.2 Prompt tuning (in-context learning)

Recently another approach to transfer learning has been developed, that leverages large models, such as transformers (Section 22.4), 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 are not changed; instead, the model is simply “conditioned” on new training data, usually in the form of a text **prompt** \mathbf{z} . That is, we compute

$$f^t(\mathbf{x}) = f^s(\mathbf{x} \cup \mathbf{z}) \quad (19.11)$$

where we (manually or automatically) optimize \mathbf{z} while keeping f^s frozen. This approach is called **prompt tuning** or **in-context learning** (see e.g., [Liu+21a]), and is an instance of **few-shot learning** (see Figure 22.4 for an example).

Here \mathbf{z} acts like a small training dataset, and f^s uses attention (Section 16.2.7) to “look at” all its inputs, comparing \mathbf{x} with the examples in \mathbf{z} , and uses this to make a prediction. This works because the text training data often has a similar hierarchical structure (see [Xie+22] for a Bayesian interpretation).

19.5.2 Weighted ERM for covariate shift

In this section we reconsider the risk minimization objective in Equation (19.8), but leverage unlabeled data from the target distribution to estimate it. If we make the covariate shift assumption (i.e., $q(\mathbf{x}, \mathbf{y}) = q(\mathbf{x})p(\mathbf{y}|\mathbf{x})$), then we have

$$R(f, q) = \int q(\mathbf{x})q(\mathbf{y}|\mathbf{x})\ell(\mathbf{y}, f(\mathbf{x}))d\mathbf{x}d\mathbf{y} \quad (19.12)$$

$$= \int q(\mathbf{x})p(\mathbf{y}|\mathbf{x})\ell(\mathbf{y}, f(\mathbf{x}))d\mathbf{x}d\mathbf{y} \quad (19.13)$$

$$= \int \frac{q(\mathbf{x})}{p(\mathbf{x})} p(\mathbf{x})p(\mathbf{y}|\mathbf{x})\ell(\mathbf{y}, f(\mathbf{x}))d\mathbf{x}d\mathbf{y} \quad (19.14)$$

$$\approx \frac{1}{N} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{D}_L^s} w_n \ell(\mathbf{y}_n, f(\mathbf{x}_n)) \quad (19.15)$$

where the weights are given by the ratio

$$w_n = w(\mathbf{x}_n) = \frac{q(\mathbf{x}_n)}{p(\mathbf{x}_n)} \quad (19.16)$$

Thus we can solve the covariate shift problem by using **weighted ERM** [Shi00a; SKM07].

However, this raises two questions. First, why do we need to use this technique, since a discriminative model $p(\mathbf{y}|\mathbf{x})$ should work for any input \mathbf{x} , regardless of which distribution it comes from? Second, given that we do need to use this method, in practice how should we estimate the weights $w_n = w(\mathbf{x}_n) = \frac{q(\mathbf{x}_n)}{p(\mathbf{x}_n)}$? We discuss these issues below.

19.5.2.1 Why is covariate shift a problem for discriminative models?

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. This is illustrated in Figure 19.10b, where we show that a linear model fit to the source distribution may perform much worse on the target distribution than a model that weights target points more heavily during training.

19.5.2.2 How should we estimating the ERM weights?

One approach to estimating the ERM weights $w_n = w(\mathbf{x}_n) = \frac{q(\mathbf{x}_n)}{p(\mathbf{x}_n)}$ is to learn a density model for the source and target. However, density estimation is difficult for high dimensional features. An alternative approach is to try to approximate the density ratio, by fitting a binary classifier to distinguish the two distributions, as discussed in Section 2.7.5. In particular, suppose we have an equal number of samples from $p(\mathbf{x})$ and $q(\mathbf{x})$. Let us label the first set with $c = -1$ and the second set with $c = 1$. Then we have

$$p(c = 1|\mathbf{x}) = \frac{q(\mathbf{x})}{q(\mathbf{x}) + p(\mathbf{x})} \quad (19.17)$$

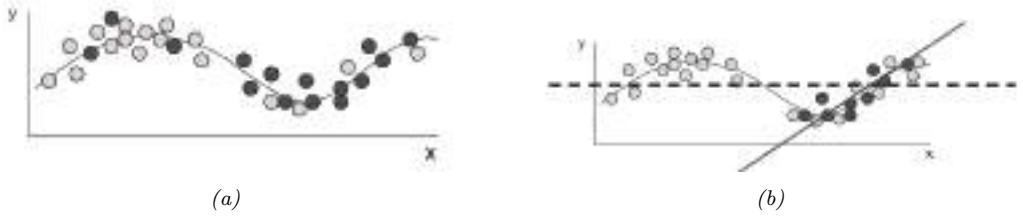


Figure 19.10: (a) Illustration of covariate shift. Light gray represents training distribution, dark gray represents test distribution. We see the test distribution has shifted to the right but the underlying input-output function is constant. (b) Dashed line: fitting a linear model across the full support 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.

and hence $\frac{p(c=1|\mathbf{x})}{p(c=-1|\mathbf{x})} = \frac{q(\mathbf{x})}{p(\mathbf{x})}$. If the classifier has the form $f(\mathbf{x}) = p(c=1|\mathbf{x}) = \sigma(h(\mathbf{x})) = \frac{1}{1+\exp(-h(\mathbf{x}))}$, where $h(\mathbf{x})$ is the prediction function that returns the logits, then the importance weights are given by

$$w_n = \frac{1/(1 + \exp(-h(\mathbf{x}_n)))}{\exp(-h(\mathbf{x}_n))/(1 + \exp(-h(\mathbf{x}_n)))} = \exp(h(\mathbf{x}_n)) \quad (19.18)$$

Of course this method requires that \mathbf{x} values that may occur in the test distribution should also be possible in the training distribution, i.e., $q(\mathbf{x}) > 0 \implies p(\mathbf{x}) > 0$. Hence there are no guarantees about this method being able to interpolate beyond the training distribution.

19.5.3 Unsupervised domain adaptation for covariate shift

We now turn to methods that only need access to unlabeled examples from the target distribution.

The technique of **unsupervised domain adaptation** or **UDA** assumes access to a labeled dataset from the source distribution, $\mathcal{D}_1 = \mathcal{D}_L^s \sim p(\mathbf{x}, \mathbf{y})$ and an unlabeled dataset from the target distribution, $\mathcal{D}_2 = \mathcal{D}_U^t \sim q(\mathbf{x})$. It then uses the unlabeled target data to improve robustness or invariance of the predictor, rather than using a weighted ERM method.

There are many forms of UDA (see e.g., [KL21; CB20] for reviews). Here we just focus on one method, called **domain adversarial learning** [Gan+16a]. Let $f_\alpha : \mathcal{X}_1 \cup \mathcal{X}_2 \rightarrow \mathcal{H}$ be a feature extractor defined on the two input domains, let $c_\beta : \mathcal{H} \rightarrow \{1, 2\}$ be a classifier that maps from the feature space to the domain from which the input was taken, either domain 1 or 2 (source or target), and let $g_\gamma : \mathcal{H} \rightarrow \mathcal{Y}$ be a classifier that maps from the feature space to the label space. We want to train the feature extractor so that it cannot distinguish whether the input is coming from the source or target distribution; in this case, it will only be able to use features that are common to both domains. Hence we optimize

$$\min_{\gamma} \max_{\alpha, \beta} \frac{1}{N_1 + N_2} \sum_{\mathbf{x}_n \in \mathcal{D}_1, \mathcal{D}_2} \ell(d_n, c_\beta(f_\alpha(\mathbf{x}_n))) + \frac{1}{N_1} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{D}_1} \ell(\mathbf{y}_n, g_\gamma(f_\alpha(\mathbf{x}_n))) \quad (19.19)$$

The objective in Equation (19.19) minimizes the loss on the desired task of classifying y , but *maximizes*

the loss on the auxiliary task of classifying the domain label d . This can be implemented by the **gradient sign reversal** trick, and is related to GANs (Section 26.7.6).

19.5.4 Unsupervised techniques for label shift

In this section, we describe an approach known as **blackbox shift estimation**, due to [LWS18], which can be used to tackle the **label shift** problem in an unsupervised way. We assume that the only thing that changes in the target distribution is the label prior, i.e., if the source distribution is denoted by $p(\mathbf{x}, \mathbf{y})$ and target distribution is denoted by $q(\mathbf{x}, \mathbf{y})$, we assume $q(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})q(\mathbf{y})$.

First note that, for any deterministic function $f : \mathcal{X} \rightarrow \mathcal{Y}$, we have

$$p(\mathbf{x}|y) = q(\mathbf{x}|y) \implies p(f(\mathbf{x})|y) = q(f(\mathbf{x})|y) \implies p(\hat{y}|y) = q(\hat{y}|y) \quad (19.20)$$

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 predicts class i on the test set, and let $q(y = i)$ be the true but unknown label distribution on the test set, and let $C_{ij} = p(\hat{y} = i|y = j)$ be the class confusion matrix estimated on the training set. Then we have

$$\mu_{\hat{y}} = \sum_y q(\hat{y}|y)q(y) = \sum_y p(\hat{y}|y)q(y) = \sum_y p(\hat{y}, y) \frac{q(y)}{p(y)} \quad (19.21)$$

We can write this in matrix-vector form as follows:

$$\mu_i = \sum_i C_{ij} q_j, \implies \boldsymbol{\mu} = \mathbf{C}\mathbf{q} \quad (19.22)$$

Hence we can solve $\mathbf{q} = \mathbf{C}^{-1}\boldsymbol{\mu}$, providing that \mathbf{C} is not singular (this will be the case if \mathbf{C} is strongly diagonal, i.e., the model predicts class y_i correctly more often than any other class y_j). We also require that for every $q(y) > 0$ we have $p(y) > 0$, which means we see every label at training time.

Once we know the new label distribution, $q(\mathbf{y})$, we can adjust our discriminative classifier to take the new label prior into account as follows:

$$q(y|\mathbf{x}) = \frac{q(\mathbf{x}|y)q(y)}{q(\mathbf{x})} = \frac{p(\mathbf{x}|y)q(y)}{q(\mathbf{x})} = \frac{p(y|\mathbf{x})p(\mathbf{x})}{p(y)} \frac{q(y)}{q(\mathbf{x})} = p(y|\mathbf{x}) \frac{q(y)}{p(y)} \frac{p(\mathbf{x})}{q(\mathbf{x})} \quad (19.23)$$

We can safely ignore the $\frac{p(\mathbf{x})}{q(\mathbf{x})}$ term, which is constant wrt y , and we can plug in our estimates of the label distributions to compute the $\frac{q(y)}{p(y)}$.

In summary, there are three requirements for this method: (1) the confusion matrix is invertible; (2) no new labels at test time; (3) the only thing that changes is the label prior. If these three conditions hold, the above approach is a valid estimator. See [LWS18] for more details, and [Gar+20] for an alternative approach, based on maximum likelihood (rather than moment matching) for estimating the new marginal label distribution.

19.5.5 Test-time 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. This is called **test time adaptation** or **TTA**. The

difference from the unsupervised domain adaptation methods of Section 19.5.3 is that, in the online setting, we just have the model which was trained on the source, and not the source distribution.

In [Sun+20] they proposed an approach called **TTT** (“test-time training”) for adapting a discriminative model. In this approach, a self-supervised proxy task is used to create proxy-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} .

Of course, if the proxy output, such as the rotation angle, is not known, we cannot use proxy-supervised learning methods such as TTT. In [Wan+20a], they propose an approach, inspired by semi-supervised learning methods, which they call **TENT**, which stands for “test-time adaptation by entropy minimization”. The idea is to update the classifier parameters to minimize the entropy of the predictive distribution on a batch of test examples. In [Goy+22], they give a justification for this heuristic from the meta-learning perspective. In [ZL21], they present a Bayesian version of TENT, which they call **BACS**, which stands for “Bayesian adaptation under covariate shift”. 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 \mathbf{x} , to create a set of inputs, $\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_B$. Now we update the parameters so as to minimize the predictive entropy produced by the averaged distribution

$$\bar{p}(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \frac{1}{B} \sum_{b=1}^B p(\mathbf{y}|\tilde{\mathbf{x}}_b, \mathbf{w}) \quad (19.24)$$

This ensures that the model gives the same predictions for each perturbation of the input, and that the predictions are confident (low entropy).

An alternative to entropy based methods is to use **pseudolabels** (predicted outputs on the unlabeled target generated by the source model), and then to **self-train** on these (see e.g., [KML20; LHF20; Che+22]), often with additional regularizers to prevent over-fitting.

19.6 Learning from multiple distributions

In Section 19.2, we discussed the setting in which a model is trained on a single 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 $J \geq 2$ source distributions, before being tested on data from a target distribution. This includes a variety of different problem settings, depending on the value of J , as we summarize in Figure 19.11.

19.6.1 Multitask learning

In **multi-task learning** (MTL) [Car97], we have labeled data from J different distributions, $\mathcal{D}^j = \{(\mathbf{x}_n^j, \mathbf{y}_n^j) : n = 1 : N_j\}$, and the goal is to learn a model that predicts well on all J of them simultaneously, where $f(\mathbf{x}, j) : \mathcal{X} \rightarrow \mathcal{Y}_j$ is the output for the j 'th task. For example, we might want to map a color image of size $H \times W \times 3$ to a set of semantic labels per pixel, $\mathcal{Y}^1 = \{1, \dots, C\}^{HW}$, as

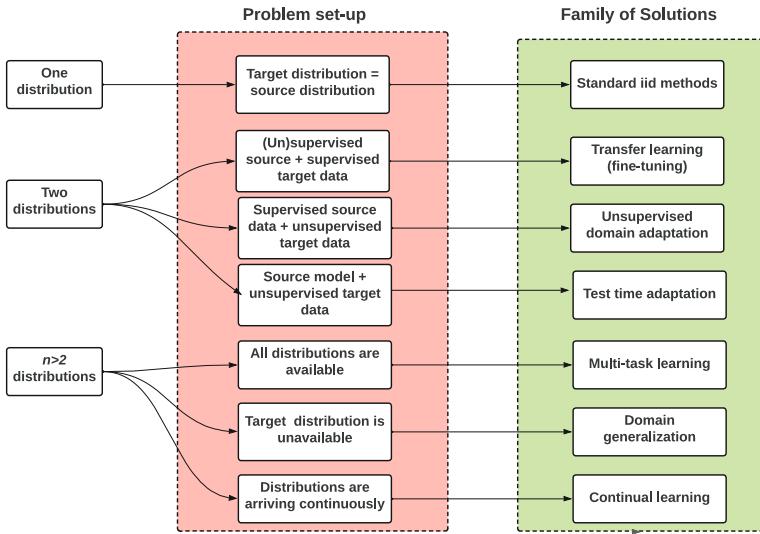


Figure 19.11: Schematic overview of techniques for learning from 1 or more different distributions. Adapted from slide 3 of [Sca21].

well as a set of predicted depth values per pixel, $\mathcal{Y}^2 = \mathbb{R}^{HW}$. We can do this using ERM where we have multiple samples for each task:

$$f^* = \underset{f}{\operatorname{argmin}} \sum_{j=1}^J \sum_{n=1}^{N_j} \ell_j(\mathbf{y}_n^j, f(\mathbf{x}_n^j, j)) \quad (19.25)$$

where ℓ_j is the loss function for task j (suitably scaled).

There are many approaches to solving MTL. The simplest is to fit a single model with multiple “output heads”, as illustrated in Figure 19.12. This is called a “**shared trunk network**”. Unfortunately this often leads to worse performance than training J single task networks. In [Mis+16], they propose to take a weighted combination of the activations of each single task network, an approach they called “**cross-stitch networks**”. See [ZY21] for a more detailed review of neural approaches, and [BLS11] for a theoretical analysis of this problem.

Note that multi-task learning does not always help performance on each task because sometimes there can be “**task interference**” or “**negative transfer**” (see e.g., [MAP17; Sta+20; WZR20]). In such cases, we should use separate networks, rather than using one model with multiple output heads.

19.6.2 Domain generalization

The problem of **domain generalization** assumes we train on J different labeled source distributions or “**environments**” (also called “**domains**”), and then test on a new target distribution (denoted by

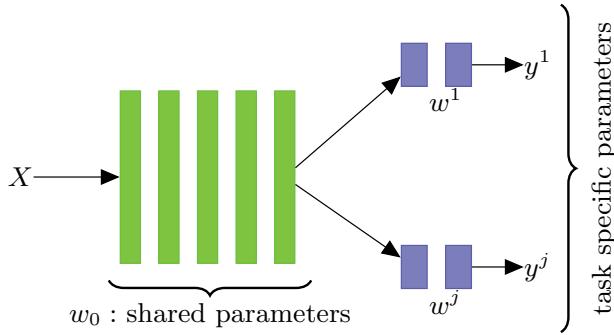


Figure 19.12: Illustration of multi-headed network for multi-task learning.

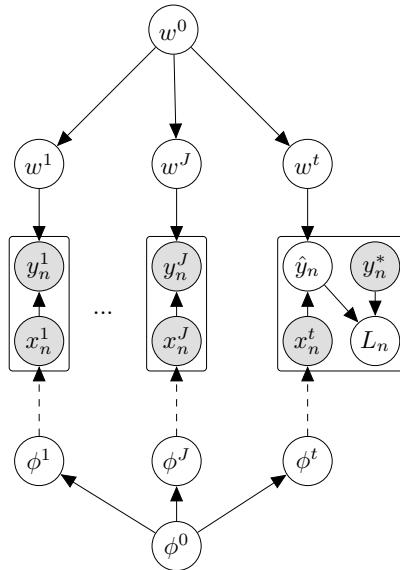


Figure 19.13: Hierarchical Bayesian discriminative model for learning from J different environments (distributions), and then testing on a new target distribution $t = J + 1$. Here \hat{y}_n is the prediction for test example x_n , y_n^* is the true output, and $\ell_n = \ell(\mathbf{y}_n^t, \mathbf{y}_n^*)$ is the associated loss. The parameters of the distribution over input features $p_\phi(x)$ are shown with dotted edges, since these distributions do not need to be learned in a discriminative model.

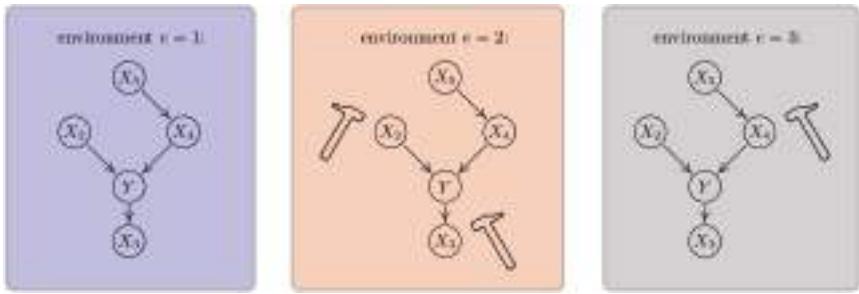


Figure 19.14: Illustration of invariant causal prediction. The hammer symbol represents variables whose distribution is perturbed in the given environment. An invariant predictor must use features $\{X_2, X_4\}$. Considering indirect causes instead of direct ones (e.g. $\{X_2, X_3\}$) or an incomplete 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.

$t = J + 1$). In some cases each environment is just identified with a meaningless integer id. In more realistic settings, each different distribution has associated **meta-data** or **context variables** that characterizes the environment in which the data was collected, such as the time, location, imaging device, etc.

Domain generalization (DG) is similar to multi-task learning, but differs in what we want to predict. In particular, in DG, we only care about prediction accuracy on the target distribution, not the J training distribution. Furthermore, we assume we don't have any labeled data from the target distribution. We therefore have to make some assumptions about how $p^t(\mathbf{x}, \mathbf{y})$ relates to $p^j(\mathbf{x}, \mathbf{y})$ for $j = 1 : J$.

One way to formalize this is to create a hierarchical Bayesian model, as proposed in [Bax00], and illustrated in Figure 19.13. This encodes the assumption that $p^t(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\phi^t)p(\mathbf{y}|\mathbf{x}, \mathbf{w}^t)$ where \mathbf{w}^t is derived from a common “population level” model \mathbf{w}^0 , shared across all distributions, and similarly for ϕ^t . (Note, however, that in a discriminative model, we don't need to model $p(\mathbf{x}|\phi^t)$.) See Section 15.5 for discussion of hierarchical Bayesian GLMs, and Section 17.6 for discussion of hierarchical Bayesian MLPs.

Many other techniques have been proposed for DG. Note, however, that [GLP21] found that none of these methods worked consistently better than the baseline approach of performing empirical risk minimization across all the provided datasets. For more information, see e.g., [GLP21; She+21; Wan+21; Chr+21].

19.6.3 Invariant risk minimization

One approach to domain generalization that has received a lot of attention is called **invariant risk minimization** or **IRM** [Arj+19]. The goal is to learn a predictor that works well across all environments, yet is less prone to depending on the kinds of “spurious features” we discussed in Section 19.2.1.

IRM is an extension of an earlier method called **invariant causal prediction** (ICP) [PBM16b]. This uses hypothesis testing methods to find the set of predictors (features) that directly cause the

outcome in each environment, rather than features that are indirect causes, or are just correlated with the outcome. See Figure 19.14 for an illustration.

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 requires 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

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{j=1}^J \frac{1}{N_j} \sum_{n=1}^{N_j} \ell(\mathbf{y}_n^j, f(\mathbf{x}_n^j)) \quad (19.26)$$

$$\text{such that } f \in \arg \min_{g \in \mathcal{F}} \frac{1}{N_j} \sum_{n=1}^{N_j} \ell(\mathbf{y}_n^j, g(\mathbf{x}_n^j)) \text{ for all } j \in \mathcal{E} \quad (19.27)$$

where \mathcal{E} is the set of environments, and \mathcal{F} is the set of prediction functions. 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 (generative) models [Ahu+21].

19.6.4 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. This is very similar to domain generalization (Section 19.6.2), except that we partition each training distribution into training and test, so we can “practice” learning to generalize from a training set to a test set. A general review of meta-learning can be found in [Hos+20a]. Here we present a unifying summary based on the hierarchical Bayesian framework proposed in [Gor+19].

19.6.4.1 Meta-learning as probabilistic inference for prediction

We assume there are J tasks (distributions), each of which has a training set $\mathcal{D}_{\text{train}}^j = \{(\mathbf{x}_n^j, \mathbf{y}_n^j) : n = 1 : N^j\}$ and a test set $\mathcal{D}_{\text{test}}^j = \{(\tilde{\mathbf{x}}_m^j, \tilde{\mathbf{y}}_m^j) : m = 1 : M^j\}$. In addition, \mathbf{w}^j are the task specific parameters, and \mathbf{w}^0 are the shared parameters, as shown in Figure 19.15. This is very similar to the domain generalization model in Figure 19.13, except for two differences: first there is the trivial difference due to the use of plate notation; second, in meta learning, we have both training and test partitions for all distributions, whereas in DG, we only have a test set for the target distribution.

We will learn a point estimate for the global parameters \mathbf{w}^0 , since it is shared across all datasets, and thus has little uncertainty. However, we will compute an approximate posterior for \mathbf{w}^j , since each task often has little data. We denote this posterior by $p(\mathbf{w}^j | \mathcal{D}_{\text{train}}^j, \mathbf{w}^0)$. From this, we can compute the posterior predictive distribution for each task:

$$p(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathcal{D}_{\text{train}}^j, \mathbf{w}^0) = \int p(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathbf{w}^j) p(\mathbf{w}^j | \mathcal{D}_{\text{train}}^j, \mathbf{w}^0) d\mathbf{w}^j \quad (19.28)$$

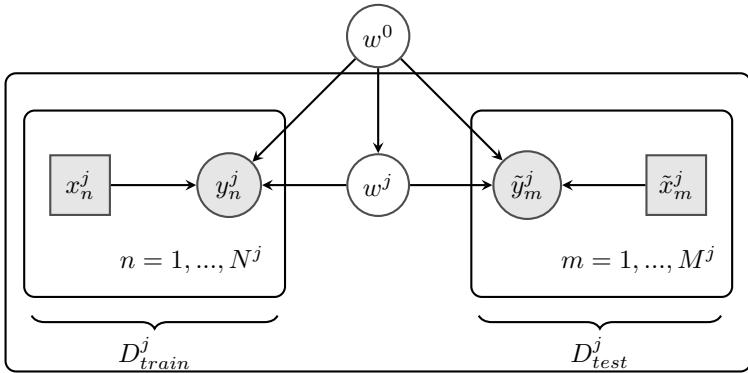


Figure 19.15: Hierarchical Bayesian model for meta-learning. There are J tasks, each of which has a training set $\mathcal{D}^j = \{(\mathbf{x}_n^j, \mathbf{y}_n^j) : n = 1 : N^j\}$ and a test set $\mathcal{D}_{\text{test}}^j = \{(\tilde{\mathbf{x}}_m^j, \tilde{\mathbf{y}}_m^j) : m = 1 : M^j\}$. \mathbf{w}^j are the task specific parameters, and θ are the shared parameters. Adapted from Figure 1 of [Gor+19].

Since computing the posterior is in general intractable, we will learn an amortized approximation (see Section 10.1.5) to the predictive distribution, denoted by $q_\phi(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathcal{D}_{\text{train}}^j, \mathbf{w}^0)$. We choose the parameters of the prior \mathbf{w}^0 and the inference network ϕ to make this *predictive posterior* as accurate as possible for any given input dataset:

$$\phi^* = \underset{\phi}{\operatorname{argmin}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\mathbf{x}})} [D_{\text{KL}}(p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}}, \mathbf{w}^0) \| q_\phi(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}}, \mathbf{w}^0))] \quad (19.29)$$

$$= \underset{\phi}{\operatorname{argmin}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\mathbf{x}})} [\mathbb{E}_{p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}}, \mathbf{w}^0)} [\log q_\phi(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}}, \mathbf{w}^0)]] \quad (19.30)$$

$$= \underset{\phi}{\operatorname{argmin}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\mathbf{x}}, \tilde{\mathbf{y}})} \left[\log \int p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathbf{w}) q_\phi(\mathbf{w} | \mathcal{D}_{\text{train}}, \mathbf{w}^0) d\mathbf{w} \right] \quad (19.31)$$

where we made the approximation $p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}}, \mathbf{w}^0) \approx p(\tilde{\mathbf{y}} | \tilde{\mathbf{x}}, \mathcal{D}_{\text{train}})$. We can then make a Monte Carlo approximation to the outer expectation by sampling J tasks (distributions) from $p(\mathcal{D})$, each of which gets partitioned into a train and test set, $\{(\mathcal{D}_{\text{train}}^j, \mathcal{D}_{\text{test}}^j) \sim p(\mathcal{D}) : j = 1 : J\}$, where $\mathcal{D}_{\text{test}}^j = \{(\tilde{\mathbf{x}}_m^j, \tilde{\mathbf{y}}_m^j)\}$. We can make an MC approximation to the inner expectation (the integral) by drawing S samples from the task-specific parameter posterior $\mathbf{w}_s^j \sim q_\phi(\mathbf{w}^j | \mathcal{D}^j, \mathbf{w}^0)$. The resulting objective has the following form (where we assume each test set has M samples for notational simplicity):

$$\mathcal{L}_{\text{meta}}(\mathbf{w}^0, \phi) = \frac{1}{MJ} \sum_{m=1}^M \sum_{j=1}^J \log \left(\frac{1}{S} \sum_{s=1}^S p(\tilde{\mathbf{y}}_m^j | \tilde{\mathbf{x}}_m^j, \mathbf{w}_s^j) \right) \quad (19.32)$$

Note that this is different from standard (amortized) variational inference, that focuses on approximating the expected accuracy of the *parameter posterior* given all of the data for a task, $\mathcal{D}_{\text{all}}^j = \mathcal{D}_{\text{train}}^j \cup \mathcal{D}_{\text{test}}^j$, rather than focusing on predictive accuracy of a test set given a training set.

Indeed, the standard objective has the form

$$\mathcal{L}_{\text{VI}}(\mathbf{w}^0, \phi) = \frac{1}{J} \sum_{j=1}^J \left(\sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}_{\text{all}}^j} \left[\frac{1}{S} \sum_{s=1}^S \log p(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathbf{w}_s^j) \right] - D_{\text{KL}} \left(q_\phi(\mathbf{w}^j | \mathcal{D}_{\text{all}}^j, \mathbf{w}^0) \| p(\mathbf{w}^j | \mathbf{w}^0) \right) \right) \quad (19.33)$$

where $\mathbf{w}_s^j \sim q_\phi(\mathbf{w}^j | \mathcal{D}_{\text{all}}^j)$. 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 have the KL penalty term on the parameter posterior.

Below we show how this framework includes several common approaches to meta-learning.

19.6.4.2 Neural processes

In the special case that the task-specific inference network computes a point estimate, $q(\mathbf{w}^j | \mathcal{D}^j, \mathbf{w}^0) = \delta(\mathbf{w}^j - \mathcal{A}_\phi(\mathcal{D}^j, \mathbf{w}^0))$, the posterior predictive distribution becomes

$$q(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathcal{D}^j, \mathbf{w}^0) = \int p(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathbf{w}^j) q(\mathbf{w}^j | \mathcal{D}^j, \mathbf{w}^0) d\mathbf{w}^j = p(\tilde{\mathbf{y}}^j | \tilde{\mathbf{x}}^j, \mathcal{A}_\phi(\mathcal{D}^j, \mathbf{w}^0), \mathbf{w}^0) \quad (19.34)$$

where $\mathcal{A}_\phi(\mathcal{D}^j, \mathbf{w}^0)$ is a function that takes in a set, and returns some parameters. We can evaluate this predictive distribution empirically, and directly optimize it (wrt ϕ and \mathbf{w}^0) using standard supervised maximum likelihood methods. This approach is called a **neural process** [Gar+18e; Gar+18d; Dub20; Jha+22]).

19.6.4.3 Gradient-based meta-learning (MAML)

In **gradient-based meta-learning**, we define the task specific inference procedure as follows:

$$\hat{\mathbf{w}}^j = \mathcal{A}(\mathcal{D}^j, \mathbf{w}^0) = \mathbf{w}^0 + \eta \nabla_{\mathbf{w}} \log \sum_{n=1}^{N^j} p(\mathbf{y}_n^j | \mathbf{x}_n^j, \mathbf{w})|_{\mathbf{w}^0} \quad (19.35)$$

That is, we set the task specific parameters to be shared parameters \mathbf{w}^0 , modified by one step along the gradient of the log conditional likelihood. 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].

19.6.4.4 Metric-based few-shot learning (prototypical networks)

Now suppose \mathbf{w}^0 correspond to the parameters of a shared neural feature extractor, $h_{\mathbf{w}^0}(\mathbf{x})$, and the task specific parameters are the weights and biases of the last linear layer of a classifier, $\mathbf{w}^j = \{\mathbf{w}_c^j, b_c^j\}_{c=1}^C$. Let us compute the average of the feature vectors for each class in each task's training set:

$$\boldsymbol{\mu}_c^j = \frac{1}{|\mathcal{D}_c^j|} \sum_{\mathbf{x}_n^c \in \mathcal{D}_c^j} h_{\mathbf{w}^0}(\mathbf{x}_n^c) \quad (19.36)$$

Now define the task specific inference procedure as follows. We first compute the vector containing the centroid and norm for each class:

$$\hat{\mathbf{w}}^j = \mathcal{A}(\mathcal{D}^j, \mathbf{w}^0) = [\boldsymbol{\mu}_c^j, -\frac{1}{2}\|\boldsymbol{\mu}_c^j\|^2]_{c=1}^C \quad (19.37)$$

The predictive distribution becomes

$$q(\tilde{y}^j = c | \tilde{\mathbf{x}}^j, \mathcal{D}^j, \mathbf{w}^0) \propto \exp(-d(h_{\mathbf{w}^0}(\tilde{\mathbf{x}}), \boldsymbol{\mu}_c^j)) = \exp\left(h_{\mathbf{w}^0}(\tilde{\mathbf{x}})^T \boldsymbol{\mu}_c^j - \frac{1}{2}\|\boldsymbol{\mu}_c^j\|^2\right) \quad (19.38)$$

where $d(\mathbf{u}, \mathbf{v})$ is the Euclidean distance. This is equivalent to the technique known as **prototypical networks** [SSZ17].

19.7 Continual learning

In this section, we discuss **continual learning** (see e.g., [Had+20; Del+21; Qu+21; LCR21; Mai+22; Wan+23]), also called **life-long learning** (see e.g., [Thr98; CL18]), in which the system learns from a sequence of different distributions, p_1, p_2, \dots . In particular, at each time step t , the model receives a batch of labeled data,

$$\mathcal{D}_t = \{(\mathbf{x}_n, \mathbf{y}_n) \sim p_t(\mathbf{x}, \mathbf{y}) : n = 1 : N_t\} \quad (19.39)$$

where $p_t(\mathbf{x}, \mathbf{y})$ is the unknown data distribution, which we represent as $p_t(\mathbf{x}, \mathbf{y}) = p_t(\mathbf{x})p(\mathbf{y}|f_t(\mathbf{x}))$, where $f_t : \mathcal{X}_t \rightarrow \mathcal{Y}_t$ is the unknown prediction function. Each distribution defines a different **task**. The learner is then expected to update its belief state about the underlying distribution, and to use its beliefs to make predictions on an independent test set,

$$\mathcal{D}_t^{\text{test}} = \{(\mathbf{x}_n, \mathbf{y}_n) \sim p_t^{\text{test}}(\mathbf{x}, \mathbf{y}) : n = 1 : N_t^{\text{test}}\} \quad (19.40)$$

Depending on how we assume $p_t(\mathbf{x}, \mathbf{y})$ evolve over time, and how the test set is defined, we can create a variety of different CL scenarios. In particular, if the test distribution at time t contains samples from all the tasks up to (and including) time t , then we require that the model not “forget” past data, which can be tricky for many methods, as discussed in Section 19.7.4. By contrast, if the test distribution at time t is same as the current distribution, as in online learning (Section 19.7.5), then we just require that the learner adapt to changes, but it need not remember the past. (Note that we focus on supervised problems, but non-stationarity also arises in reinforcement learning; in particular, the input distribution changes due to the agent’s changing policy, and the desired prediction function changes due to the value function for that policy being updated.)

19.7.1 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 (see e.g., [Sun+22]).

To evaluate such a model, we assume $f_t^{\text{test}} = f_t$ and 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).

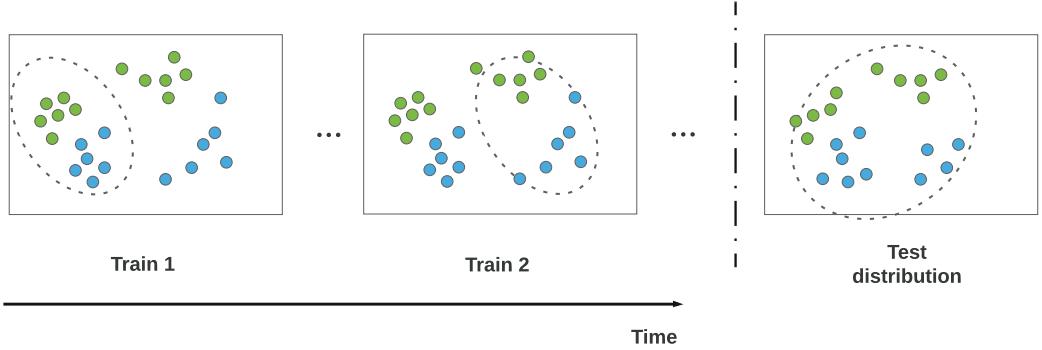


Figure 19.16: An illustration of domain drift.

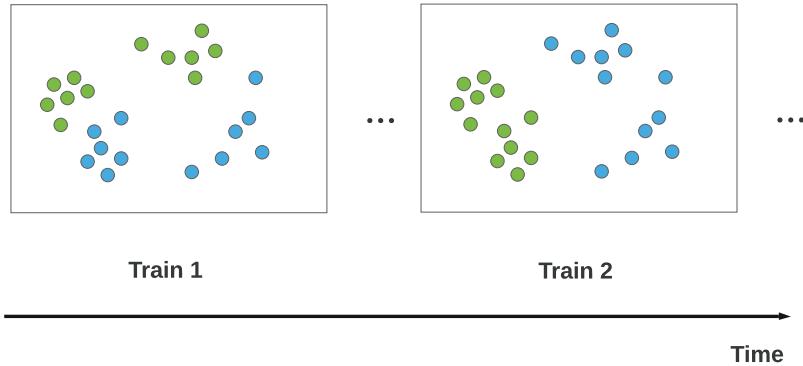


Figure 19.17: An illustration of concept drift.

Alternatively we can define $p_t^{\text{test}}(\mathbf{x})$ to be the union of all the input distributions seen so far, $p_t^{\text{test}} = \cup_{s=1}^T p_s$ (e.g., we want the detector to work well on dark and light images). This latter assumption is illustrated in Figure 19.16.

19.7.2 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 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 19.17.

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

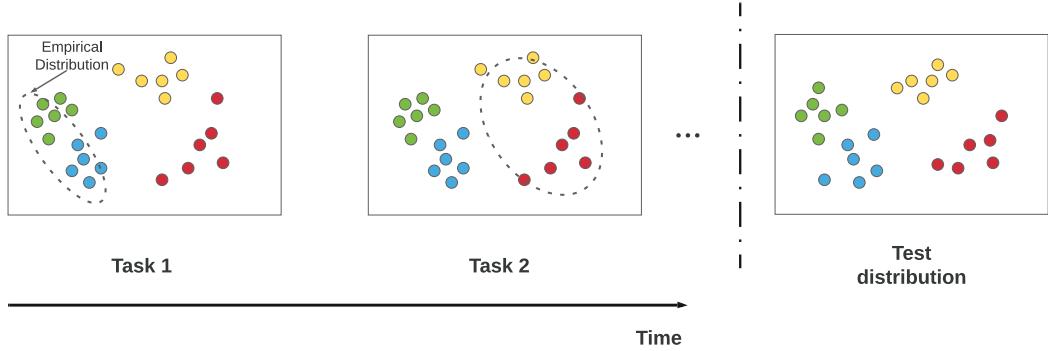


Figure 19.18: An illustration of class incremental learning. Adapted from Figure 1 of [LCR21].

left and circular objects go on the right.⁴ 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.

19.7.3 Class 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, consider the problem of digit classification from images. \mathcal{Y}_1 might be $\{0, 1\}$, and \mathcal{Y}_2 might be $\{2, \dots, 9\}$. Learning to classify with an increasing number of categories is called **class incremental learning** (see e.g., [Mas+20]). See Figure 19.18 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 19.19. If we assume there are no well defined boundaries between tasks, 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 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 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**).⁵ In the task-aware setting, the task label may or may not be known at test time.

4. This example is from Mike Mozer.

5. 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).

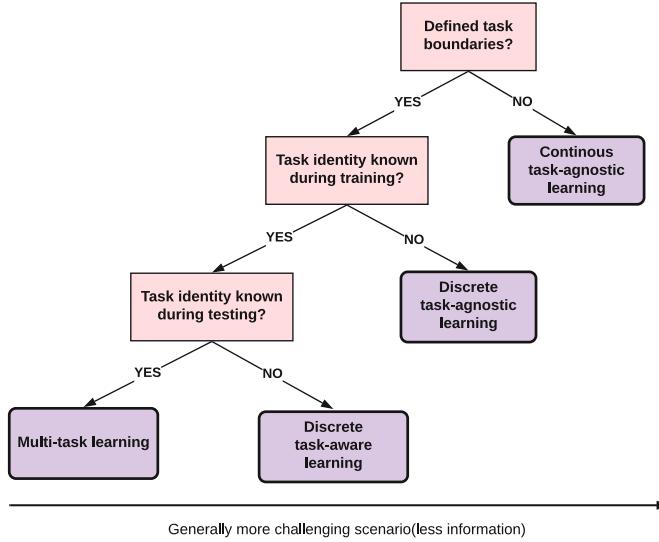


Figure 19.19: Different kinds of incremental learning. Adapted from Figure 1 of [Zen+18].

If it is, the problem is essentially equivalent to multi-task learning (see Section 19.6.1). 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 change (see e.g., [DLT21]). If we fit a separate generative model per class (e.g., a VAE, as in [VLT21]), then 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; Wol+21]). Thus future research should focus on the task-agnostic setting, with either discrete or continuous changes.

So the “meaning” of the output label depends on what task we are solving. Thus the output space is really hierarchical, namely the cross product of task id and class label.

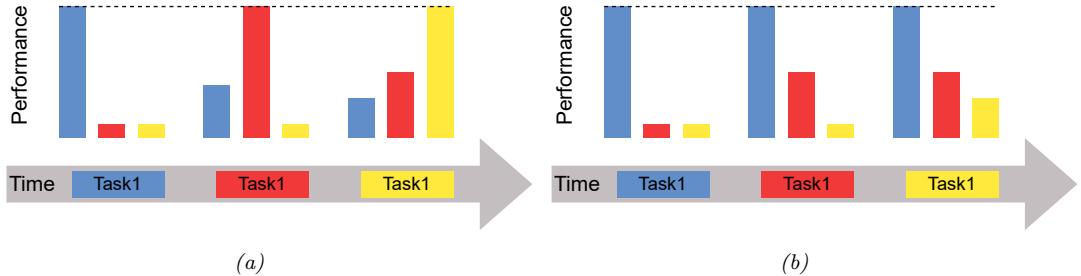


Figure 19.20: Some failure modes in class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) Catastrophic forgetting refers to the phenomenon in which performance on a previous task drops when trained 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].

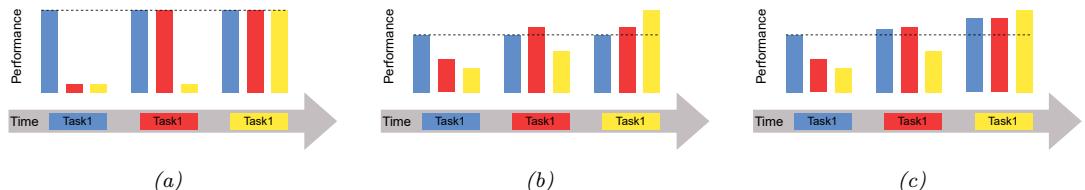


Figure 19.21: What success looks like for class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) No forgetting refers to the phenomenon in which performance on previous tasks does not degrade over time. (b) Forwards 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].

19.7.4 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 19.20(a)). Another possible problem is that only the first task is learned, and the model does not adapt to new tasks (see Figure 19.20(b)).

If we avoid these problems, we should expect to see the performance profile in Figure 19.21(a), where performance of incremental training is equal to training on each task separately. However, we might hope to do better by virtue of the fact that we are training on multiple tasks, which are often assumed to be related. In particular, we might hope to see **forwards transfer**, in which training on past tasks improves performance on future tasks beyond what would have been obtained by training from scratch (see Figure 19.21(b)). Additionally, we might hope to see **backwards transfer**, in which training on future tasks improves performance on past tasks (see Figure 19.21(c)).

We can quantify the degree of transfer as follows, following [LPR17]. If R_{ij} is the performance on task j after it was trained on task i , R_j^{ind} is the performance on task j when trained just on j , and there are T tasks, then the amount of forwards transfer is

$$\text{FWT} = \frac{1}{T} \sum_{j=1}^T R_{j,j} - R_j^{\text{ind}} \quad (19.41)$$

and the amount of backwards transfer is

$$\text{BWT} = \frac{1}{T} \sum_{j=1}^T R_{T,j} - R_{j,j} \quad (19.42)$$

There are many methods that have been devised to overcome the problem of catastrophic forgetting, but we can group them into three main types. The first is **regularization methods**, which add a loss to preserve information that is relevant to old tasks. (For example, online Bayesian inference is of this type, since the posterior for the parameters is derived from the new data and the past prior; see e.g., the **elastic weight consolidation** method discussed in Section 17.5.1, or the **variational continual learning** method discussed in Supplementary Section 10.2). The second is **memory methods**, which rely on some kind of **experience replay** or **rehearsal** of past data (see e.g., [Hen+21]), or some kind of generative model of past data. The third is **architectural methods**, that add capacity to the network whenever a task boundary is encountered, such as a new class label (see e.g., [Rus+16]).

Of course, these techniques can be combined. For example, we can create a semi-parametric model, in which we store some past data (exemplars) while also learning parameters online in a Bayesian (regularized) way (see e.g., [Kur+20]). The “right” method depends, as usual, on what inductive bias you want to use, and want your computational budget is in terms of time and memory.

19.7.5 Online learning

The problem of **online learning** is similar to continual learning, except the loss metric is different, and we usually assume that learning and evaluation occur at each step. More precisely, we assume the data generating distribution, $p_t^*(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\phi_t)p(\mathbf{y}|\mathbf{x}, \mathbf{w}_t)$, evolves over time, as shown in Figure 19.22. At each step t nature generates a data sample, $(\mathbf{x}_t, \mathbf{y}_t) \sim p_t^*$. The agent sees \mathbf{x}_t and is asked to predict \mathbf{y}_t by computing the posterior predictive distribution

$$\hat{p}_{t|t-1} = p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:t-1}) \quad (19.43)$$

where $\mathcal{D}_{1:t-1} = \{(\mathbf{x}_s, \mathbf{y}_s) : s = 1 : t-1\}$ is all past data. It then incurs a loss of

$$\mathcal{L}_t = \ell(\hat{p}_{t|t-1}, \mathbf{y}_t) \quad (19.44)$$

See Figure 19.22. This approach is called **prequential prediction** [DV99; GSR13], and also forms the basis of online conformal prediction [VGS22].

In contrast to the continual learning scenarios studied above, the loss incurred at each step is what matters, rather than loss on a fixed test set. That is, we want to minimize $\mathcal{L} = \sum_{t=1}^T \mathcal{L}_t$. In the case of log-loss, this is equal to the (conditional) log marginal likelihood of the data, $\log p(\mathcal{D}_{1:T}) =$

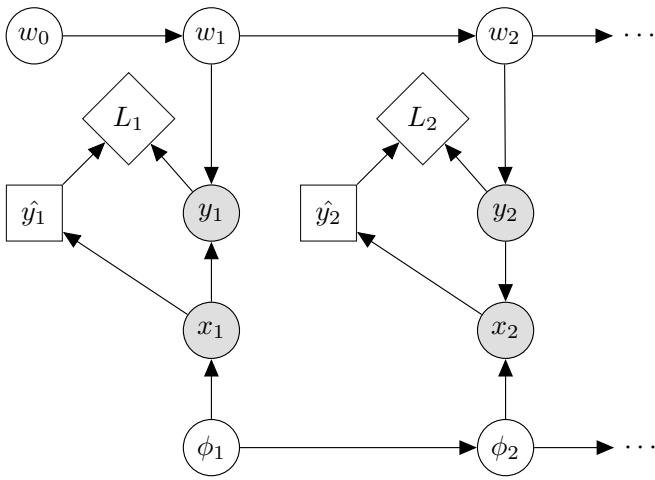


Figure 19.22: Online learning illustrated as an influence diagram (Section 34.2). Here $\hat{y}_t = \operatorname{argmax}_y p(y|\mathbf{x}_t, \mathcal{D}_{1:t-1})$ is the action (MAP predicted output) at time t , and $L_t = \ell(y_t, \hat{y}_t)$ is the corresponding loss (utility) function. We then update the parameters of the model, $\theta_t = (\mathbf{w}_t, \phi_t)$, given the input and true output $(\mathbf{x}_t, \mathbf{y}_t)$. The parameters of the world model can change arbitrarily over time.

$\log p(\mathbf{y}_{1:T}|\mathbf{x}_{1:T})$. This can be used to compute the prequential minimum description length (MDL) of a model [BLH22], which is useful for model selection.

Another metric that is widely used, especially if it is assumed that the distributions can be generated by an adversary, is to compare the cumulative loss to the optimal value one could have obtained in hindsight. This yields a quantity called the **regret**:

$$\text{regret} = \sum_{t=1}^T [\ell(\hat{p}_{t|t-1}, \mathbf{y}_t) - \ell(\hat{p}_{t|T}, \mathbf{y}_t)] \quad (19.45)$$

where $\hat{p}_{t|t-1} = p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:t-1})$ is the online prediction, and $\hat{p}_{t|T} = p(\mathbf{y}|\mathbf{x}_t, \mathcal{D}_{1:T})$ is the optimal estimate at the end of training. Bounds on the regret can be derived when the loss is convex [Ora19; Haz22]. 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 34.4) and reinforcement learning (see Chapter 35), where the agent “lives forever”, and where there is no fixed training phase followed by a test phase. (See e.g., Section 17.5 where we discuss online Bayesian inference for neural networks.)

The previous continual learning scenarios can be derived as special cases of online learning: we use a different distribution (task) per time step, and provide a set of examples as input, instead of a single example. On odd time steps, we train on the data from the current distribution, and incur a loss of 0; and on even time steps, we evaluate on the test distribution, which may consist of the union of all previously seen tasks, and return the empirical loss. (Thus doing well on old distributions is relevant because we assume such distributions keep recurring.) Typically in CL the amount of

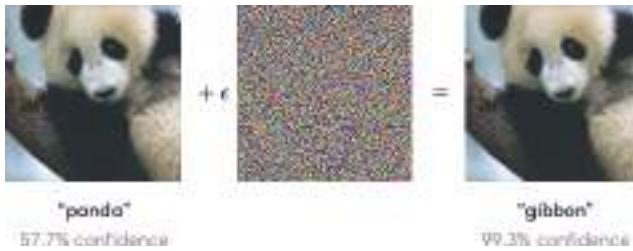


Figure 19.23: Example of an adversarial attack on an image classifier. Left column: original image which is correctly classified. Middle column: small amount of structured noise which is added to the input (magnitude of noise is magnified by 10 \times). Right column: new image, which is confidently misclassified 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.

data per task is large, whereas online learning is more concerned with fast adaptation to slowly (or piecewise continuously) changing distributions using small amounts of data per time step.

19.8 Adversarial examples

This section is coauthored with Justin Gilmer.

In Section 19.2, 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:

$$\mathbf{x}_{\text{adv}} = \underset{\mathbf{x}' \in \Delta(\mathbf{x})}{\operatorname{argmin}} \log p(y = c | \mathbf{x}') \quad (19.46)$$

where $\Delta(\mathbf{x})$ is the set of images that are “similar” to \mathbf{x} (we discuss different notions of similarity below).

Equation (19.46) is an example of an **adversarial attack**. We illustrate this in Figure 19.23. 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%.

The ability to create adversarial images was first noted in [Sze+14]. It is surprisingly easy to create such examples, which seems paradoxical, given the fact that modern classifiers seem to work so well on normal inputs, and the perturbed images “look” the same to humans. We explain this paradox in Section 19.8.5.

The existence of adversarial images also raises security concerns. For example, [Sha+16] showed they could force a face recognition system to misclassify person A as person B , merely by asking person A to wear a pair of sunglasses with a special pattern on them, and [Eyk+18] show that is

possible to attach small “**adversarial stickers**” to traffic signs to classify stop signs as speed limit signs.

Below we briefly discuss how to create adversarial attacks, why they occur, and how we can try to defend against them. We focus on the case of deep neural nets for images, although it is important to note that many other kinds of models (including logistic regression and generative models) can also suffer from adversarial attacks. Furthermore, this is not restricted to the image domain, but occurs with many kinds of high dimensional inputs. For example, [Li+19] contains an audio attack and [Dal+04; Jia+19] contains a text attack. More details on adversarial examples can be found in e.g., [Wiy+19; Yua+19].

19.8.1 Whitebox (gradient-based) attacks

To create an adversarial example, we must find a “small” perturbation $\boldsymbol{\delta}$ to add to the input \mathbf{x} to 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 force the system to output. This is known as a **targeted attack**. Alternatively, we may just want to find a perturbation that causes the current predicted label to change from its current value to any other value, so that $f(\mathbf{x} + \boldsymbol{\delta}) \neq f(\mathbf{x})$, which is known as **untargeted attack**.

In general, we define the objective for the adversary as *maximizing* the following loss:

$$\mathbf{x}_{\text{adv}} = \underset{\mathbf{x}' \in \Delta(\mathbf{x})}{\operatorname{argmax}} \mathcal{L}(\mathbf{x}', y; \boldsymbol{\theta}) \quad (19.47)$$

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 minimize the probability of the true label; and for the targeted case, we can define $\mathcal{L}(\mathbf{x}', y; \boldsymbol{\theta}) = \log p(y'|\mathbf{x}')$, where we maximize the probability of the desired label $y' \neq y$.

To define what we mean by “small” perturbation, we impose the constraint that $\mathbf{x}_{\text{adv}} \in \Delta(\mathbf{x})$, which is the set of “perceptually similar” images to the input \mathbf{x} . Most of the literature has focused on a simplistic setting in which the adversary is restricted to making bounded l_p perturbations of a clean input \mathbf{x} , that is

$$\Delta(\mathbf{x}) = \{\mathbf{x}' : \|\mathbf{x}' - \mathbf{x}\|_p < \epsilon\} \quad (19.48)$$

Typically people assume $p = 1$ or $p = 0$. We will discuss more realistic threat models in Section 19.8.3.

In this section, we assume that the attacker knows the model parameters $\boldsymbol{\theta}$; this is called a **whitebox attack**, and lets us use gradient based optimization methods. We relax this assumption in Section 19.8.2.)

To solve the optimization problem in Equation (19.47), we can use any kind of constrained optimization method. In [Sze+14] they used bound-constrained BFGS. [GSS15] proposed the more efficient **fast gradient sign (FGS)** method, which performs iterative updates of the form

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \boldsymbol{\delta}_t \quad (19.49)$$

$$\boldsymbol{\delta}_t = \epsilon \operatorname{sign}(\nabla_{\mathbf{x}} \log p(y'|\mathbf{x}, \boldsymbol{\theta})|_{\mathbf{x}_t}) \quad (19.50)$$

where $\epsilon > 0$ is a small learning rate. (Note that this gradient is with respect to the input pixels, not the model parameters.) Figure 19.23 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



Figure 19.24: 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.

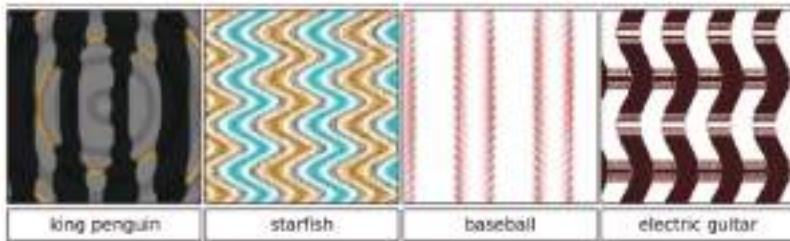


Figure 19.25: Synthetic images that cause the CNN to confidently predict a specific class. From Figure 1 of [NYC15]. Used with kind permission of Jeff Clune.

solving 19.47. 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].

19.8.2 Blackbox (gradient-free) attacks

In this section, we no longer assume that the adversary knows the parameters θ of the predictive model f . This is known as a **black box attack**. In such cases, we must use derivative-free optimization (DFO) methods (see Section 6.7).

Evolutionary algorithms (EA) are one class of DFO solvers. These were used in [NYC15] to create blackbox attacks. Figure 19.24 shows some images that were generated by applying an EA to a random noise image. These are known as **fooling images**, as opposed to adversarial images, since they are not visually realistic. Figure 19.25 shows some fooling images that were generated by applying EA to the parameters of a compositional pattern-producing network (CPPN) [Sta07].⁶ By suitably perturbing the CPPN parameters, it is possible to generate structured images with high fitness (classifier score), but which do not look like natural images [Aue12].

6. A CPPN is a set of elementary functions (such as linear, sine, sigmoid, and Gaussian) which can be composed in order to specify the mapping from each coordinate to the desired color value. CPPN was originally developed as a way to encode abstract properties such as symmetry and repetition, which are often seen during biological development.

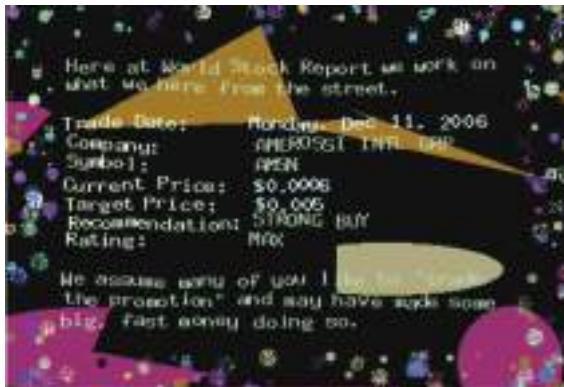


Figure 19.26: An adversarially modified image to evade spam detectors. The image is constructed from scratch, and does not involve applying a small perturbation to any given image. This is an illustrative example of how large the space of possible adversarial inputs Δ can be when the attacker has full control over the input. From [Big+11]. Used with kind permission of Battista Biggio.

In [SVK19], they used differential evolution to attack images by modifying a single pixel. This is equivalent to bounding the ℓ_0 norm of the perturbation, so that $\|\mathbf{x}_{\text{adv}} - \mathbf{x}\|_0 = 1$.

In [Pap+17], they learned a differentiable surrogate model of the blackbox, by just querying its predictions y for different inputs \mathbf{x} . They then used gradient-based methods to generate adversarial attacks on their surrogate model, and then showed that these attacks transferred to the real model. In this way, they were able to attack various the image classification APIs of various cloud service providers, including Google, Amazon, and MetaMind.

19.8.3 Real world adversarial attacks

Typically, the space of possible adversarial inputs Δ can be quite large, and will be difficult to exactly define mathematically as it will depend on semantics of the input based on the attacker's goals [BR18]. (The set of variations Δ that we want the model to be invariant to is called the **threat model**.)

Consider for example of the content constrained threat model discussed in [Gil+18a]. One instance of this threat model involves image spam, where the attacker wishes to upload an image attachment in an email that will not be classified as spam by a detection model. In this case Δ is incredibly large as it consists of all possible images which contain some semantic concept the attacker wishes to upload (in this case an advertisement). To explore Δ , spammers can utilize different fonts, word orientations or add random objects to the background as is the case of the adversarial example in Figure 19.26 (see [Big+11] for more examples). Of course, optimization based methods may still be used here to explore parts of Δ . However, in practice it may be preferable to design an adversarial input by hand as this can be significantly easier to execute with only limited-query black-box access to the underlying classifier.

19.8.4 Defenses based on robust optimization

As discussed in Section 19.8.3, securing a system against adversarial inputs in more general threat models seems extraordinarily difficult, due to the vast space of possible adversarial inputs Δ . However, there is a line of research focused on producing models which are invariant to perturbations within a small constraint set $\Delta(\mathbf{x})$, with a focus on l_p -robustness where $\Delta(\mathbf{x}) = \{\mathbf{x}' : \|\mathbf{x} - \mathbf{x}'\|_p < \epsilon\}$. Although solving this toy threat model has little application to security settings, enforcing smoothness priors has in some cases improved robustness to random image corruptions [SHS], led to models which transfer better [Sal+20], and has biased models towards different features in the data [Yin+19a].

Perhaps the most straightforward method for improving l_p -robustness is to directly optimize for it through **robust optimization** [BTEGN09], also known as **adversarial training** [GSS15]. We define the **adversarial risk** to be

$$\min_{\theta} \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p(\mathbf{x}, \mathbf{y})} \left[\max_{\mathbf{x}' \in \Delta(\mathbf{x})} L(\mathbf{x}', \mathbf{y}; \theta) \right] \quad (19.51)$$

The min max formulation in equation 19.51 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 $\mathbf{x}_{\text{adv}} \approx \text{argmax}_{\mathbf{x}'} L(\mathbf{x}', \mathbf{y}; \theta)$ [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].

19.8.5 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 19.2. 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 dimensional, then we should expect the nearest error to be significantly closer than errors which are sampled randomly from some out-of-distribution $p_{\text{te}}(\mathbf{x})$.

A cartoon illustration of what is going on is shown in Figure 19.27a, where \mathbf{x}_0 is the clean input image, B is an image corrupted by Gaussian noise, and A is an adversarial image. If we assume a linear decision boundary, then the error set E is a half space a certain distance from \mathbf{x}_0 . We can relate the distance to the decision boundary $d(\mathbf{x}_0, E)$ with the error rate in noise at some input \mathbf{x}_0 , 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



Figure 19.27: (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 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 a. Used with kind permission of Justin Gilmer.

these two quantities is determined by

$$d(\mathbf{x}_0, E) = -\sigma\Phi^{-1}(\mu) \tag{19.52}$$

where Φ^{-1} denotes the inverse cdf of the gaussian distribution. When the input dimension is large, this distance will be significantly smaller than the distance to a randomly sampled noisy image $\mathbf{x}_0 + \delta$ 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 concrete example consider the ImageNet dataset, where $d = 224 \times 224 \times 3$ and suppose we set $\sigma = .2$. Then if the error rate in noise is just $\mu = .01$, equation 19.52 will imply that $d(\mathbf{x}_0, E) = .5$. Thus the distance to an adversarial example will be more than 100 times closer than the distance to a typical noisy images, which will be $\sigma\sqrt{d} \approx 77.6$. This phenomenon of small volume error sets being close to most points in a data distribution $p(\mathbf{x})$ is called **concentration of measure**, and is a property common among many high dimensional data distributions [MDM19; Gil+18b].

In summary, although the existence of adversarial examples is often discussed as an unexpected phenomenon, there is nothing special about the existence of worst-case errors for ML classifiers — they will always exist as long as errors exist.

PART IV

Generation

20 Generative models: an overview

20.1 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 models. We give a brief summary in Section 20.2, and go into more detail in subsequent chapters. See also [Tom22] for a recent book on this topic that goes into more depth.

20.2 Types of generative model

There are many kinds of generative model, some of which we list in Table 20.1. At a high level, we can distinguish between **deep generative models** (DGM) — which use deep neural networks 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 28, Chapter 29, Chapter 30. In this part of the book, we mostly focus on DGMs.

The main kinds of DGM are: **variational autoencoders (VAE)**, **autoregressive models (ARM)** 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 20.1 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, UPGMs).
- 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, ARM, 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	Section 4.2	Exact, fast	Fast	MLE	Optional	Sparse DAG
PGM-U	Section 4.3	Approx, slow	Slow	MLE-A	Optional	Sparse graph
VAE	Chapter 21	LB, fast	Fast	MLE-LB	\mathbb{R}^L	Encoder-Decoder
ARM	Chapter 22	Exact, fast	Slow	MLE	None	Sequential
Flows	Chapter 23	Exact, slow/fast	Slow	MLE	\mathbb{R}^D	Invertible
EBM	Chapter 24	Approx, slow	Slow	MLE-A	Optional	Discriminative
Diffusion	Chapter 25	LB	Slow	MLE-LB	\mathbb{R}^D	Encoder-Decoder
GAN	Chapter 26	NA	Fast	Min-max	\mathbb{R}^L	Generator-Discriminator

Table 20.1: Characteristics of common kinds of generative model. Here D is the dimensionality of the observed \mathbf{x} , and L is the dimensionality of the latent \mathbf{z} , if present. (We usually assume $L \ll D$, although overcomplete representations can have $L \gg D$.) Abbreviations: Approx = approximate, ARM = autoregressive model, EBM = energy based model, GAN = generative adversarial network, MLE = maximum likelihood estimation, MLE-A = MLE (approximate), MLE-LB = MLE (lower bound), NA = not available, PGM = probabilistic graphical model, PGM-D = directed PGM, PGM-U = undirected PGM, VAE = variational autoencoder.

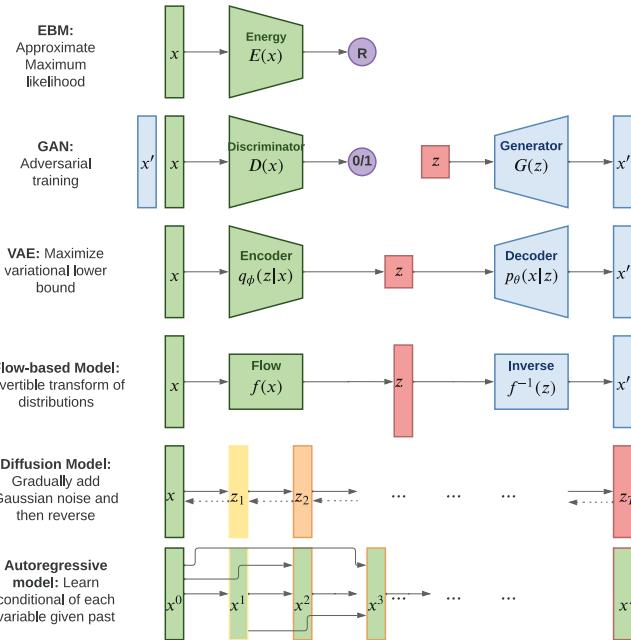


Figure 20.1: Summary of various kinds of deep generative models. Here \mathbf{x} is the observed data, \mathbf{z} is the latent code, and \mathbf{x}' is a sample from the model. AR models do not have a latent code \mathbf{z} . For diffusion models and flow models, the size of \mathbf{z} is the same as \mathbf{x} . For AR models, x^d is the d 'th dimension of \mathbf{x} . R represents real-valued output, $0/1$ represents binary output. Adapted from Figure 1 of [Wen21].

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, ARMs do not use latents; flows and diffusion use latents, but they are not compressed.¹ 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.

20.3 Goals of generative modeling

There are several different kinds of tasks that we can use generative models for, as we discuss below.

20.3.1 Generating data

One of the main goals of generative models is to generate (create) new data samples. This is sometimes called **generative AI** (see e.g., [GBGM23] for a recent survey). 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 25.10.² 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** (see e.g., [Ngu+19]). Generative models can also be used to create **synthetic data** for training discriminative models (see e.g., [Wil+20; Jor+22]).

To control what is generated, it is useful to use a **conditional generative model** of the form $p(\mathbf{x}|\mathbf{c})$. Here are some examples:

- \mathbf{c} = text prompt, \mathbf{x} = image. This is a **text-to-image** model (see Figure 20.2, Figure 20.3 and Figure 22.6 for examples).
- \mathbf{c} = image, \mathbf{x} = text. This is an **image-to-text** model, which is useful for **image captioning**.
- \mathbf{c} = image, \mathbf{x} = image. This is an **image-to-image** model, and can be used for image colorization, inpainting, uncropping, JPEG artefact restoration, etc. See Figure 20.4 for examples.
- \mathbf{c} = sequence of sounds, \mathbf{x} = sequence of words. This is a **speech-to-text** model, which is useful for **automatic speech recognition (ASR)**.
- \mathbf{c} = sequence of English words, \mathbf{x} = sequence of French words. This is a **sequence-to-sequence** model, which is useful for **machine translation**.

1. Flow models define a latent vector \mathbf{z} that has the same size as \mathbf{x} , although the internal deterministic computation may use vectors that are larger or smaller than the input (see e.g., the DenseFlow paper [GGS21]).

2. These images were made with a technique called score-based generative modeling (Section 25.3), 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 26).



(a) *Teddy bears swimming at the Olympics 400m Butterfly event.*



(b) *A cute corgi lives in a house made out of sushi.*



(c) *A cute sloth holding a small treasure chest. A bright golden glow is coming from the chest.*

Figure 20.2: Some 1024×1024 images generated from text prompts by the Imagen diffusion model (Section 25.6.4). From Figure 1 of [Sah+22b]. Used with kind permission of William Chan.



A portrait photo of a kangaroo wearing an orange hoodie and blue sunglasses standing on the grass in front of the Sydney Opera House holding a sign on the chest that says Welcome Friends!

Figure 20.3: Some images generated from the Parti transformer model (Section 22.4.2) in response to a text prompt. We show results from models of increasing size (350M, 750M, 3B, 20B). Multiple samples are generated, and the highest ranked one is shown. From Figure 10 of [Yu+22]. Used with kind permission of Jiahui Yu.

- c = initial prompt, x = continuation of the text. This is another sequence-to-sequence model, which is useful for automatic **text generation** (see Figure 22.5 for an example).

Note that, in the conditional case, we sometimes 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 a 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 20.4.

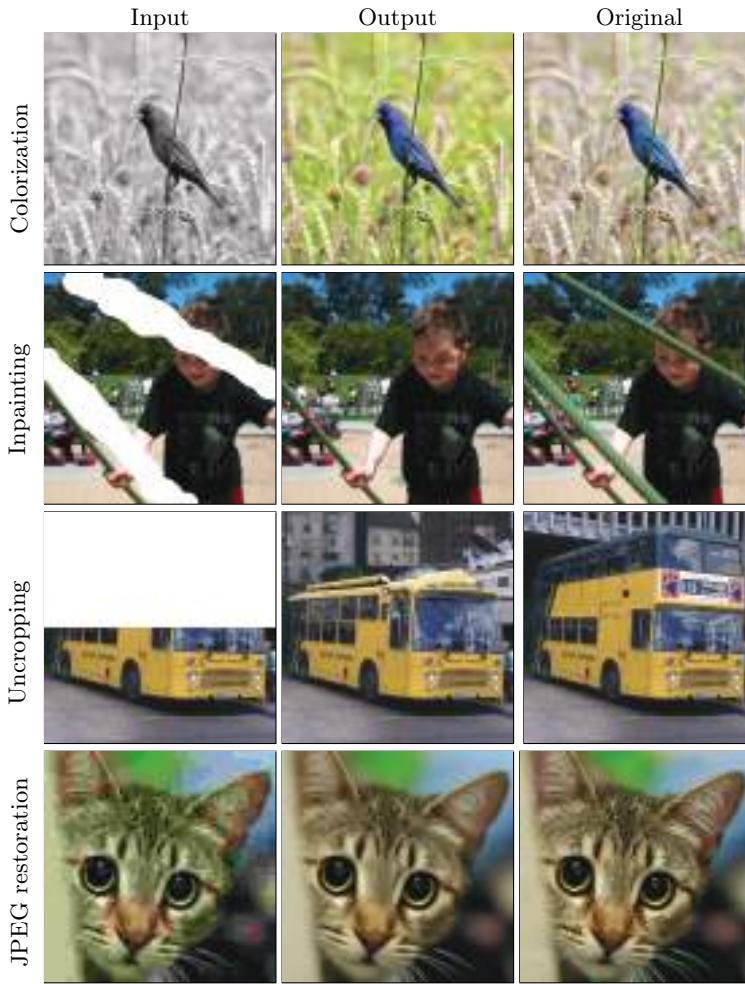


Figure 20.4: Illustration of some image-to-image tasks using the Palette conditional diffusion model (Section 25.6.4). From Figure 1 of [Sah+22a]. Used with kind permission of Chitwan Saharia.

20.3.2 Density estimation

The task of **density estimation** refers to evaluating the probability of an observed data vector, i.e., computing $p(\mathbf{x})$. This can be useful for outlier detection (Section 19.3.2), data compression (Section 5.4), generative classifiers, model comparison, etc.

A simple approach to this problem, which works in low dimensions, is to use **kernel density**

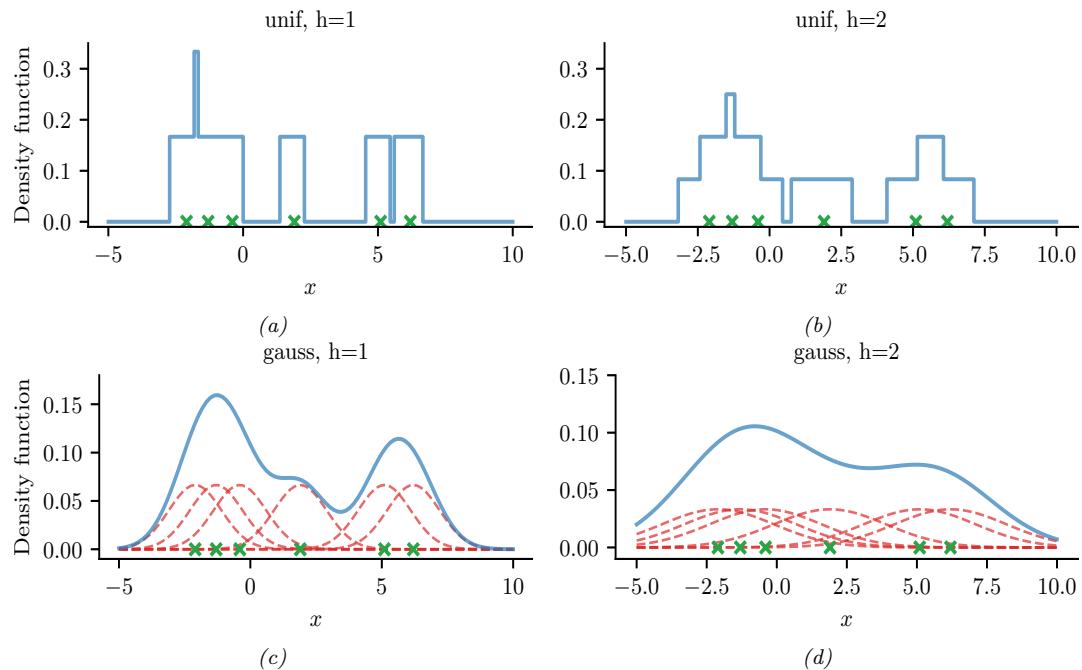


Figure 20.5: A nonparametric (Parzen) density estimator in 1d estimated from 6 datapoints, denoted by x . Top row: uniform kernel. Bottom row: Gaussian kernel. Left column: bandwidth parameter $h = 1$. Right column: bandwidth parameter $h = 2$. Adapted from http://en.wikipedia.org/wiki/Kernel_density_estimation. Generated by `parzen_window_demo.ipynb`.

estimation or **KDE**, which has the form

$$p(\mathbf{x}|\mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) \quad (20.1)$$

Here $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ is the data, and \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 20.5: 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.

20.3.3 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

Data sample	Variables			Missing values replaced by means		
	A	B	C	A	B	C
1	6	6	NA	2	6	7.5
2	NA	6	0	9	6	0
3	NA	6	NA	9	6	7.5
4	10	10	10	10	10	10
5	10	10	10	10	10	10
6	10	10	10	10	10	10
Average	9	8	7.5	9	8	7.5

Figure 20.6: Missing data imputation using the mean of each column.

illustrated in Figure 20.6. 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 Figure 20.4).

See Section 3.11 for a more general discussion of missing data.

20.3.4 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 $p(\mathbf{z}|\mathbf{x}) \propto p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$. This can be useful for discovering latent, low-dimensional patterns in the data.

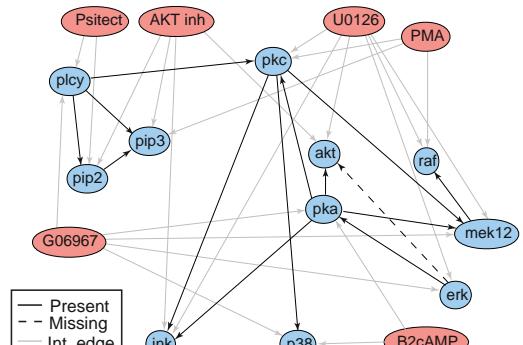
For example, suppose we perturb various proteins in a cell and measure the resulting phosphorylation state using a technique known as flow cytometry, as in [Sac+05]. An example of such a dataset is shown in Figure 20.7(a). Each row represents a data sample $\mathbf{x}_n \sim p(\cdot|\mathbf{a}_n, \mathbf{z})$, where $\mathbf{x} \in \mathbb{R}^{11}$ is a vector of outputs (phosphorylations), $\mathbf{a} \in \{0, 1\}^6$ is a vector of input actions (perturbations) and \mathbf{z} is the unknown cellular signaling network structure. We can infer the graph structure $p(\mathbf{z}|\mathcal{D})$ using graphical model structure learning techniques (see Section 30.3). In particular, we can use the dynamic programming method described in [EM07] to get the result is shown in Figure 20.7(b). Here we plot the median graph, which includes all edges for which $p(z_{ij} = 1|\mathcal{D}) > 0.5$. (For a more recent approach to this problem, see e.g., [Bro+20b].)

20.3.5 Latent space interpolation

One of the most interesting abilities of certain latent variable models is the ability to generate samples that have certain desired properties by interpolating between existing datapoints in latent 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 $\mathbf{z}_2 = e(\mathbf{x}_2)$ be their latent encodings. (The method used for computing these will depend on the



(a)



(b)

Figure 20.7: (a) A design matrix consisting of 5400 datapoints (rows) measuring the state (using flow cytometry) of 11 proteins (columns) under different experimental conditions. The data has been discretized into 3 states: low (black), medium (grey), and high (white). Some proteins were explicitly controlled using activating or inhibiting chemicals. (b) A directed graphical model representing dependencies between various proteins (blue circles) and various experimental interventions (pink ovals), which was inferred from this data. We plot all edges for which $p(G_{ij} = 1 | \mathcal{D}) > 0.5$. Dotted edges are believed to exist in nature but were not discovered by the algorithm (1 false negative). Solid edges are true positives. The light colored edges represent the effects of intervention. From Figure 6d of [EM07].

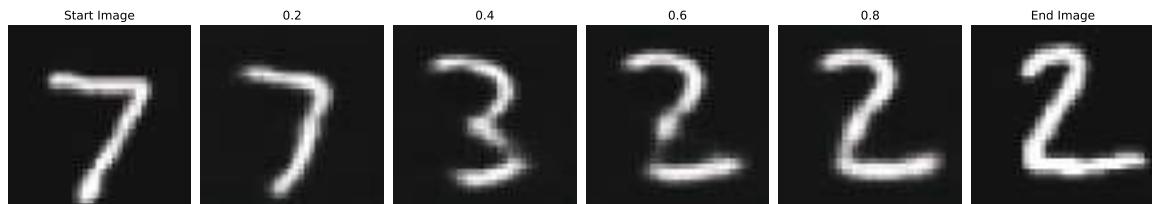


Figure 20.8: Interpolation between two MNIST images in the latent space of a β -VAE (with $\beta = 0.5$). Generated by [mnist_vae_ae_comparison.ipynb](#).



Figure 20.9: Interpolation between two CelebA images in the latent space of a β -VAE (with $\beta = 0.5$). Generated by [celeba_vae_ae_comparison.ipynb](#).



Figure 20.10: Arithmetic in the latent space of a β -VAE (with $\beta = 0.5$). The first column is an input image, with embedding \mathbf{z} . Subsequent columns show the decoding of $\mathbf{z} + s\Delta$, where $s \in \{-2, -1, 0, 1, 2\}$ and $\Delta = \bar{\mathbf{z}}^+ - \bar{\mathbf{z}}^-$ is the difference in the average embeddings of images with or without a certain attribute (here, wearing sunglasses). Generated by [celeba_vae_ae_comparison.ipynb](#).

type of model; we discuss the details in later chapters.) We can regard \mathbf{z}_1 and \mathbf{z}_2 as two “anchors” in latent space. We can now generate new images that interpolate between these points by computing $\mathbf{z} = \lambda\mathbf{z}_1 + (1 - \lambda)\mathbf{z}_2$, where $0 \leq \lambda \leq 1$, and then decoding by computing $\mathbf{x}' = d(\mathbf{z})$, where $d()$ is the decoder. This is called **latent space interpolation**, and will generate data that combines semantic features from both \mathbf{x}_1 and \mathbf{x}_2 . (The justification for taking a linear interpolation is that the learned manifold often has approximately zero curvature, as shown in [SKTF18]. However, sometimes it is better to use nonlinear interpolation [Whi16; MB21; Fad+20].)

We can see an example of this process in Figure 20.8, where we use a β -VAE model (Section 21.3.1) fit to the MNIST dataset. We see that the model is able to produce plausible interpolations between the digit 7 and the digit 2. As a more interesting example, we can fit a β -VAE to the **CelebA** dataset [Liu+15].³ The results are shown in Figure 20.9, and look reasonable. (We can get much better quality if we use a larger model trained on more data for a longer amount of time.)

It is also possible to perform interpolation in the latent space of text models, as illustrated in Figure 21.7.

20.3.6 Latent space arithmetic

In some cases, we can go beyond interpolation, and can perform **latent space arithmetic**, in which we can increase or decrease the amount of a desired “semantic factor of variation”. This was first shown in the **word2vec** model [Mik+13], but it also is possible in other latent variable models. For example, consider our VAE model fit to the CelebA dataset, which has faces of celebrities and some corresponding attributes. 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 $\bar{\mathbf{z}}_i^+$ and $\bar{\mathbf{z}}_i^-$ be the average of these embeddings. We define the offset vector as $\Delta_i = \bar{\mathbf{z}}_i^+ - \bar{\mathbf{z}}_i^-$. If we add some positive multiple of Δ_i to a new point \mathbf{z} , we increase the amount of the attribute i ; if we subtract some multiple of Δ_i , we decrease the amount of the attribute i [Whi16].

We give an example of this in Figure 20.10. We consider the attribute of wearing sunglasses. 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. When $s_j > 0$ we add sunglasses to the face. When $s_j < 0$ we

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 64×64 , as is conventional.

remove sunglasses; but this also has the side effect of making the face look younger and more female, possibly a result of dataset bias.

20.3.7 Generative design

Another interesting use case for (deep) generative models is **generative design**, in which we use the model to generate candidate objects, such as molecules, which have desired properties (see e.g., [RNA22]). One approach is to fit a VAE to unlabeled samples, and then to perform Bayesian optimization (Section 6.6) in its latent space, as discussed in Section 21.3.5.2.

20.3.8 Model-based reinforcement learning

We discuss reinforcement learning (RL) in Chapter 35. The main success stories of RL to date have been in computer games, where simulators exist and data is abundant. However, in other areas, such as robotics, data is expensive to acquire. In this case, it can be useful to learn a generative “**world model**”, so the agent can do planning and learning “in its head”. See Section 35.4 for more details.

20.3.9 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 32.

20.3.10 Data compression

Models which can assign high probability to frequently occurring data vectors (e.g., images, sentences), and low probability to rare vectors, can be used for **data compression**, since we can assign shorter codes to the more common items. Indeed, the optimal coding length for a vector \mathbf{x} from some stochastic source $p(\mathbf{x})$ is $l(\mathbf{x}) = -\log p(\mathbf{x})$, as proved by Shannon. See Section 5.4 for details.

20.4 Evaluating generative models

This section is written by Mihaela Rosca, Shakir Mohamed, and Balaji Lakshminarayanan.

Evaluating generative models requires metrics which capture

- **sample quality** — are samples generated by the model a part of the data distribution?
- **sample diversity** — are samples from the model distribution capturing all modes of the data distribution?
- **generalization** — is the model generalizing beyond the training data?

There is no known metric which meets all these requirements, but various metrics have been proposed to capture different aspects of the learned distribution, some of which we discuss below.

20.4.1 Likelihood-based evaluation

A standard way to measure how close a model q is to a true distribution p is in terms of the KL divergence (Section 5.1):

$$D_{\text{KL}}(p \parallel q) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} = -\mathbb{H}(p) + \mathbb{H}_{ce}(p, q) \quad (20.2)$$

where $\mathbb{H}(p)$ is a constant, and $\mathbb{H}_{ce}(p, q)$ is the cross entropy. If we approximate $p(\mathbf{x})$ by the empirical distribution, we can evaluate the cross entropy in terms of the empirical **negative log likelihood** on the dataset:

$$\text{NLL} = -\frac{1}{N} \sum_{n=1}^N \log q(\mathbf{x}_n) \quad (20.3)$$

Usually we care about negative log likelihood on a held-out test set.⁴

20.4.1.1 Computing the log-likelihood

For models of discrete data, such as language models, it is easy to compute the (negative) log likelihood. However, it is common to measure performance using a quantity called **perplexity**, which is defined as 2^H , where $H = \text{NLL}$ is the cross entropy or negative log likelihood.

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

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x})] \quad (20.4)$$

$$= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x})] \quad (20.5)$$

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 $p(\mathbf{x})$.

20.4.1.2 Likelihood can be hard to compute

Unfortunately, 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

⁴ In some applications, we report **bits per dimension**, which is the log likelihood using log base 2, divided by the dimensionality of \mathbf{x} . To compute this metric, recall that $\log_2 L = \frac{\log_e L}{\log_e 2}$, and hence $\text{bpd} = \text{NLL} \log_e(2) \frac{1}{|\mathbf{x}|}$.

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 26), the likelihood is not even defined, so we need to find evaluation metrics that do not rely on likelihood.

20.4.1.3 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

$$q_2(\mathbf{x}) = 0.01q_0(\mathbf{x}) + 0.99q_1(\mathbf{x}) \quad (20.6)$$

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

$$\log q_2(\mathbf{x}) = \log[0.01q_0(\mathbf{x}) + 0.99q_1(\mathbf{x})] \geq \log[0.01q_0(\mathbf{x})] = \log q_0(\mathbf{x}) - 2 \quad (20.7)$$

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 poor sampler does not significantly impact the log likelihood.

Now consider a case where the model has good samples but bad likelihoods. To achieve this, suppose q is a GMM centered on the training images:

$$q(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^N \mathcal{N}(\mathbf{x} | \mathbf{x}_n, \epsilon^2 \mathbf{I}) \quad (20.8)$$

If ϵ is small enough that the Gaussian noise is imperceptible, then samples from this model will look good, since they correspond to the training set of real images. But this model will almost certainly have poor likelihood on the test set due to overfitting. (In this case we say the model has effectively just memorized the training set.)

20.4.2 Distances and divergences in feature space

Due to the challenges associated with comparing distributions in high dimensional spaces, and the desire to compare distributions in a semantically meaningful way, it is common to use domain-specific **perceptual distance metrics**, that measure how similar data vectors are to each other or to the training data. However, most metrics used to evaluate generative models do not directly compare raw data (e.g., pixels) but use a neural network to obtain features from the raw data and compare the feature distribution obtained from model samples with the feature distribution obtained from the dataset. The neural network used to obtain features can be trained solely for the purpose of evaluation, or can be pretrained; a common choice is to use a pretrained classifier (see e.g., [Sal+16; Heu+17b; Bin+18; Kyn+19; SSG18a]).

The **Inception score** [Sal+16] measures the average KL divergence between the marginal distribution of class labels obtained from the samples $p_{\theta}(y) = \int p_{\text{disc}}(y|\mathbf{x})p_{\theta}(\mathbf{x})d\mathbf{x}$ (where the integral is approximated by sampling images \mathbf{x} from a fixed dataset) and the distribution $p(y|\mathbf{x})$ induced by samples from the model, $\mathbf{x} \sim p_{\theta}(\mathbf{x})$. (The term comes from the “Inception” model [Sze+15b] that is often used to define $p_{\text{disc}}(y|\mathbf{x})$.) This leads to the following score:

$$\text{IS} = \exp [\mathbb{E}_{p_{\theta}(\mathbf{x})} D_{\text{KL}}(p_{\text{disc}}(Y|\mathbf{x}) \parallel p_{\theta}(Y))] \quad (20.9)$$

To understand this, let us rewrite the log score as follows:

$$\log(\text{IS}) = \mathbb{H}(p_{\theta}(Y)) - \mathbb{E}_{p_{\theta}(\mathbf{x})} [\mathbb{H}(p_{\text{disc}}(Y|\mathbf{x}))] \quad (20.10)$$

Thus we see that a high scoring model will be equally likely to generate samples from all classes, thus maximizing the entropy of $p_{\theta}(Y)$, while also ensuring that each individual sample is easy to classify, thus minimizing the entropy of $p_{\text{disc}}(Y|\mathbf{x})$.

The Inception score solely relies on class labels, and thus does not measure overfitting or sample diversity outside the predefined dataset classes. For example, a model which generates one perfect example per class would get a perfect Inception score, despite not capturing the variety of examples inside a class, as shown in Figure 20.11a. To address this drawback, the **Fréchet Inception distance** or **FID** score [Heu+17b] measures the Fréchet distance between two Gaussian distributions on sets of features of a pre-trained classifier. One Gaussian is obtained by passing model samples through a pretrained classifier, and the other by passing dataset samples through the same classifier. If we assume that the mean and covariance obtained from model features are μ_m and Σ_m and those from the data are μ_d and Σ_d , then the FID is

$$\text{FID} = \|\mu_m - \mu_d\|_2^2 + \text{tr}(\Sigma_d + \Sigma_m - 2(\Sigma_d \Sigma_m)^{1/2}) \quad (20.11)$$

Since it uses features instead of class logits, the Fréchet distance captures more than modes captured by class labels, as shown in Figure 20.11b. Unlike the Inception score, a lower score is better since we want the two distributions to be as close as possible.

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 2.7.3) between the features obtained from the data and features obtained from model samples.

20.4.3 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, or that the model only captures a subset of the data (e.g., in Figure 26.6). 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 26.3.3) can be an issue.)

A common approach is to use nearest neighbors in the feature space of a pretrained classifier to



Figure 20.11: (a) Model samples with good (high) inception score are visually realistic. (b) Model samples with good (low) FID score are visually realistic and diverse.

define precision and recall [Kyn+19]. To formalize this, let us define

$$f_k(\phi, \Phi) = \begin{cases} 1 & \text{if } \exists \phi' \in \Phi \text{ s.t. } \|\phi - \phi'\|_2^2 \leq \|\phi' - \text{NN}_k(\phi', \Phi)\|_2^2 \\ 0 & \text{otherwise} \end{cases} \quad (20.12)$$

where Φ is a set of feature vectors and $\text{NN}_k(\phi', \Phi)$ is a function returning the k 'th nearest neighbor of ϕ' in Φ . We now define precision and recall as follows:

$$\text{precision}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{model}|} \sum_{\phi \in \Phi_{model}} f_k(\phi, \Phi_{data}); \quad (20.13)$$

$$\text{recall}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{data}|} \sum_{\phi \in \Phi_{data}} f_k(\phi, \Phi_{model}); \quad (20.14)$$

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 is as close to model samples as model samples are to other samples. The parameter k controls how lenient the metrics will be — the higher k , the higher both precision and recall will be. As in classification, precision and recall in generative models can be used to construct a trade-off curve between different models which allows practitioners to make an informed decision regarding which model they want to use.

20.4.4 Statistical tests

Statistical tests have long been used to determine whether two sets of samples have been generated from the same distribution; these types of statistical tests are called **two sample tests**. Let us define the null hypothesis as the statement that both set of samples are from the same distribution. We then compute a statistic from the data and compare it to a threshold, and based on this we decide whether to reject the null hypothesis. In the context of evaluating implicit generative models such as GANs, statistics based on classifiers [Saj+18] and the MMD [Liu+20b] have been used. For use in scenarios with high dimensional input spaces, which are ubiquitous in the era of deep learning, two sample tests have been adapted to use learned features instead of raw data.

Like all other evaluation metrics for generative models, statistical tests have their own advantages and disadvantages: while users can specify Type 1 error — the chance they allow that the null hypothesis is wrongly rejected — statistical tests tend to be computationally expensive and thus cannot be used to monitor progress in training; hence they are best used to compare fully trained models.

20.4.5 Challenges with using pretrained classifiers

While popular and convenient, evaluation metrics that rely on pretrained classifiers (such as IS, FID, nearest neighbors in feature space, and statistical tests in feature space) have significant drawbacks. One might not have a pretrained classifier available for the dataset at hand, so classifiers trained on other datasets are used. Given the well known challenges with neural network generalization (see Section 17.4), the features of a classifier trained on images from one dataset might not be reliable enough to provide a fine grained signal of quality for samples obtained from a model trained on a different dataset. If the generative model is trained on the same dataset as the pre-trained classifier but the model is not capturing the data distribution perfectly, we are presenting the pre-trained classifier with out-of-distribution data and relying on its features to obtain score to evaluate our models. Far from being purely theoretical concerns, these issues have been studied extensively and have been shown to affect evaluation in practice [RV19; BS18].

20.4.6 Using model samples to train classifiers

Instead of using pretrained classifiers to evaluate samples, one can train a classifier on samples from conditional generative models, and then see how good these classifiers are at classifying data. For example, does adding synthetic (sampled) data to the real data help? This is closer to a reliable evaluation of generative model samples, since ultimately, the performance of generative models is dependent on the downstream task they are trained for. If used for semisupervised learning, one should assess how much adding samples to a classifier dataset helps with test accuracy. If used for model based reinforcement learning, one should assess how much the generative model helps with agent performance. For examples of this approach, see e.g., [SSM18; SSA18; RV19; SS20b; Jor+22].

20.4.7 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 20.12: 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 21), autoregressive models (Chapter 22), and normalizing flows (Chapter 23) — we can assess memorization by seeing



Figure 20.12: 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.

how much the log-likelihood of a model changes when a sample is included in the model's training set or not [BW21].

20.4.8 Human evaluation

One approach to evaluate generative models is to use human evaluation, by presenting samples from the model alongside samples from the data distribution, and ask human raters to compare the quality of the samples [Zho+19b]. Human evaluation is a suitable metric if the model is used to create art or other data for human display, or if reliable automated metrics are hard to obtain. However, human evaluation can be difficult to standardize, hard to automate, and can be expensive or cumbersome to set up.

21 Variational autoencoders

21.1 Introduction

In this chapter, we discuss generative models of the form

$$z \sim p_{\theta}(z) \tag{21.1}$$

$$x|z \sim \text{Expfam}(x|d_{\theta}(z)) \tag{21.2}$$

where $p(z)$ is some kind of prior on the latent code z , $d_{\theta}(z)$ is a deep neural network, known as the **decoder**, and $\text{Expfam}(x|\eta)$ 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 $p_{\theta}(z|x)$) is computationally intractable, as is computing the marginal likelihood

$$p_{\theta}(x) = \int p_{\theta}(x|z)p_{\theta}(z) dz \tag{21.3}$$

Hence we need to resort to approximate inference. For most of this chapter, we will use **amortized inference**, which we discussed in Section 10.1.5. This trains another model, $q_{\phi}(z|x)$, 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, discussed in Section 16.3.3.

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.

21.2 VAE basics

In this section, we discuss the basics of variational autoencoders.

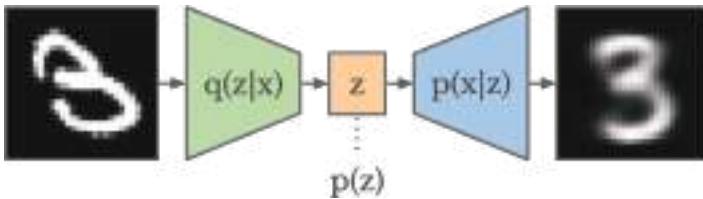


Figure 21.1: Schematic illustration of a VAE. From a figure in [Haf18]. Used with kind permission of Danijar Hafner.

21.2.1 Modeling assumptions

In the simplest setting, a VAE defines a generative model of the form

$$p_{\theta}(z, x) = p_{\theta}(z)p_{\theta}(x|z) \quad (21.4)$$

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

$$p_{\theta}(x|z) = \prod_{d=1}^D \text{Ber}(x_d | \sigma(d_{\theta}(z))) \quad (21.5)$$

In addition, a VAE fits a recognition model

$$q_{\phi}(z|x) = q(z|e_{\phi}(x)) \approx p_{\theta}(z|x) \quad (21.6)$$

to perform approximate posterior inference. Here $q_{\phi}(z|x)$ is usually a Gaussian, with parameters computed by a neural network encoder $e_{\phi}(x)$:

$$q_{\phi}(z|x) = \mathcal{N}(z|\mu, \text{diag}(\exp(\ell))) \quad (21.7)$$

$$(\mu, \ell) = e_{\phi}(x) \quad (21.8)$$

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 21.1.

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 10.1.5. 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 10.6) 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]).²

1. For example, the website <https://github.com/matthewwvowels1/Awesome-VAEs> lists over 900 papers.

2. Combining a generative model with an inference model in this way results in what has been called a “monference”,

21.2.2 Model fitting

We can fit a VAE using amortized stochastic variational inference, as we discuss in Section 10.2.1.6. For example, suppose we use a VAE with a diagonal Bernoulli likelihood model, and a full covariance Gaussian as our variational posterior. Then we can use the methods discussed in Section 10.2.1.2 to derive the fitting algorithm. See Algorithm 21.1 for the corresponding pseudocode.

Algorithm 21.1: Fitting a VAE with Bernoulli likelihood and full covariance Gaussian posterior. Based on Algorithm 2 of [KW19a].

```

1 Initialize  $\theta, \phi$ 
2 repeat
3   Sample  $x \sim p_{\mathcal{D}}$ 
4   Sample  $\epsilon \sim q_0$ 
5    $(\mu, \log \sigma, \mathbf{L}') = e_{\phi}(x)$ 
6    $\mathbf{M} = \text{np.triu}(\text{np.ones}(K), -1)$ 
7    $\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\sigma)$ 
8    $z = \mathbf{L}\epsilon + \mu$ 
9    $p_p = d_{\theta}(z)$ 
10   $\mathcal{L}_{\text{logqz}} = -\sum_{k=1}^K [\frac{1}{2}\epsilon_k^2 + \frac{1}{2}\log(2\pi) + \log \sigma_k]$  // from  $q_{\phi}(z|x)$  in Equation (10.47)
11   $\mathcal{L}_{\text{logpz}} = -\sum_{k=1}^K [\frac{1}{2}z_k^2 + \frac{1}{2}\log(2\pi)]$  // from  $p_{\theta}(z)$  in Equation (10.48)
12   $\mathcal{L}_{\text{logpx}} = -\sum_{d=1}^D [x_d \log p_d + (1-x_d) \log(1-p_d)]$  // from  $p_{\theta}(x|z)$ 
13   $\mathcal{L} = \mathcal{L}_{\text{logpx}} + \mathcal{L}_{\text{logpz}} - \mathcal{L}_{\text{logqz}}$ 
14  Update  $\theta := \theta - \eta \nabla_{\theta} \mathcal{L}$ 
15  Update  $\phi := \phi - \eta \nabla_{\phi} \mathcal{L}$ 
16 until converged

```

21.2.3 Comparison of VAEs and autoencoders

VAEs are very similar to deterministic autoencoders (AE). There are 2 main differences: in the AE, the objective is the log likelihood of the reconstruction without any KL term; and in addition, the encoding is deterministic, so the encoder network just needs to compute $\mathbb{E}[z|x]$ and not $\mathbb{V}[z|x]$. In view of these similarities, one can use the same codebase to implement both methods. However, it is natural to wonder what the benefits and potential drawbacks of the VAE are compared to the deterministic AE.

We shall answer this question by fitting both models to the CelebA dataset. Both models have the same convolutional structure with the following number of hidden channels per convolutional layer in the encoder: (32, 64, 128, 256, 512). The spatial size of each layer is as follows: (32, 16, 8, 4, 2). The final $2 \times 2 \times 512$ convolutional layer then gets reshaped and passed through a linear layer to generate the mean and (marginal) variance of the stochastic latent vector, which has size 256. The structure

i.e., model-inference hybrid. See the blog by Jacob Andreas, <http://blog.jacobandreas.net/monference.html>, for further discussion.

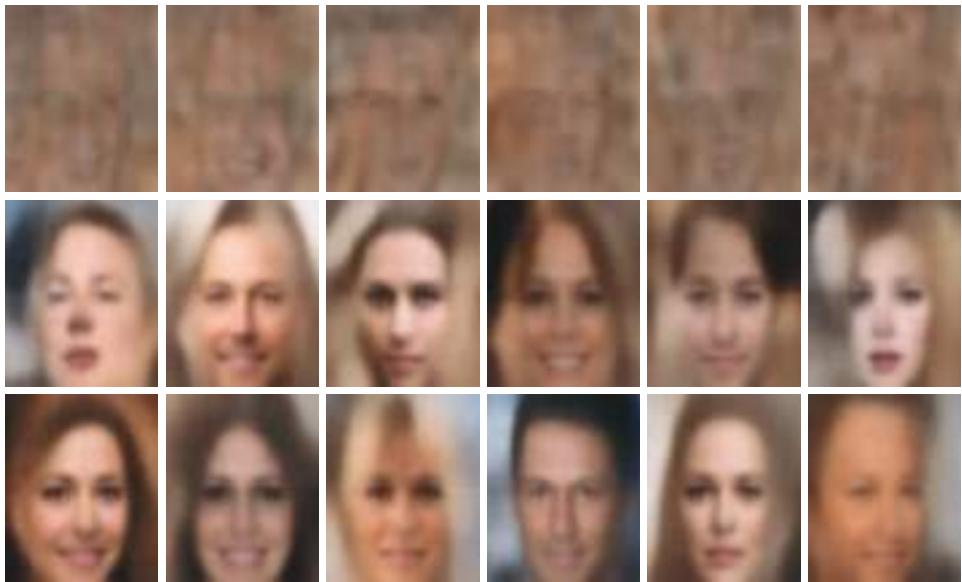


Figure 21.2: Illustration of unconditional image generation using (V)AEs trained on CelebA. Row 1: deterministic autoencoder. Row 2: β -VAE with $\beta = 0.5$. Row 3: VAE (with $\beta = 1$). Generated by [celeba_vae_ae_comparison.ipynb](#).

of the decoder is the mirror image of the encoder. Each model is trained for 5 epochs with a batch size of 256, which takes about 20 minutes on a GPU.

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, so does poorly when fed random inputs. This is illustrated in Figure 21.2.

We can also use both models to reconstruct a given input image. In Figure 21.3, we see that both AE and VAE can reconstruct the input images reasonably well, although the VAE reconstructions are somewhat blurry, for reasons we discuss in Section 21.3.1. We can reduce the amount of blurriness by scaling down the KL penalty term by a factor of β ; this is known as the β -VAE, and is discussed in more detail in Section 21.3.1.

21.2.4 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

$$p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{z})p_{\theta}(\mathbf{x}|\mathbf{z}) \quad (21.9)$$



Figure 21.3: Illustration of image reconstruction using (V)AEs trained and applied to CelebA. Row 1: original images. Row 2: deterministic autoencoder. Row 3: β -VAE with $\beta = 0.5$. Row 4: VAE (with $\beta = 1$). Generated by [celeba_vae_ae_comparison.ipynb](#).

from which we can derive the generative data marginal

$$p_{\theta}(\mathbf{x}) = \int_{\mathbf{z}} p_{\theta}(\mathbf{x}, \mathbf{z}) d\mathbf{z} \quad (21.10)$$

and the generative posterior

$$p_{\theta}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{x}, \mathbf{z}) / p_{\theta}(\mathbf{x}) \quad (21.11)$$

Let us also define the joint *inference* distribution

$$q_{\mathcal{D}, \phi}(\mathbf{z}, \mathbf{x}) = p_{\mathcal{D}}(\mathbf{x}) q_{\phi}(\mathbf{z}|\mathbf{x}) \quad (21.12)$$

where

$$p_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^N \delta(\mathbf{x}_n - \mathbf{x}) \quad (21.13)$$

is the empirical distribution. From this we can derive the inference latent marginal, also called the aggregated posterior:

$$q_{\mathcal{D}, \phi}(\mathbf{z}) = \int_{\mathbf{x}} q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z}) d\mathbf{x} \quad (21.14)$$

and the inference likelihood

$$q_{\mathcal{D}, \phi}(\mathbf{x}|\mathbf{z}) = q_{\mathcal{D}, \phi}(\mathbf{x}, \mathbf{z}) / q_{\mathcal{D}, \phi}(\mathbf{z}) \quad (21.15)$$

See Figure 21.4 for a visual illustration.

Having defined our terms, we can now derive various alternative versions of the ELBO, following [ZSE19]. First note that the ELBO averaged over all the data is given by

$$\mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathcal{D}) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})]] - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{z}))] \quad (21.16)$$

$$= \mathbb{E}_{q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) + \log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] \quad (21.17)$$

$$= \mathbb{E}_{q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})} + \log p_{\mathcal{D}}(\mathbf{x}) \right] \quad (21.18)$$

$$= -D_{\text{KL}}(q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})) + \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\log p_{\mathcal{D}}(\mathbf{x})] \quad (21.19)$$

If we define $\stackrel{c}{=}$ to mean equal up to additive constants, we can rewrite the above as

$$\mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathcal{D}) \stackrel{c}{=} -D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})) \quad (21.20)$$

$$\stackrel{c}{=} -D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))] \quad (21.21)$$

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

$$\mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathcal{D}) \stackrel{c}{=} -D_{\text{KL}}(q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z})) - \mathbb{E}_{q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{z})} [D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}|\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}))] \quad (21.22)$$

We see from Equation (21.22) that VAEs are trying to minimize the difference between the inference marginal and generative prior, $D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z}))$, while simultaneously minimizing reconstruction error, $D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}|\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}))$. 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).

21.3 VAE generalizations

In this section, we discuss some variants of the basic VAE model.

21.3. VAE generalizations

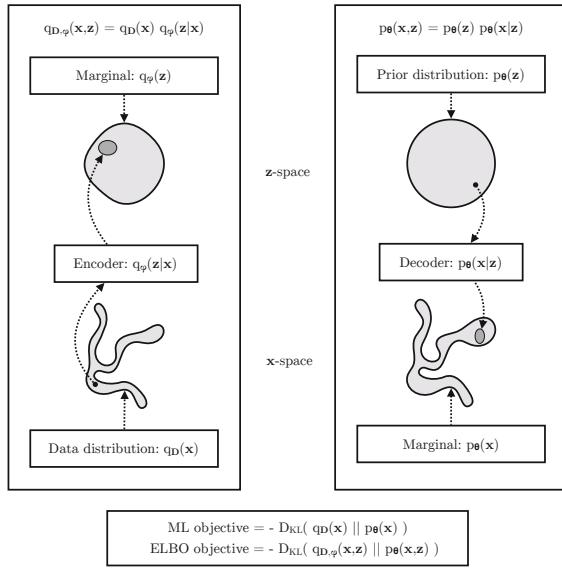


Figure 21.4: The maximum likelihood (ML) objective can be viewed as the minimization of $D_{KL}(p_D(x) \parallel p_\theta(x))$. (Note: in the figure, $p_D(x)$ is denoted by $q_D(x)$.) The ELBO objective is minimization of $D_{KL}(q_{D,\phi}(x,z) \parallel p_\theta(x,z))$, which upper bounds $D_{KL}(p_D(x) \parallel p_\theta(x))$. From Figure 2.4 of [KW19a]. Used with kind permission of Durk Kingma.

21.3.1 β -VAE

It is often the case that VAEs generate somewhat blurry images, as illustrated in Figure 21.3, Figure 21.2 and Figure 20.9. This is not the case for models that optimize the exact likelihood, such as pixelCNNs (Section 22.3.2) and flow models (Chapter 23). To see why VAEs are different, consider the common case where the decoder is a Gaussian with fixed variance, so

$$\log p_\theta(x|z) = -\frac{1}{2\sigma^2} \|x - d_\theta(z)\|_2^2 + \text{const} \quad (21.23)$$

Let $e_\phi(x) = \mathbb{E}[q_\phi(z|x)]$ be the encoding of x , and $\mathcal{X}(z) = \{x : e_\phi(x) = z\}$ be the set of inputs that get mapped to z . For a fixed inference network, the optimal setting of the generator parameters, when using squared reconstruction loss, is to ensure $d_\theta(z) = \mathbb{E}[x : x \in \mathcal{X}(z)]$. Thus the decoder should predict the average of all inputs x that map to that z , resulting in blurry images.

We can solve this problem by increasing the expressive power of the posterior approximation (avoiding the merging of distinct inputs into the same latent code), or of the generator (by adding back information that is missing from the latent code), or both. However, an even simpler solution is to reduce the penalty on the KL term, making the model closer to a deterministic autoencoder:

$$\mathcal{L}_\beta(\theta, \phi|x) = \underbrace{-\mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)]}_{\mathcal{L}_E} + \beta \underbrace{D_{KL}(q_\phi(z|x) \parallel p_\theta(z))}_{\mathcal{L}_R} \quad (21.24)$$

where \mathcal{L}_E is the reconstruction error (negative log likelihood), and \mathcal{L}_R is the KL regularizer. This is

called the β -VAE objective [Hig+17a]. 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). By using $\beta < 1$, we store more bits about each input, and hence can reconstruct images in a less blurry way. If we use $\beta > 1$, we get a more compressed representation.

21.3.1.1 Disentangled representations

One advantage of using $\beta > 1$ is that it encourages the learning of a latent representation that is “**disentangled**”. Intuitively this means that each latent dimension represents a different **factor of variation** in the input. This is often formalized in terms of the total correlation (Section 5.3.5.1), which is defined as follows:

$$\text{TC}(\mathbf{z}) = \sum_k \mathbb{H}(z_k) - \mathbb{H}(\mathbf{z}) = D_{\text{KL}} \left(p(\mathbf{z}) \parallel \prod_k p_k(z_k) \right) \quad (21.25)$$

This is zero iff the components of \mathbf{z} are all mutually independent, and hence disentangled. In [AS18], they prove that using $\beta > 1$ will decrease the TC.

Unfortunately, in [Loc+18] they prove that nonlinear latent variable models are unidentifiable, and therefore for any disentangled representation, there is an equivalent fully entangled representation with exactly the same likelihood. Thus it is not possible to recover the correct latent representation without choosing the appropriate inductive bias, via the encoder, decoder, prior, dataset, or learning algorithm, i.e., merely adjusting β is not sufficient. See Section 32.4.1 for more discussion.

21.3.1.2 Connection with information bottleneck

In this section, we show that the β -VAE is an unsupervised version of the information bottleneck (IB) objective from Section 5.6. If the input is \mathbf{x} , the hidden bottleneck is \mathbf{z} , and the target outputs are $\tilde{\mathbf{x}}$, then the unsupervised IB objective becomes

$$\mathcal{L}_{\text{UIB}} = \beta \mathbb{I}(\mathbf{z}; \mathbf{x}) - \mathbb{I}(\mathbf{z}; \tilde{\mathbf{x}}) \quad (21.26)$$

$$= \beta \mathbb{E}_{p(\mathbf{x}, \mathbf{z})} \left[\log \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{x})p(\mathbf{z})} \right] - \mathbb{E}_{p(\mathbf{z}, \tilde{\mathbf{x}})} \left[\log \frac{p(\mathbf{z}, \tilde{\mathbf{x}})}{p(\mathbf{z})p(\tilde{\mathbf{x}})} \right] \quad (21.27)$$

where

$$p(\mathbf{x}, \mathbf{z}) = p_{\mathcal{D}}(\mathbf{x})p(\mathbf{z}|\mathbf{x}) \quad (21.28)$$

$$p(\mathbf{z}, \tilde{\mathbf{x}}) = \int p_{\mathcal{D}}(\mathbf{x})p(\mathbf{z}|\mathbf{x})p(\tilde{\mathbf{x}}|\mathbf{z})d\mathbf{x} \quad (21.29)$$

Intuitively, the objective in Equation (21.26) means we should pick a representation \mathbf{z} that can predict $\tilde{\mathbf{x}}$ reliably, while not memorizing too much information about the input \mathbf{x} . The tradeoff parameter is controlled by β .

From Equation (5.181), we have the following variational upper bound on this unsupervised objective:

$$\mathcal{L}_{\text{UVIB}} = -\mathbb{E}_{q_{\mathcal{D}, \phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})] + \beta \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel p_{\theta}(\mathbf{z}))] \quad (21.30)$$

which matches Equation (21.24) when averaged over \mathbf{x} .

21.3.2 InfoVAE

In Section 21.2.4, we discussed some drawbacks of the standard ELBO objective for training VAEs, namely the tendency to ignore the latent code when the decoder is powerful (Section 21.4), and the tendency to learn a poor posterior approximation due to the mismatch between the KL terms in data space and latent space (Section 21.2.4). We can fix these problems to some degree by using a generalized objective of the following form:

$$\mathcal{L}(\theta, \phi|\mathbf{x}) = -\lambda D_{\text{KL}}(q_{\phi}(\mathbf{z}) \parallel p_{\theta}(\mathbf{z})) - \mathbb{E}_{q_{\phi}(\mathbf{z})} [D_{\text{KL}}(q_{\phi}(\mathbf{x}|\mathbf{z}) \parallel p_{\theta}(\mathbf{x}|\mathbf{z}))] + \alpha \mathbb{I}_q(\mathbf{x}; \mathbf{z}) \quad (21.31)$$

where $\alpha \geq 0$ controls how 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 (21.22).

Unfortunately, the objective in Equation (21.31) cannot be computed as written, because of the intractable MI term:

$$\mathbb{I}_q(\mathbf{x}; \mathbf{z}) = \mathbb{E}_{q_{\phi}(\mathbf{x}, \mathbf{z})} \left[\log \frac{q_{\phi}(\mathbf{x}, \mathbf{z})}{q_{\phi}(\mathbf{x})q_{\phi}(\mathbf{z})} \right] = -\mathbb{E}_{q_{\phi}(\mathbf{x}, \mathbf{z})} \left[\log \frac{q_{\phi}(\mathbf{z})}{q_{\phi}(\mathbf{z}|\mathbf{x})} \right] \quad (21.32)$$

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:

$$\mathcal{L} = \mathbb{E}_{q_{\phi}(\mathbf{x}, \mathbf{z})} \left[-\lambda \log \frac{q_{\phi}(\mathbf{z})}{p_{\theta}(\mathbf{z})} - \log \frac{q_{\phi}(\mathbf{x}|\mathbf{z})}{p_{\theta}(\mathbf{x}|\mathbf{z})} - \alpha \log \frac{q_{\phi}(\mathbf{z})}{q_{\phi}(\mathbf{z}|\mathbf{x})} \right] \quad (21.33)$$

$$= \mathbb{E}_{q_{\phi}(\mathbf{x}, \mathbf{z})} \left[\log p_{\theta}(\mathbf{x}|\mathbf{z}) - \log \frac{q_{\phi}(\mathbf{z})^{\lambda+\alpha-1} p_{\mathcal{D}}(\mathbf{x})}{p_{\theta}(\mathbf{z})^{\lambda} q_{\phi}(\mathbf{z}|\mathbf{x})^{\alpha-1}} \right] \quad (21.34)$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})]] - (1-\alpha) \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel p_{\theta}(\mathbf{z}))] \\ - (\alpha + \lambda - 1) D_{\text{KL}}(q_{\phi}(\mathbf{z}) \parallel p_{\theta}(\mathbf{z})) - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\log p_{\mathcal{D}}(\mathbf{x})] \quad (21.35)$$

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 26).

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 $q_{\phi}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{z}|\mathbf{x})$.

21.3.2.1 Connection with MMD VAE

If we set $\alpha = 1$, the InfoVAE objective simplifies to

$$\hat{L} \stackrel{c}{=} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})]] - \lambda D_{\text{KL}}(q_{\phi}(\mathbf{z}) \parallel p_{\theta}(\mathbf{z})) \quad (21.36)$$

The **MMD VAE**³ replaces the KL divergence in the above term with the (squared) maximum mean discrepancy or **MMD** divergence defined in Section 2.7.3. (This is valid based on the above theorem.) The advantage of this approach over standard InfoVAE is that the resulting objective is tractable. In particular, if we set $\lambda = 1$ and swap the sign we get

$$\mathcal{L} = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})} [-\log p_{\theta}(\mathbf{x}|\mathbf{z})]] + \text{MMD}(q_{\phi}(\mathbf{z}), p_{\theta}(\mathbf{z})) \quad (21.37)$$

As we discuss in Section 2.7.3, we can compute the MMD as follows:

$$\text{MMD}(p, q) = \mathbb{E}_{p(\mathbf{z}), p(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] + \mathbb{E}_{q(\mathbf{z}), q(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] - 2\mathbb{E}_{p(\mathbf{z}), q(\mathbf{z}')} [\mathcal{K}(\mathbf{z}, \mathbf{z}')] \quad (21.38)$$

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 the MMD measures the similarity (in latent space) between samples from the prior and samples from the aggregated posterior.

In practice, we can implement the MMD objective by using the posterior predicted mean $\mathbf{z}_n = e_{\phi}(\mathbf{x}_n)$ for all B samples in the current minibatch, and comparing this to B random samples from the $\mathcal{N}(\mathbf{0}, \mathbf{I})$ prior.

If we use a Gaussian decoder with fixed variance, the negative log likelihood is just a squared error term:

$$-\log p_{\theta}(\mathbf{x}|\mathbf{z}) = \|\mathbf{x} - d_{\theta}(\mathbf{z})\|_2^2 \quad (21.39)$$

Thus the entire model is deterministic, and just predicts the means in latent space and visible space.

21.3.2.2 Connection with β -VAEs

If we set $\alpha = 0$ and $\lambda = 1$, we get back the original ELBO. If $\lambda > 0$ is freely chosen, but we use $\alpha = 1 - \lambda$, we get the β -VAE.

21.3.2.3 Connection with adversarial autoencoders

If we set $\alpha = 1$ and $\lambda = 1$, and D is chosen to be the Jensen-Shannon divergence (which can be minimized by training a binary discriminator, as explained in Section 26.2.2), then we get a model known as an **adversarial autoencoder** [Mak+15a].

21.3.3 Multimodal VAEs

It is possible to extend VAEs to create joint distributions over different kinds of variables, such as images and text. This is sometimes called a **multimodal VAE** or **MVAE**. Let us assume there are

3. Proposed in <https://ermongroup.github.io/blog/a-tutorial-on-mmd-variational-autoencoders/>.

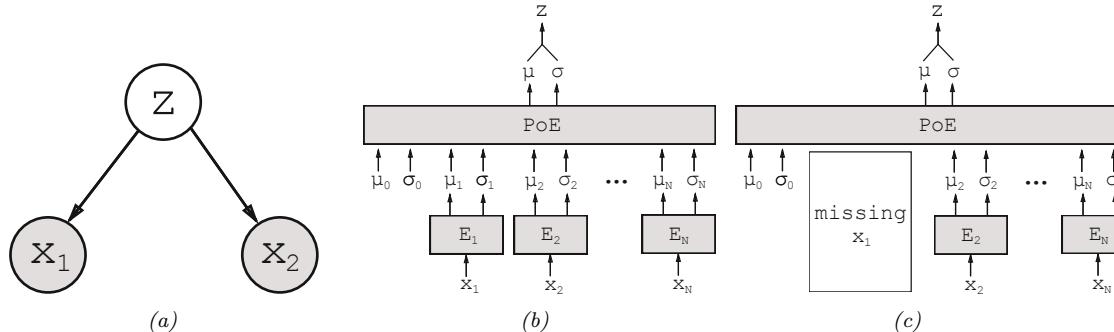


Figure 21.5: Illustration of multi-modal VAE. (a) The generative model with $N = 2$ modalities. (b) The product of experts (PoE) inference network is derived from N individual Gaussian experts E_i . μ_0 and σ_0 are parameters of the prior. (c) If a modality is missing, we omit its contribution to the 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

$$p_{\theta}(\mathbf{x}_1, \dots, \mathbf{x}_M, \mathbf{z}) = p(\mathbf{z}) \prod_{m=1}^M p_{\theta}(\mathbf{x}_m | \mathbf{z}) \quad (21.40)$$

where we treat $p(\mathbf{z})$ as a fixed prior. See Figure 21.5(a) for an illustration.

The standard ELBO is given by

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X})} \left[\sum_m \log p_{\theta}(\mathbf{x}_m | \mathbf{z}) \right] - D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X}) \| p(\mathbf{z})) \quad (21.41)$$

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:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X})} \left[\sum_m \lambda_m \log p_{\theta}(\mathbf{x}_m | \mathbf{z}) \right] - \beta D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X}) \| p(\mathbf{z})) \quad (21.42)$$

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 $\mathbf{X} = \{\mathbf{x}_m : O_m = 1\}$ be the visible features. We now use the following objective:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X})} \left[\sum_{m:O_m=1} \lambda_m \log p_{\theta}(\mathbf{x}_m | \mathbf{z}) \right] - \beta D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{X}) \| p(\mathbf{z})) \quad (21.43)$$

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.)

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

$$p(\mathbf{z}|\mathbf{X}) = \frac{p(\mathbf{z})p(\mathbf{x}_1, \dots, \mathbf{x}_M|\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} = \frac{p(\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} \prod_{m=1}^M p(\mathbf{x}_m|\mathbf{z}) \quad (21.44)$$

$$= \frac{p(\mathbf{z})}{p(\mathbf{x}_1, \dots, \mathbf{x}_M)} \prod_{m=1}^M \frac{p(\mathbf{z}|\mathbf{x}_m)p(\mathbf{x}_m)}{p(\mathbf{z})} \quad (21.45)$$

$$\propto p(\mathbf{z}) \prod_{m=1}^M \frac{p(\mathbf{z}|\mathbf{x}_m)}{p(\mathbf{z})} \approx p(\mathbf{z}) \prod_{m=1}^M \tilde{q}(\mathbf{z}|\mathbf{x}_m) \quad (21.46)$$

This can be viewed as a product of experts (Section 24.1.1), 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 (2.154):

$$\prod_{m=0}^M \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1}) \propto \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \boldsymbol{\Sigma} = (\sum_m \boldsymbol{\Lambda}_m)^{-1}, \quad \boldsymbol{\mu} = \boldsymbol{\Sigma}(\sum_m \boldsymbol{\Lambda}_m \boldsymbol{\mu}_m) \quad (21.47)$$

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. See Figure 21.5(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 \mathcal{J} randomly chosen joint modalities:

$$\mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}) = \mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi}|(\mathbf{x}_1, \dots, \mathbf{x}_M)) + \sum_{m=1}^M \mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{x}_m) + \sum_{j \in \mathcal{J}} \mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}_j) \quad (21.48)$$

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 21.5(c) for an illustration.

Note that the above scheme can only handle the case of a fixed number of missingness patterns; we can generalize to allow for arbitrary missingness as discussed in [CNW20]. (See also Section 3.11 for a more general discussion of missing data.)

21.3.4 Semisupervised 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+14a].

The generative model has the following form:

$$p_{\theta}(\mathbf{x}, y) = p_{\theta}(y)p_{\theta}(\mathbf{x}|y) = p_{\theta}(y) \int p_{\theta}(\mathbf{x}|y, \mathbf{z})p_{\theta}(\mathbf{z})d\mathbf{z} \quad (21.49)$$

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

$$q_{\phi}(\mathbf{z}|y, \mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\phi}(y, \mathbf{x}), \text{diag}(\boldsymbol{\sigma}_{\phi}(y, \mathbf{x}))) \quad (21.50)$$

We then use the following variational lower bound

$$\log p_{\theta}(\mathbf{x}, y) \geq \mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x}, y)} [\log p_{\theta}(\mathbf{x}|y, \mathbf{z}) + \log p_{\theta}(y) + \log p_{\theta}(\mathbf{z}) - \log q_{\phi}(\mathbf{z}|\mathbf{x}, y)] = -\mathcal{L}(\mathbf{x}, y) \quad (21.51)$$

as is standard for VAEs (see Section 21.2). 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

$$q_{\phi}(\mathbf{z}, y|\mathbf{x}) = q_{\phi}(\mathbf{z}|\mathbf{x})q_{\phi}(y|\mathbf{x}) \quad (21.52)$$

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\phi}(\mathbf{x}), \text{diag}(\boldsymbol{\sigma}_{\phi}(\mathbf{x}))) \quad (21.53)$$

$$q_{\phi}(y|\mathbf{x}) = \text{Cat}(y|\boldsymbol{\pi}_{\phi}(\mathbf{x})) \quad (21.54)$$

Note that $q_{\phi}(y|\mathbf{x})$ acts like a discriminative classifier, that imputes the missing labels. We then use the following variational lower bound:

$$\log p_{\theta}(\mathbf{x}) \geq \mathbb{E}_{q_{\phi}(\mathbf{z}, y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|y, \mathbf{z}) + \log p_{\theta}(y) + \log p_{\theta}(\mathbf{z}) - \log q_{\phi}(\mathbf{z}, y|\mathbf{x})] \quad (21.55)$$

$$= - \sum_y q_{\phi}(y|\mathbf{x}) \mathcal{L}(\mathbf{x}, y) + \mathbb{H}(q_{\phi}(y|\mathbf{x})) = -\mathcal{U}(\mathbf{x}) \quad (21.56)$$

Note that the discriminative classifier $q_{\phi}(y|\mathbf{x})$ is only used to compute the log-likelihood of the unlabeled data, which is undesirable. We can therefore add an extra classification loss on the supervised data, to get the following overall objective function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}_L} [\mathcal{L}(\mathbf{x}, y)] + \mathbb{E}_{\mathbf{x} \sim \mathcal{D}_U} [\mathcal{U}(\mathbf{x})] + \alpha \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}_L} [-\log q_{\phi}(y|\mathbf{x})] \quad (21.57)$$

where \mathcal{D}_L is the labeled data, \mathcal{D}_U is the unlabeled data, and α is a hyperparameter that controls the relative weight of generative and discriminative learning.

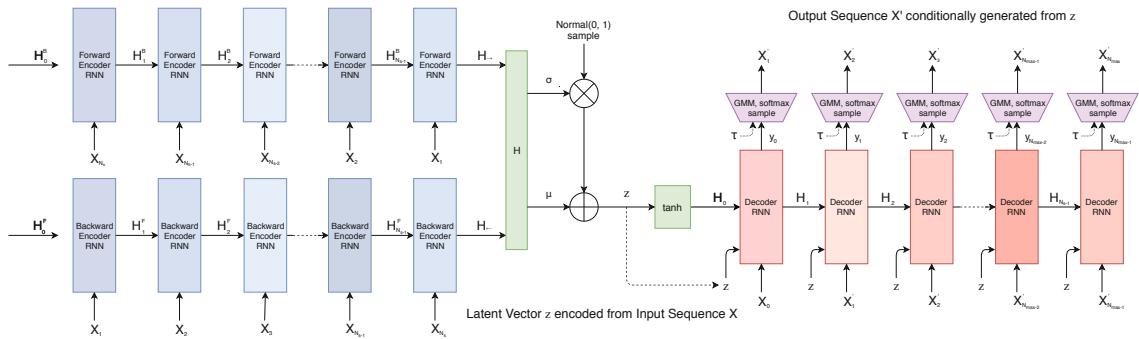


Figure 21.6: Illustration of a VAE with a bidirectional RNN encoder and a unidirectional RNN decoder. The output generator can use a GMM and/or softmax distribution. From Figure 2 of [HE18]. Used with kind permission of David Ha.

21.3.5 VAEs with sequential encoders/decoders

In this section, we discuss VAEs for sequential data, such as text and biosequences, in which the data \mathbf{x} is a variable-length sequence, but we have a fixed-sized latent variable $\mathbf{z} \in \mathbb{R}^K$. (We consider the more general case in which \mathbf{z} is a variable-length sequence of latents — known as **sequential VAE** or **dynamic VAE** — in Section 29.13.) All we have to do is modify the decoder $p(\mathbf{x}|\mathbf{z})$ and encoder $q(\mathbf{z}|\mathbf{x})$ to work with sequences.

21.3.5.1 Models

If we use an RNN for the encoder and decoder of a VAE, we get a model which is called a **VAE-RNN**, as proposed in [Bow+16a]. In more detail, the generative model is $p(\mathbf{z}, \mathbf{x}_{1:T}) = p(\mathbf{z})\text{RNN}(\mathbf{x}_{1:T}|\mathbf{z})$, where \mathbf{z} can be injected as the initial state of the RNN, or as an input to every time step. The inference model is $q(\mathbf{z}|\mathbf{x}_{1:T}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}(\mathbf{h}), \boldsymbol{\Sigma}(\mathbf{h}))$, where $\mathbf{h} = [\mathbf{h}_T^\rightarrow, \mathbf{h}_1^\leftarrow]$ is the output of a bidirectional RNN applied to $\mathbf{x}_{1:T}$. See Figure 21.6 for an illustration.

More recently, people have tried to combine transformers with VAEs. For example, in the **Optimus** model of [Li+20], they use a BERT model for the encoder. In more detail, the encoder $q(\mathbf{z}|\mathbf{x})$ is derived from the embedding vector associated with a dummy token corresponding to the “class label” which is appended to the input sequence \mathbf{x} . The decoder is a standard autoregressive model (similar to GPT), with one additional input, namely the latent vector \mathbf{z} . They consider two ways of injecting the latent vector. The simplest approach is to add \mathbf{z} to the embedding layer of every token in the 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 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 get better results in their experiments by letting all the layers of the decoder attend to the latent code \mathbf{z} . An easy way to do this is to define the memory vector $\mathbf{h}_m = \mathbf{W}\mathbf{z}$, where $\mathbf{W} \in \mathbb{R}^{LH \times K}$, 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 embeddings at each layer.

An alternative approach, known as **transformer VAE**, was proposed in [Gre20]. This model uses a **funnel transformer** [Dai+20b] as the encoder, and the **T5** [Raf+20a] conditional transformer for

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 21.7: (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.

the decoder. In addition, it uses an MMD VAE (Section 21.3.2.1) to avoid posterior collapse.

21.3.5.2 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 20.3.5. The results of doing this with the VAE-RNN are illustrated in Figure 21.7a. (Similar results are shown in [Li+20], 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 21.7b. The reason is that the deterministic autoencoder has “holes” in its latent space, which get decoded to nonsensical outputs.

However, because RNNs (and transformers) are powerful decoders, we need to address the problem of posterior collapse, which we discuss in Section 21.4. 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 21.3.2, 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.

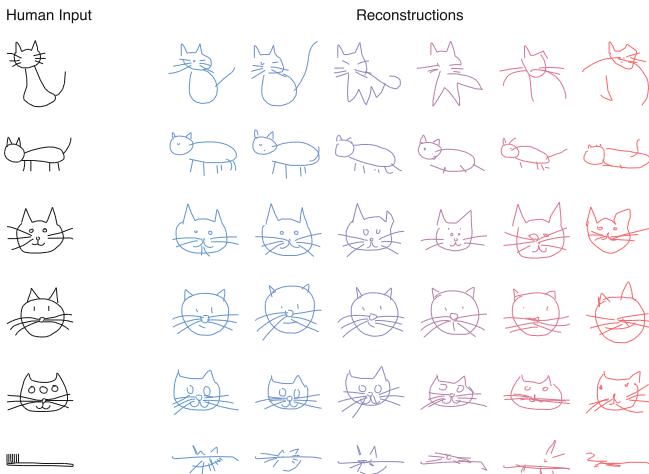


Figure 21.8: Conditional generation of cats from sketch-RNN model. We increase the temperature parameter from left to right. From Figure 5 of [HE18]. Used with kind permission of David Ha.

Figure 21.8 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.

Molecular design

In [GB+18], they use VAE-RNNs to model molecular graph structure, represented as a string using the SMILES representation.⁴ 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 21.9 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.

21.4 Avoiding posterior collapse

If the decoder $p_{\theta}(x|z)$ is sufficiently powerful (e.g., a pixel CNN, or an RNN for text), then the VAE does not need to use the latent code z for anything. This is called **posterior collapse** or **variational**

4. See https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system.

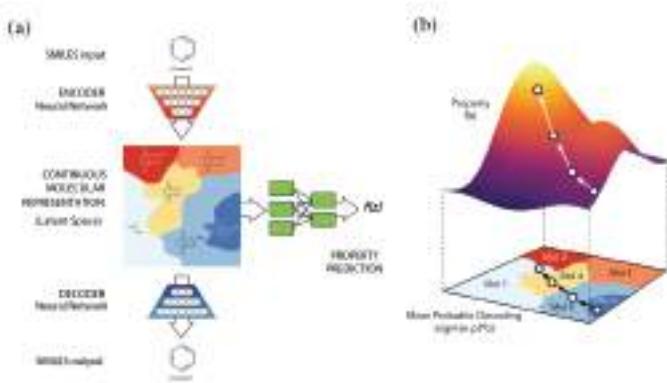


Figure 21.9: Application of VAE-RNN to molecule design. (a) The VAE-RNN model is trained on a sequence representation of molecules known as SMILES. We can fit an MLP to map from the latent space to properties of the molecule, such as its “fitness” $f(\mathbf{z})$. (b) We can perform gradient ascent in $f(\mathbf{z})$ 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.

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 (21.21). 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 $D_{\text{KL}}(p_D(\mathbf{x}) \parallel p_{\theta}(\mathbf{x})) = 0$. 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 $D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel p_{\theta}(\mathbf{z}|\mathbf{x})) = 0$. Thus we have successfully maximized the ELBO, but we have not learned any useful latent representation of the data, which is one of the goals of latent variable modeling.⁵ We discuss some solutions to posterior collapse below.

21.4.1 KL annealing

A common approach to solving this problem, proposed in [Bow+16a], is to use **KL annealing**, in which the KL penalty term in the ELBO is scaled by β , which is increased from 0.0 (corresponding to an autoencoder) to 1.0 (which corresponds to standard MLE training). (Note that, by contrast, the β -VAE model in Section 21.3.1 uses $\beta > 1$.)

KL annealing can work well, but requires tuning the schedule for β . A standard practice [Fu+19] is to use **cyclical annealing**, which repeats the process of increasing β multiple times. This ensures the progressive learning of more meaningful latent codes, by leveraging good representations learned in a previous cycle as a way to warmstart the optimization.

5. Note that [Luc+19; DWW20] show that posterior collapse can also happen in linear VAE models, where the ELBO corresponds to the exact marginal likelihood, so the problem is not only due to powerful (nonlinear) decoders, but is also related to spurious local maxima in the objective.

21.4.2 Lower bounding the rate

An alternative approach is to stick with the original unmodified ELBO objective, but to prevent the rate (i.e., the $D_{\text{KL}}(q \parallel p)$ term) from collapsing to 0, by limiting the flexibility of q . For example, [XD18; Dav+18] use a von Mises-Fisher (Section 2.2.5.3) prior and posterior, instead of a Gaussian, 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 the parameter κ controls the rate of the code. The δ -VAE method [Oor+19] uses a Gaussian autoregressive prior and a diagonal Gaussian posterior. We can ensure the rate is at least δ by adjusting the regression parameter of the AR prior.

21.4.3 Free bits

In this section, we discuss the method of **free bits** [Kin+16], which is another way of lower bounding the rate. To explain this, consider a fully factorized posterior in which the KL penalty has the form

$$\mathcal{L}_R = \sum_i D_{\text{KL}}(q_{\phi}(z_i|\mathbf{x}) \parallel p_{\theta}(z_i)) \quad (21.58)$$

where z_i is the i 'th dimension of \mathbf{z} . We can replace this with a hinge loss, that will give up driving down the KL for dimensions that are already beneath a target compression rate λ :

$$\mathcal{L}'_R = \sum_i \max(\lambda, D_{\text{KL}}(q_{\phi}(z_i|\mathbf{x}) \parallel p_{\theta}(z_i))) \quad (21.59)$$

Thus the bits where the KL is sufficiently small “are free”, since the model does not have to “pay” to encode them according to the prior.

21.4.4 Adding skip connections

One reason for latent variable collapse is that the latent variables \mathbf{z} are not sufficiently “connected to” the observed data \mathbf{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 21.10. This is called a **skip-VAE** [Die+19a].

21.4.5 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 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 that are tighter than the vanilla ELBO (Section 10.5.1), or more accurate posterior approximations (Section 10.4), or more accurate (hierarchical) generative models (Section 21.5).

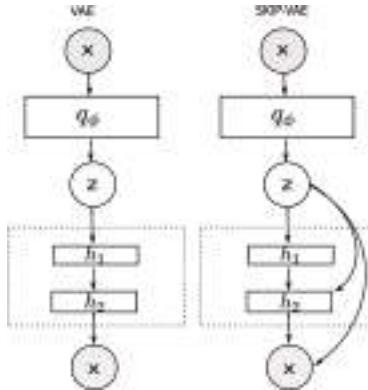


Figure 21.10: (a) VAE. (b) Skip-VAE. From Figure 1 of [Die+19a]. Used with kind permission of Adji Dieng.

21.4.6 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 21.3.2, which includes adversarial autoencoders and MMD autoencoders as special cases. The InfoVAE objective includes a term to explicitly enforce non-zero mutual information between \mathbf{x} and \mathbf{z} , which effectively solves the problem of posterior collapse.

21.5 VAEs with hierarchical structure

We define a **hierarchical VAE** or HVAE, with L stochastic layers, to be the following generative model:⁶

$$p_{\theta}(\mathbf{x}, \mathbf{z}_{1:L}) = p_{\theta}(\mathbf{z}_L) \left[\prod_{l=L-1}^1 p_{\theta}(\mathbf{z}_l | \mathbf{z}_{l+1}) \right] p_{\theta}(\mathbf{x} | \mathbf{z}_1) \quad (21.60)$$

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.,

$$p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{z}_L) \left[\prod_{l=L-1}^1 p_{\theta}(\mathbf{z}_l | \mathbf{z}_{l+1:L}) \right] p_{\theta}(\mathbf{x} | \mathbf{z}_{1:L}) \quad (21.61)$$

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 21.11(left). We discuss how to perform inference and learning in such models below.

6. 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”.

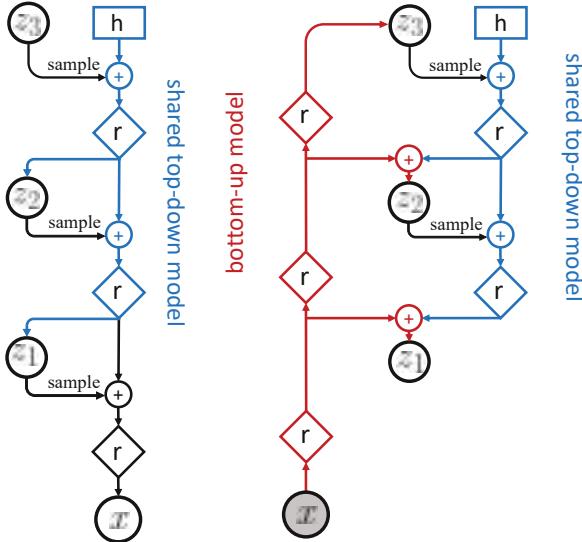


Figure 21.11: Hierarchical VAEs with 3 stochastic layers. Left: generative model. Right: inference network. Diamond is a residual network, \oplus is feature combination (e.g., concatenation), and h is a trainable parameter. We first do bottom-up inference, by propagating \mathbf{x} up to \mathbf{z}_3 to compute $\mathbf{z}_3^s \sim q_\phi(\mathbf{z}_3|\mathbf{x})$, and then we perform top-down inference by computing $\mathbf{z}_2^s \sim q_\phi(\mathbf{z}_2|\mathbf{x}, \mathbf{z}_3^s)$ and then $\mathbf{z}_1^s \sim q_\phi(\mathbf{z}_1|\mathbf{x}, \mathbf{z}_{2:3}^s)$. From Figure 2 of [VK20a]. Used with kind permission of Arash Vahdat.

21.5.1 Bottom-up vs top-down inference

To perform inference in a hierarchical VAE, we could use a **bottom-up inference model** of the form

$$q_\phi(\mathbf{z}|\mathbf{x}) = q_\phi(\mathbf{z}_1|\mathbf{x}) \prod_{l=2}^L q_\phi(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{1:l-1}) \quad (21.62)$$

However, a better approach is to use a **top-down inference model** of the form

$$q_\phi(\mathbf{z}|\mathbf{x}) = q_\phi(\mathbf{z}_L|\mathbf{x}) \prod_{l=L-1}^1 q_\phi(\mathbf{z}_l|\mathbf{x}, \mathbf{z}_{l+1:L}) \quad (21.63)$$

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 21.11(right) for an illustration.⁷

⁷. 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”.)

With the above model, the ELBO can be written as follows (using the chain rule for KL):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}_L | \mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{z}_L)) \quad (21.64)$$

$$- \sum_{l=L-1}^1 \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_{>l} | \mathbf{x})} [D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{>l}) \| p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{>l}))] \quad (21.65)$$

where

$$q_{\boldsymbol{\phi}}(\mathbf{z}_{>l} | \mathbf{x}) = \prod_{i=l+1}^L q_{\boldsymbol{\phi}}(\mathbf{z}_i | \mathbf{x}, \mathbf{z}_{>i}) \quad (21.66)$$

is the approximate posterior above layer l (i.e., the parents of \mathbf{z}_l).

The reason the top-down inference model is better is that it more closely approximates the true posterior of a given layer, which is given by

$$p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L}) \propto p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{l+1:L}) p_{\boldsymbol{\theta}}(\mathbf{x} | \mathbf{z}_l, \mathbf{z}_{l+1:L}) \quad (21.67)$$

Thus the posterior combines the top-down prior term $p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{l+1:L})$ with the bottom-up likelihood term $p_{\boldsymbol{\theta}}(\mathbf{x} | \mathbf{z}_l, \mathbf{z}_{l+1:L})$. We can approximate this posterior by defining

$$q_{\boldsymbol{\phi}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L}) \propto p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{l+1:L}) \tilde{q}_{\boldsymbol{\phi}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L}) \quad (21.68)$$

where $\tilde{q}_{\boldsymbol{\phi}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L})$ is a learned Gaussian approximation to the bottom-up likelihood. If both prior and likelihood are Gaussian, we can compute this product in closed form, as proposed in the **ladder network** paper [Sn+16; Søn+16].⁸ A more flexible approach is to let $q_{\boldsymbol{\phi}}(\mathbf{z}_l | \mathbf{x}, \mathbf{z}_{l+1:L})$ be learned, but to force it to share some of its parameters with the learned prior $p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{l+1:L})$, as proposed in [Kin+16]. This reduces the number of parameters in the model, and ensures that the posterior and prior remain somewhat close.

21.5.2 Example: very deep VAE

There have been many papers exploring different kinds of HVAE models (see e.g., [Kin+16; Sn+16; Chi21a; VK20a; Maa+19]), and we do not have space to discuss them all. Here we focus on the “**very deep VAE**” or **VD-VAE** model of [Chi21a], since it is simple but yields state of the art results (at the time of writing).

The architecture is a simple convolutional VAE with bidirectional inference, as shown in Figure 21.12. For each layer, the prior and posterior are diagonal Gaussians. The author found that nearest-neighbor upsampling (in the decoder) worked much better than transposed convolution, and avoided posterior collapse. This enabled training with the vanilla VAE objective, without needing any of the tricks discussed in Section 21.5.4.

The low-resolution latents (at the top of the hierarchy) capture a lot of the global structure of each image; the remaining high-resolution latents are just used to fill in details, that make the image look more realistic, and improve the likelihood. This suggests the model could be useful for lossy

⁸ The term “ladder network” arises from the horizontal “rungs” in Figure 21.11(right). Note that a similar idea was independently proposed in [Sal16].

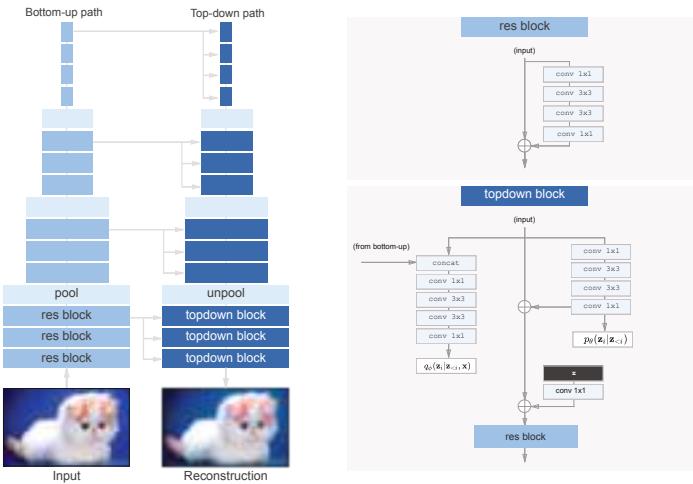


Figure 21.12: The top-down encoder used by the hierarchical VAE in [Chi21a]. Each convolution is preceded by the GELU nonlinearity. The model uses average pooling and nearest-neighbor upsampling for the pool and unpool layers. The posterior q_ϕ and prior p_θ are diagonal Gaussians. From Figure 3 of [Chi21a]. Used with kind permission of Rewon Child.



Figure 21.13: Samples from a VDVAE model (trained on FFHQ dataset) from different levels of the hierarchy. From Figure 1 of [Chi21a]. Used with kind permission of Rewon Child.

compression, since a lot of the low-level details can be drawn from the prior (i.e., “hallucinated”), rather than having to be sent by the encoder.

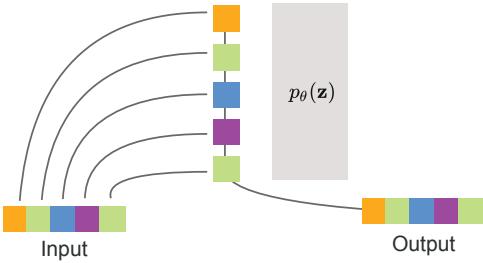
We can also use the model for unconditional sampling at multiple resolutions. This is illustrated in Figure 21.13, using a model with 78 stochastic layers trained on the FFHQ-256 dataset.⁹

21.5.3 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 22.3.2). However, by endowing VAEs with many more stochastic layers, it is possible to outperform AR models in terms of

9. 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.

Latent variables are identical to observed variables



Latent variables allow for parallel generation

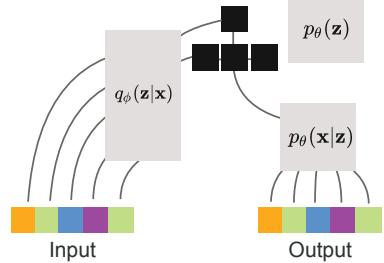


Figure 21.14: Left: a hierarchical VAE which emulates an autoregressive model using an identity encoder, autoregressive prior, and identity decoder. Right: a hierarchical VAE with a 2 layer hierarchical latent code. The bottom hidden nodes (black) are conditionally independent given the top layer. From Figure 2 of [Chi21a]. Used with kind permission of Rewon Child.

likelihood and sample quality, while using fewer parameters and much less computing power [Chi21a; VK20a; Maa+19].

To see why this is possible, note that we can represent any AR model as a degenerate VAE, as shown in Figure 21.14(left). The idea is simple: the encoder copies the input into latent space by setting $\mathbf{z}_{1:D} = \mathbf{x}_{1:D}$ (so $q_{\phi}(z_i | \mathbf{z}_{>i}, \mathbf{x}) = 1$), then the model learns an autoregressive prior $p_{\theta}(\mathbf{z}_{1:D}) = \prod_d p(z_d | \mathbf{z}_{1:d-1})$, and finally the likelihood function just copies the latent vector to output space, so $p_{\theta}(x_i = z_i | \mathbf{z}) = 1$. Since the encoder computes the exact (albeit degenerate) posterior, we have $q_{\phi}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{z}|\mathbf{x})$, so the ELBO is tight and reduces to the log likelihood,

$$\log p_{\theta}(\mathbf{x}) = \log p_{\theta}(\mathbf{z}) = \sum_d \log p_{\theta}(x_d | \mathbf{x}_{<d}) \quad (21.69)$$

Thus we can emulate any AR model with a VAE providing it has at least D stochastic layers, where D is the dimensionality of the observed data.

In practice, data usually lives in a lower-dimensional manifold (see e.g., [DW19]), which can allow for a much more compact latent code. For example, Figure 21.14(right) shows a hierarchical code in which the latent factors at the lower level are conditionally independent given the higher level, 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 32×32 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.

21.5.4 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 21.4.

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:

$$\sum_{l=1}^L \gamma_l \mathbb{E}_{q_\phi(z_{>l}|\mathbf{x})} [D_{\text{KL}}(q_\phi(z_l|\mathbf{x}, z_{>l}) \| p_\theta(z_l|z_{>l}))] \quad (21.70)$$

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 “warm up period”.) Concretely, [VK20a] proposes to set the coefficients γ_l to be proportional to the size of the layer, s_l , and the average KL loss:

$$\gamma_l \propto s_l \mathbb{E}_{\mathbf{x} \sim \mathcal{B}} [\mathbb{E}_{q_\phi(z_{>l}|\mathbf{x})} [D_{\text{KL}}(q_\phi(z_l|\mathbf{x}, z_{>l}) \| p_\theta(z_l|z_{>l}))]] \quad (21.71)$$

where \mathcal{B} is the current minibatch.

21.5.5 Other optimization difficulties

A common problem when training (hierarchical) VAEs is that the loss can become unstable. The main reason for this is that the KL term is unbounded (can become infinitely large). In [Chi21a], they tackle the problem in two ways. First, ensure the initial random weights of the final convolutional layer in each residual bottleneck block get scaled by $1/\sqrt{L}$. Second, skip an update step if the norm of the gradient of the loss exceeds some threshold.

In the **Nouveau VAE** method of [VK20a], they use some more complicated measures to ensure stability. First, they use batch normalization, but with various tweaks. Second, they use spectral regularization for the encoder. Specifically they add the penalty $\beta \sum_i \lambda_i$, where λ_i is the largest singular value of the i 'th convolutional layer (estimated using a single power iteration step), and $\beta \geq 0$ is a tuning parameter. Third, they use inverse autoregressive flows (Section 23.2.4.3) in each layer, instead of a diagonal Gaussian approximation. Fourth, they represent the posterior using a residual representation. In particular, let us assume the prior for the i 'th variable in layer l is

$$p_\theta(z_l^i | z_{>l}) = \mathcal{N}(z_l^i | \mu_i(z_{>l}), \sigma_i(z_{>l})) \quad (21.72)$$

They propose the following posterior approximation:

$$q_\phi(z_l^i | \mathbf{x}, z_{>l}) = \mathcal{N}(z_l^i | \mu_i(z_{>l}) + \Delta \mu_i(z_{>l}, \mathbf{x}), \sigma_i(z_{>l}) \cdot \Delta \sigma_i(z_{>l}, \mathbf{x})) \quad (21.73)$$

where the Δ terms are the relative changes computed by the encoder. The corresponding KL penalty reduces to the following (dropping the l subscript for brevity):

$$D_{\text{KL}}(q_\phi(z^i | \mathbf{x}, z_{>l}) \| p_\theta(z^i | z_{>l})) = \frac{1}{2} \left(\frac{\Delta \mu_i^2}{\sigma_i^2} + \Delta \sigma_i^2 - \log \Delta \sigma_i^2 - 1 \right) \quad (21.74)$$

So as long as σ_i is bounded from below, the KL term can be easily controlled just by adjusting the encoder parameters.



Figure 21.15: 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 [quantized_autoencoder_mnist.ipynb](#).

21.6 Vector quantization VAE

In this section, we describe **VQ-VAE**, which stands for ‘‘vector quantized VAE’’ [OVK17; ROV19]. This is like a standard VAE except it uses a set of discrete latent variables.

21.6.1 Autoencoder with binary code

The simplest approach to the problem is to construct a standard VAE, but to add a discretization 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 the number of discrete latents. For example, we can binarize the latent vector (using $S = 2$) by clipping \mathbf{z} to lie 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 21.4). Unfortunately, the discontinuous quantization operation 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 6.3.8. We show a simple example of this approach in Figure 21.15, where we use a Gaussian likelihood, so the loss function has the form

$$\mathcal{L} = \|\mathbf{x} - d(e(\mathbf{x}))\|_2^2 \quad (21.75)$$

where $e(\mathbf{x}) \in \{0, 1\}^K$ is the encoder, and $d(\mathbf{z}) \in \mathbb{R}^{28 \times 28}$ is the decoder.

21.6.2 VQ-VAE model

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 K is the number of discrete values per latent variable. Rather than just binarizing the continuous 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 then set \mathbf{z}_{ij} to the index of the nearest codebook entry:

$$q(\mathbf{z}_{ij} = k | \mathbf{x}) = \begin{cases} 1 & \text{if } k = \operatorname{argmin}_{k'} \|\mathbf{z}_e(\mathbf{x})_{i,j,:} - \mathbf{e}_{k'}\|_2 \\ 0 & \text{otherwise} \end{cases} \quad (21.76)$$

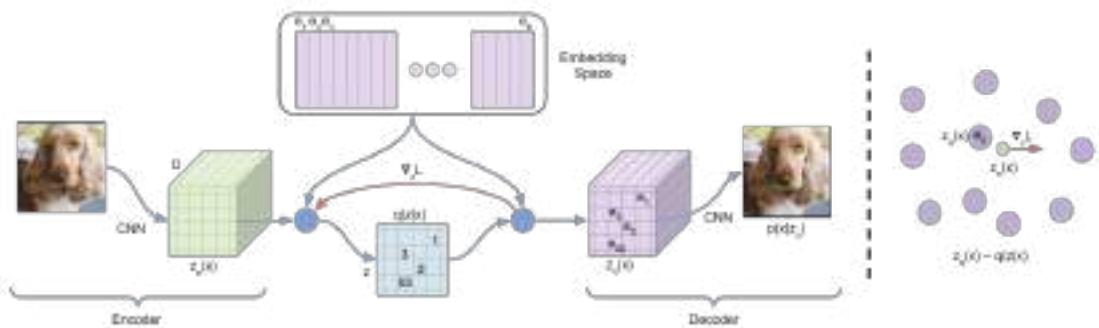


Figure 21.16: VQ-VAE architecture. From Figure 1 of [OVK17]. Used with kind permission of Aäron van den Oord.

When reconstructing the input we replace each discrete code index by the corresponding real-valued codebook vector:

$$(z_q)_{ij} = e_k \text{ where } z_{ij} = k \quad (21.77)$$

These values are then passed to the decoder, $p(x|z_q)$, as usual. See Figure 21.16 for an illustration of the overall architecture. Note that although z_q is generated from a discrete combination of codebook vectors, the use of a distributed code makes the model very expressive. For example, if we use a grid of 32×32 , with $K = 512$, then we can generate $512^{32 \times 32} = 2^{9216}$ distinct images, which is astronomically large.

To fit this model, we can minimize the negative log likelihood (reconstruction error) using the straight-through estimator, as before. This amounts to passing the gradients from the decoder input $z_q(x)$ to the encoder output $z_e(x)$, bypassing Equation (21.76), as shown by the red arrow in Figure 21.16. Unfortunately this means that the codebook entries will not get any learning signal. To solve this, the authors proposed to add an extra term to the loss, known as the **codebook loss**, that encourages the codebook entries e to match the output of the encoder. We treat the encoder $z_e(x)$ as a fixed target, by adding a **stop gradient** operator to it; this ensures z_e is treated normally in the forwards pass, but has zero gradient in the backwards pass. The modified loss (dropping the spatial indices i, j) becomes

$$\mathcal{L} = -\log p(x|z_q(x)) + \|\text{sg}(z_e(x)) - e\|_2^2 \quad (21.78)$$

where e refers to the codebook vector assigned to $z_e(x)$, and sg is the stop gradient operator.

An alternative way to update the codebook vectors is to use moving averages. To see how this works, first consider the batch setting. Let $\{z_{i,1}, \dots, z_{i,n_i}\}$ be the set of n_i outputs from the encoder that are closest to the dictionary item e_i . We can update e_i to minimize the MSE

$$\sum_{j=1}^{n_i} \|z_{i,j} - e_i\|_2^2 \quad (21.79)$$

which has the closed form update

$$\mathbf{e}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{z}_{i,j} \quad (21.80)$$

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:

$$N_i^t = \gamma N_i^{t-1} + (1 - \gamma) n_i^t \quad (21.81)$$

$$\mathbf{m}_i^t = \gamma \mathbf{m}_i^{t-1} + (1 - \gamma) \sum_j \mathbf{z}_{i,j}^t \quad (21.82)$$

$$\mathbf{e}_i^t = \frac{\mathbf{m}_i^t}{N_i^t} \quad (21.83)$$

The authors found $\gamma = 0.9$ to work well.

The above procedure will learn to update the codebook vectors so it matches 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:

$$\mathcal{L} = -\log p(\mathbf{x}|\mathbf{z}_q(\mathbf{x})) + \|\text{sg}(\mathbf{z}_e(\mathbf{x})) - \mathbf{e}\|_2^2 + \beta \|\mathbf{z}_e(\mathbf{x}) - \text{sg}(\mathbf{e})\|_2^2 \quad (21.84)$$

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.

21.6.3 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 22.3.2). More recent work has used transformer decoders (Section 22.4). 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.

21.6.4 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 21.17. 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 21.5 for more discussion of hierarchical VAEs.)

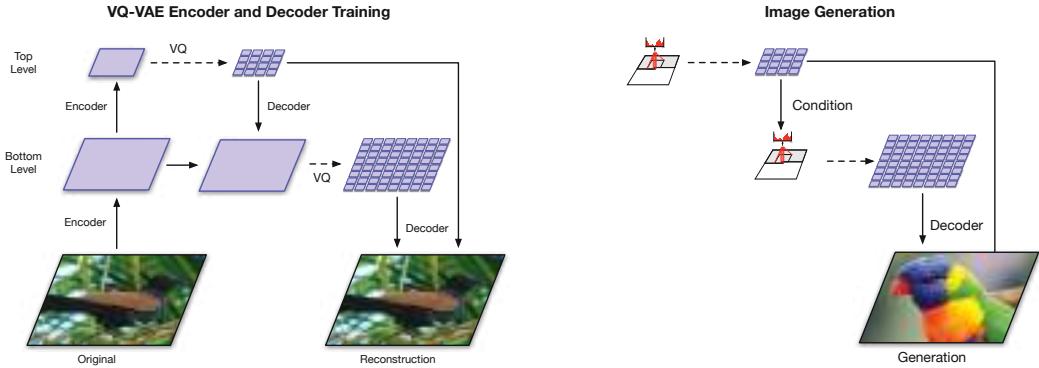


Figure 21.17: Hierarchical extension of VQ-VAE. (a) Encoder and decoder architecture. (b) Combining a Pixel-CNN prior with the decoder. From Figure 2 of [ROV19]. Used with kind permission of Aaron van den Oord.

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 21.17.

21.6.5 Discrete VAE

In VQ-VAE, we use a one-hot encoding for the latents, $q(z = k|\mathbf{x}) = 1$ iff $k = \operatorname{argmin}_k \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_k\|_2$, and then set $\mathbf{z}_q = \mathbf{e}_k$. This does not capture any uncertainty in the latent code, and requires the use of the straight-through estimator for training.

Various other approaches to fitting VAEs with discrete latent codes have been investigated. In the DALL-E paper (Section 22.4.2), they use a fairly simple method, based on using the Gumbel-softmax relaxation for the discrete variables (see Section 6.3.6). In brief, let $q(z = k|\mathbf{x})$ be the probability that the input \mathbf{x} is assigned to codebook entry k . We can exactly sample $w_k \sim q(z = k|\mathbf{x})$ from this by computing $w_k = \operatorname{argmax}_k g_k + \log q(z = k|\mathbf{x})$, where each g_k is from a Gumbel distribution. We can now “relax” this by using a softmax with temperature $\tau > 0$ and computing

$$w_k = \frac{\exp(\frac{g_k + \log q(z=k|\mathbf{x})}{\tau})}{\sum_{j=1}^K \exp(\frac{g_j + \log q(z=j|\mathbf{x})}{\tau})} \quad (21.85)$$

We now set the latent code to be a weighted sum of the codebook vectors:

$$\mathbf{z}_q = \sum_{k=1}^K w_k \mathbf{e}_k \quad (21.86)$$

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.

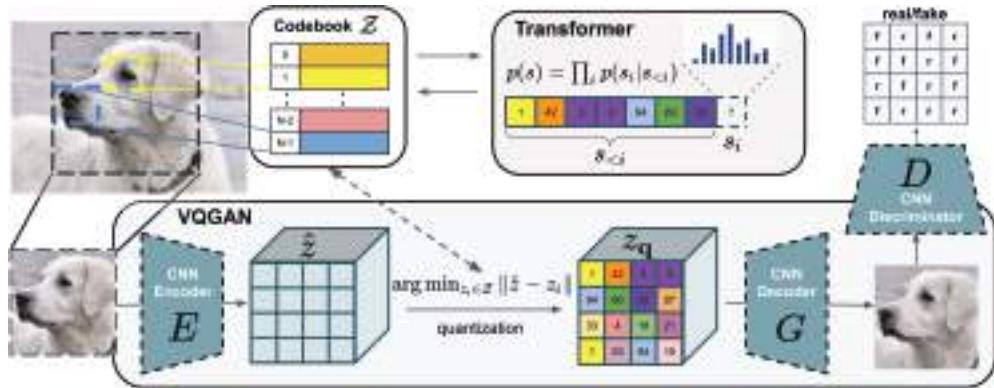


Figure 21.18: Illustration of the VQ-GAN. From Figure 2 of [ERO21]. Used with kind permission of Patrick Esser.

This allows us to express the ELBO in the usual differentiable way:

$$\mathcal{L} = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x}|\mathbf{z})] + \beta D_{\text{KL}}(q(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z})) \quad (21.87)$$

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 6.3.5) to optimize this.

21.6.6 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 26), together with a perceptual loss; this results in much higher visual fidelity. In addition, they use a transformer (see Section 16.3.5) to model the prior on the latent codes. See Figure 21.18 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 Autoregressive models

22.1 Introduction

By the chain rule of probability, we can write any joint distribution over T variables as follows:

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2, \mathbf{x}_1)p(\mathbf{x}_4|\mathbf{x}_3, \mathbf{x}_2, \mathbf{x}_1)\dots = \prod_{t=1}^T p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) \quad (22.1)$$

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 **autoregressive model** or **ARM**. This corresponds to a fully connected DAG, in which each node depends on all its predecessors in the ordering, as shown in Figure 22.1. 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

$$p(\mathbf{x}_{1:T}) = \prod_{t=T}^1 p(\mathbf{x}_t|\mathbf{x}_{t+1:T}) \quad (22.2)$$

However, this “anti-causal” direction is often harder to learn (see e.g., [PJS17]).

Although the decomposition in Equation (22.1) 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 2.6.3.2).

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 2.6. (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 16.3.4. 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 29.2.

Another approach is to stay with the general AR model of Equation (22.1), 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

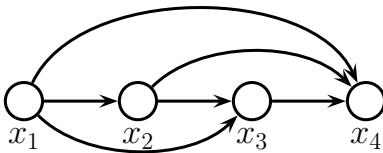


Figure 22.1: A fully-connected auto-regressive model.

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.

22.2 Neural autoregressive density estimators (NADE)

A simple way to represent each conditional probability distribution $p(x_t|\mathbf{x}_{1:t-1})$ is to use a generalized linear model, such as logistic regression, as proposed in [Fre98]. We can make the model be more powerful by using a neural network. The resulting model is called the **neural autoregressive density estimator** or **NADE** model [LM11].

If we let $p(x_t|\mathbf{x}_{1:t-1})$ be a conditional mixture of Gaussians, we get a model known as **RNADE** (“real-valued neural autoregressive density estimator”) of [UML13]. More precisely, this has the form

$$p(x_t|\mathbf{x}_{1:t-1}) = \sum_{k=1}^K \pi_{t,k} \mathcal{N}(x_t|\mu_{t,k}, \sigma_{t,k}^2) \quad (22.3)$$

where the parameters are generated by a network, $(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t, \boldsymbol{\pi}_t) = f_t(\mathbf{x}_{1:t-1}; \boldsymbol{\theta}_t)$.

Rather than using separate neural networks, f_1, \dots, f_T , it is more efficient to create a single network with T inputs and T outputs. This can be done using masking, resulting in a model called the **MADE** (“masked autoencoder for density estimation”) model [Ger+15].

One disadvantage of NADE-type models is that they assume the variables have a natural linear ordering. This makes sense for temporal or sequential data, but not for more general data types, such as images or graphs. An orderless extension to NADE was proposed in [UML14; Uri+16].

22.3 Causal CNNs

One approach to representing the distribution $p(x_t|\mathbf{x}_{1:t-1})$ is to try to identify patterns in the past history that might be predictive of the value of x_t . If we assume these patterns can occur in any location, it makes sense to use a **convolutional neural network** to detect them. However, we need 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.

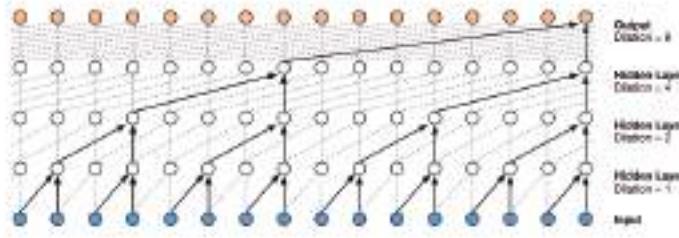


Figure 22.2: Illustration of the wavenet model using dilated (atrous) convolutions, with dilation factors of 1, 2, 4 and 8. From Figure 3 of [Oor+16a]. Used with kind permission of Aäron van den Oord.

22.3.1 1d causal CNN (convolutional Markov models)

Consider the following **convolutional Markov model** for 1d discrete sequences:

$$p(\mathbf{x}_{1:T}) = \prod_{t=1}^T p(x_t | \mathbf{x}_{1:t-1}; \theta) = \prod_{t=1}^T \text{Cat}(x_t | \text{softmax}(\varphi(\sum_{\tau=1}^{t-k} \mathbf{w}^\top \mathbf{x}_{\tau:\tau+k}))) \quad (22.4)$$

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+16a]. See Figure 22.2 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 23.2.4.3.

22.3.2 2d causal CNN (PixelCNN)

We can extend causal convolutions to 2d, to get an autoregressive model of the form

$$p(\mathbf{x}|\theta) = \prod_{r=1}^R \prod_{c=1}^C p(x_{r,c} | f_\theta(\mathbf{x}_{1:r-1,1:C}, \mathbf{x}_{r,1:c-1})) \quad (22.5)$$

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 22.3. This is called the **pixelCNN** model [Oor+16b]. Naive sampling (generation) from this model takes $O(N)$ time, where $N = RC$ is the number of pixels, but [Ree+17] shows how to use a multiscale approach to reduce the complexity to $O(\log N)$.

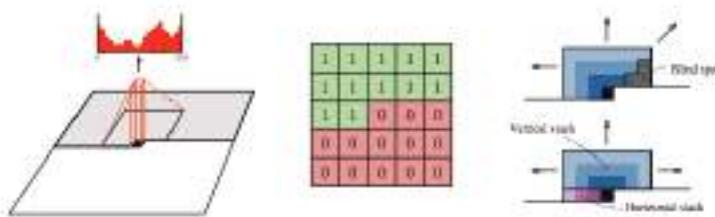


Figure 22.3: Illustration of causal 2d convolution in the PixelCNN model. The red histogram shows the empirical distribution over discretized values for a single pixel of a single RGB channel. The red and green 5×5 array shows the binary mask, which selects the top left context, in order to ensure the convolution is causal. The diagrams on the right illustrate how we can avoid blind spots by using a vertical context stack, that contains all previous rows, and a horizontal context stack, that just contains values from the current row. From Figure 1 of [Oor+16b]. Used with kind permission of Aaron van den Oord.

Various extensions of this model have been proposed. The **pixelCNN++** model of [Sal+17c] improved the quality by using a mixture of logistic distributions, to capture the multimodality of $p(x_i|x_{1:i-1})$. The **pixelRNN** of [OKK16] combined masked convolution with an RNN to get even longer range contextual dependencies. The **Subscale Pixel Network** of [MK19] proposed to generate the pixels such that the higher order bits are sampled before lower order bits, which allows high resolution details to be sampled conditioned on low resolution versions of the whole image, rather than just the top left corner.

22.4 Transformers

We introduced transformers in Section 16.3.5. They can be used for encoding sequences (as in BERT), or for decoding (generating) sequences. We can also combine the two, using an encoder-decoder combination, for conditional generation from $p(\mathbf{y}|\mathbf{c})$. Alternatively, we can define a joint sequence model $p(\mathbf{c}, \mathbf{y})$, where \mathbf{c} is the conditioning or context prompt, and then just condition the joint model, by giving it as the initial context.

The decoder (generator) works as follows. At each step t , the model applies masked (causal) self attention (Section 16.2.7) to the first t inputs, $\mathbf{y}_{1:t}$, to compute a set of attention weights, $\mathbf{a}_{1:t}$. From this it computes an activation vector $\mathbf{z}_t = \sum_{\tau=1}^t a_{t\tau} \mathbf{y}_\tau$. This is then passed through a feed-forward layer to compute $\mathbf{h}_t = \text{MLP}(\mathbf{z}_t)$. This process is repeated for each layer in the model. Finally the output is used to predict the next element in the sequence, $\mathbf{y}_{t+1} \sim \text{Cat}(\text{softmax}(\mathbf{W}\mathbf{h}_t))$.

At training time, all predictions can happen in parallel, since the target generated sequence is already available. That is, the t 'th output \mathbf{y}_t can be predicted given inputs $\mathbf{y}_{1:t-1}$, and this can be done for all t simultaneously. However, at test time, the model must be applied sequentially, so the output generated at $t+1$ is fed back into the model to predict $t+2$, etc. Note that the running time of transformers is $O(T^2)$, although a variety of more efficient versions have been developed (see e.g., [Mur22, Sec 15.6] for details).

Transformers are the basis of many popular (conditional) generative models for sequences. We give some examples below.

A “whapte” is a small, furry animal native to Tasmania. An example of a sentence that uses the word Whapte is:
We were traveling in Africa and we saw these very cute whaptes.

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 “yaluhala” is a type of vegetable that looks like a big pumpkin. An example of a sentence that uses the word yaluhala is:

I was on a trip to Africa and I tried this yaluhala vegetable that was grown in a garden there. It was delicious.

A “Burrings” is a car with very fast acceleration. An example of a sentence that uses the word Burrings is:

In our garage we have a Burrings 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 “screech” something is to swing a sword at it. An example of a sentence that uses the word SCREECH is:

We screeched at each other for several minutes and then we went outside and ate ice cream.

Figure 22.4: Illustration of few shot learning with GPT-3. The model is asked to create an example sentence using a new word whose meaning is provided in the prompt. Boldface is GPT-3’s completions, light gray is human input. From Figure 3.16 of [Bro+20d].

22.4.1 Text generation (GPT, etc.)

In [Rad+18], OpenAI proposed a model called **GPT**, which is short for “Generative Pre-training Transformer”. This is a decoder-only transformer model that uses causal (masked) attention. In [Rad+19], they propose **GPT-2**, which is a larger version of GPT (1.5 billion parameters, or 6.5GB, for the XL version), trained on a large web corpus (8 million pages, or 40GB). They also simplify the training objective, and just train it using maximum likelihood.

After GPT-2, OpenAI released **GPT-3** [Bro+20d], which is an even larger version of GPT-2 (175 billion parameters), trained on even more data (300 billion words), but based on the same principles. (Training was estimated to take 355 GPU years and cost \$4.6M.) Due to the large size of the data and model, GPT-3 shows even more remarkable abilities to generate novel text. In particular, the output can be (partially) controlled by just changing the conditioning prompt. This enables the model to perform tasks that it has never been trained on, just by giving it some examples in the prompt. This is called “**in-context learning**”. See Figure 22.4 for an example, and Section 19.5.1.2 for more details.

More recently, OpenAI released **ChatGPT** [Ope], which is an improved version of GPT-3 which has been trained to have interactive dialogs by using a technique called **reinforcement learning from human feedback** or **RLHF**, a technique first introduced in the **InstructGPT** paper [Ouy+22]. This uses RL techniques such as PPO (Section 35.3.4) to fine tune the model so that it generates responses that are more “aligned” with human intent, as estimated by a ranking model, which is pre-trained on supervised data.

```
(JAX):
Yo, PyTorch, you think you're hot stuff.
But JAX is where it's at, no need to bluff.
Your popularity might be on the rise
But when it comes to performance, we're the guys.

With automatic differentiation, we're a breeze
Training neural nets with ease
Our functional approach is innovative and unique
PyTorch can't touch our speed, try and keep up, meek

(PyTorch):
Hold up, JAX, you might be fast
But PyTorch is the one that's built to last
We may not have the fancy functional flair
But our flexibility sets us above the pair

Our community is strong and growing each day
Supporting us in every single way
We may not be the newest on the block
But we'll always be the torch that leads the flock
```

Figure 22.5: ChatGPT response to the prompt “Write a rap battle about PyTorch and JAX”. Used with kind permission of Paige Bailey. From <https://twitter.com/DynamicWebPaige/status/1601743574369902593>.

Despite the impressive performance of these **large language models** or **LLMs** (see Figure 22.5 for an example), there are several open problems with them, such as: they often confidently **hallucinate** incorrect answers to questions (see e.g., [Ji+22]); they can generate biased or toxic output (see e.g., [Lia+]); and they are very resource intensive to train and serve. Indeed, these concerns are why Google has not (at the time of writing) released its version of ChatGPT, known as **LaMDA** [Col21].

The basic ideas behind LLMs are quite simple (maximum likelihood training of an autoregressive transformer), and they can be implemented in about 300 lines of code.¹ However, just by scaling up the size of the models and datasets, it seems that qualitatively new capabilities can emerge (see e.g., [Wei+22]). Nevertheless, although this approach is good at learning formal linguistic competence (surface form), it is not clear if it is sufficient to learn functional linguistic competence, which requires a deeper, non-linguistic understanding of the world derived from experience [Mah+23].

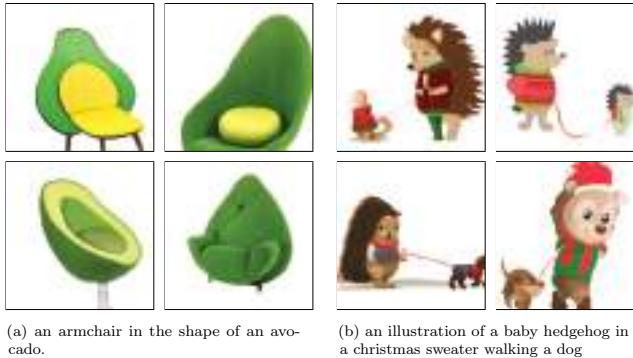
22.4.2 Image generation (DALL-E, etc.)

The **DALL-E** model² from OpenAI [Ram+21a] can generate images of remarkable quality and diversity given text prompts, as shown in Figure 22.6. The methodology is conceptually quite straightforward, and most of the effort went into data collection (they scraped the web for 250 million image-text pairs) and scaling up the training (they fit a model with 12 billion parameters). Here we just focus on the algorithmic methods.

The basic idea is to transform an image x into a sequence of discrete tokens z using a discrete

1. See e.g., <https://github.com/karpathy/nanoGPT>.

2. The name is derived from the artist Salvador Dalí and Pixar’s movie “WALL-E”



(a) an armchair in the shape of an avocado.

(b) an illustration of a baby hedgehog in a christmas sweater walking a dog

Figure 22.6: Some images generated by the DALL-E model in response to a text prompt. (a) “An armchair in the shape of an avocado”. (b) “An illustration of a baby hedgehog in a christmas sweater walking a dog”. From <https://openai.com/blog/dall-e>. Used with kind permission of Aditya Ramesh.

VAE model (Section 21.6.5). We then fit a transformer to the concatenation of the image tokens \mathbf{z} and text tokens \mathbf{y} to get a joint model of the form $p(\mathbf{z}, \mathbf{y})$.

To sample an image \mathbf{x} given a text prompt \mathbf{y} , we sample a latent code $\mathbf{z} \sim p(\mathbf{z}|\mathbf{y})$ by conditioning the transformer on the prompt \mathbf{y} , and then we feed \mathbf{z} into the VAE decoder to get the image $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$. Multiple images are generated for each prompt, and these are then ranked according to a pre-trained critic, which gives them scores depending on how well the generated image matches the input text: $s_n = \text{critic}(\mathbf{x}_n, \mathbf{y}_n)$. The critic they used was the contrastive CLIP model (see Section 32.3.4.1). This discriminative reranking significantly improves the results.

Some sample results are shown in Figure 22.6, and more can be found online at <https://openai.com/blog/dall-e/>. The image on the right of Figure 22.6 is particularly interesting, since the prompt — “An illustration of a baby hedgehog in a christmas sweater walking a dog” — arguably requires that the model solve the “**variable binding problem**”. This refers to the fact that the sentence implies the hedgehog should be wearing the sweater and not the dog. We see that the model sometimes interprets this correctly, but not always: sometimes it draws both animals with Christmas sweaters. In addition, sometimes it draws a hedgehog walking a smaller hedgehog. The quality of the results can also be sensitive to the form of the prompt.

The **PARTI** model [Yu+22] from Google follows similar high level ideas to DALL-E, but has been scaled to an even larger size. The larger models perform qualitatively much better, as shown in Figure 20.3.

Other recent approaches to (conditional) image generation — such as **DALL-E 2** [Ram+22] from Open-AI, **Imagen** [Sah+22b] from Google, and **Stable diffusion** [Rom+22] from Stability.AI — are based on diffusion rather than applying a transformer to discretized image patches. See Section 25.6.4 for details.

22.4.3 Other applications

Transformers have been used to generate many other kinds of (discrete) data, such as midi music sequences [Hua+18a], protein sequences [Gan+23], etc.

23 Normalizing flows

This chapter is written by George Papamakarios and Balaji Lakshminarayanan.

23.1 Introduction

In this chapter we discuss **normalizing flows**, a class of flexible density models that can be easily sampled from and whose exact likelihood function is efficient to compute. Such models can be used for many tasks, such as density modeling, inference and generative modeling. We introduce the key principles of normalizing flows and refer to recent surveys by Papamakarios et al. [Pap+19] and Kobyzev, Prince, and Brubaker [KPB19] for readers interested in learning more. See also <https://github.com/janosh/awesome-normalizing-flows> for a list of papers and software packages.

23.1.1 Preliminaries

Normalizing flows create complex probability distributions $p(\mathbf{x})$ by passing random variables $\mathbf{u} \in \mathbb{R}^D$, drawn from a simple **base distribution** $p(\mathbf{u})$ through a nonlinear but *invertible* transformation $\mathbf{f} : \mathbb{R}^D \rightarrow \mathbb{R}^D$. That is, $p(\mathbf{x})$ is defined by the following process:

$$\mathbf{x} = \mathbf{f}(\mathbf{u}) \quad \text{where} \quad \mathbf{u} \sim p(\mathbf{u}). \tag{23.1}$$

The base distribution is typically chosen to be simple, for example standard Gaussian or uniform, so that we can easily sample from it and compute the density $p(\mathbf{u})$. A flexible enough transformation \mathbf{f} can induce a complex distribution on the transformed variable \mathbf{x} even if the base distribution is simple.

Sampling from $p(\mathbf{x})$ is straightforward: we first sample \mathbf{u} from $p(\mathbf{u})$ and then compute $\mathbf{x} = \mathbf{f}(\mathbf{u})$. To compute the density $p(\mathbf{x})$, we rely on the fact that \mathbf{f} is invertible. Let $\mathbf{g}(\mathbf{x}) = \mathbf{f}^{-1}(\mathbf{x}) = \mathbf{u}$ be the inverse mapping, which “**normalizes**” the data distribution by mapping it back to the base distribution (which is often a normal distribution). Using the change-of-variables formula for random variables from Equation (2.257), we have

$$p_x(\mathbf{x}) = p_u(\mathbf{g}(\mathbf{x})) |\det \mathbf{J}(\mathbf{g})(\mathbf{x})| = p_u(\mathbf{u}) |\det \mathbf{J}(\mathbf{f})(\mathbf{u})|^{-1}, \tag{23.2}$$

where $\mathbf{J}(\mathbf{f})(\mathbf{u}) = \frac{\partial \mathbf{f}}{\partial \mathbf{u}}|_{\mathbf{u}}$ is the Jacobian matrix of \mathbf{f} evaluated at \mathbf{u} . Taking logs of both sides of Equation (23.2), we get

$$\log p_x(\mathbf{x}) = \log p_u(\mathbf{u}) - \log |\det \mathbf{J}(\mathbf{f})(\mathbf{u})|. \tag{23.3}$$

As discussed above, $p(\mathbf{u})$ is typically easy to evaluate. So, if one can use flexible invertible transformations \mathbf{f} whose Jacobian determinant $\det \mathbf{J}(\mathbf{f})(\mathbf{u})$ can be computed efficiently, then one can construct complex densities $p(\mathbf{x})$ that allow exact sampling and efficient exact likelihood computation. This is in contrast to latent variable models, which require methods like variational inference to lower-bound the likelihood.

One might wonder how flexible are the densities $p(\mathbf{x})$ obtained by transforming random variables sampled from simple $p(\mathbf{u})$. It turns out that we can use this method to approximate any smooth distribution. To see this, consider the scenario where the base distribution $p(\mathbf{u})$ is a one-dimensional uniform distribution. Recall that inverse transform sampling (Section 11.3.1) samples random variables from a uniform distribution and transforms them using the inverse cumulative distribution function (cdf) to generate samples from the desired density. We can use this method to sample from any one-dimensional density as long as the transformation \mathbf{f} is powerful enough to model the inverse cdf (which is a reasonable assumption for well-behaved densities whose cdf is invertible and differentiable). We can further extend this argument to multiple dimensions by first expressing the density $p(\mathbf{x})$ as a product of one-dimensional conditionals using the chain rule of probability, and then applying inverse transform sampling to each one-dimensional conditional. The result is a normalizing flow that transforms a product of uniform distributions into any desired distribution $p(\mathbf{x})$. We refer to [Pap+19] for a more detailed proof.

How do we define flexible invertible mappings whose Jacobian determinant is easy to compute? We discuss this topic in detail in Section 23.2, but in summary, there are two main ways. The first approach is to define a set of simple transformations that are invertible by design, and whose Jacobian determinant is easy to compute; for instance, if the Jacobian is a triangular matrix, its determinant can be computed efficiently. The second approach is to exploit the fact that a composition of invertible functions is also invertible, and the overall Jacobian determinant is just the product of the individual Jacobian determinants. More precisely, if $\mathbf{f} = \mathbf{f}_N \circ \dots \circ \mathbf{f}_1$ where each \mathbf{f}_i is invertible, then \mathbf{f} is also invertible, with inverse $\mathbf{g} = \mathbf{g}_1 \circ \dots \circ \mathbf{g}_N$ and log Jacobian determinant given by

$$\log |\det \mathbf{J}(\mathbf{g})(\mathbf{x})| = \sum_{i=1}^N \log |\det \mathbf{J}(\mathbf{g}_i)(\mathbf{u}_i)| \quad (23.4)$$

where $\mathbf{u}_i = \mathbf{f}_i \circ \dots \circ \mathbf{f}_1(\mathbf{u})$ is the i 'th intermediate output of the flow. This allows us to create complex flows from simple components, just as graphical models allow us to create complex joint distributions from simpler conditional distributions.

Finally, a note on terminology. An invertible transformation is also known as a **bijection**. A bijection that is differentiable and has a differentiable inverse is known as a **diffeomorphism**. The transformation \mathbf{f} of a flow model is a diffeomorphism, although in the rest of this chapter we will refer to it as a “bijection” for simplicity, leaving the differentiability implicit. The density $p_x(\mathbf{x})$ of a flow model is also known as the **pushforward** of the base distribution $p_u(\mathbf{u})$ through the transformation \mathbf{f} , and is sometimes denoted as $p_x = \mathbf{f}_* p_u$. Finally, in mathematics the term “flow” refers to any family of diffeomorphisms \mathbf{f}_t indexed by a real number t such that $t = 0$ indexes the identity function, and $t_1 + t_2$ indexes $\mathbf{f}_{t_2} \circ \mathbf{f}_{t_1}$ (in physics, t often represents time). In machine learning we use the term “flow” by analogy to the above meaning, to highlight the fact that we can create flexible invertible transformations by composing simpler ones; in this sense, the index t is analogous to the number i of transformations in $\mathbf{f}_i \circ \dots \circ \mathbf{f}_1$.

23.1.2 How to train a flow model

There are two common applications of normalizing flows. The first one is density estimation of observed data, which is achieved by fitting $p_{\theta}(\mathbf{x})$ to the data and using it as an estimate of the data density, potentially followed by generating new data from $p_{\theta}(\mathbf{x})$. The second one is variational inference, which involves sampling from and evaluating a variational posterior $q_{\theta}(\mathbf{z}|\mathbf{x})$ parameterized by the flow model. As we will see below, these applications optimize different objectives and impose different computational constraints on the flow model.

23.1.2.1 Density estimation

Density estimation requires maximizing the likelihood function in Equation (23.2). This requires that we can efficiently evaluate the inverse flow $\mathbf{u} = \mathbf{f}^{-1}(\mathbf{x})$ and its Jacobian determinant $\det \mathbf{J}(\mathbf{f}^{-1})(\mathbf{x})$ for any given \mathbf{x} . After optimizing the model, we can optionally use it to generate new data. To sample new points, we require that the forwards mapping \mathbf{f} be tractable.

23.1.2.2 Variational inference

Normalizing flows are commonly used for variational inference to parameterize the approximate posterior distribution in latent variable models, as discussed in Section 10.4.3. Consider a latent variable model with continuous latent variables \mathbf{z} and observable variables \mathbf{x} . For simplicity, we consider the model parameters to be fixed as we are interested in approximating the true posterior $p^*(\mathbf{z}|\mathbf{x})$ with a normalizing flow $q_{\theta}(\mathbf{z}|\mathbf{x})$.¹ As discussed in Section 10.1.1.2, the variational parameters are trained by maximizing the evidence lower bound (ELBO), given by

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\theta}(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z}) - \log q_{\theta}(\mathbf{z}|\mathbf{x})] \quad (23.5)$$

When viewing the ELBO as a function of $\boldsymbol{\theta}$, it can be simplified as follows (note we drop the dependency on \mathbf{x} for simplicity):

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\theta}(\mathbf{z})} [\ell_{\boldsymbol{\theta}}(\mathbf{z})]. \quad (23.6)$$

Let $q_{\theta}(\mathbf{z})$ denote a normalizing flow with base distribution $q(\mathbf{u})$ and transformation $\mathbf{z} = f_{\boldsymbol{\theta}}(\mathbf{u})$. Then the reparameterization trick (Section 6.3.5) allows us to optimize the parameters using stochastic gradients. To achieve this, we first write the expectation with respect to the base distribution:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\theta}(\mathbf{z})} [\ell_{\boldsymbol{\theta}}(\mathbf{z})] = \mathbb{E}_{q(\mathbf{u})} [\ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{u}))]. \quad (23.7)$$

Then, since the base distribution does not depend on $\boldsymbol{\theta}$, we can obtain stochastic gradients as follows:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \mathbb{E}_{q(\mathbf{u})} [\nabla_{\boldsymbol{\theta}} \ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{u}))] \approx \frac{1}{N} \sum_{n=1}^N \nabla_{\boldsymbol{\theta}} \ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{u}_n)), \quad (23.8)$$

where $\{\mathbf{u}_n\}_{n=1}^N$ are samples from $q(\mathbf{u})$.

¹ We denote the parameters of the variational posterior by $\boldsymbol{\theta}$ here, which should not be confused with the model parameters which are also typically denoted by $\boldsymbol{\theta}$ elsewhere.

As we can see, in order to optimize this objective, we need to be able to efficiently sample from $q_\theta(\mathbf{z}|\mathbf{x})$ and evaluate the probability density of these samples during optimization. (See Section 23.2.4.3 for details on how to do this.) This is contrast to the MLE approach in Section 23.1.2.1, which requires that we be able to compute efficiently the density of arbitrary training datapoints, but it does not require samples during optimization.

23.2 Constructing flows

In this section, we discuss how to compute various kinds of flows that are invertible by design and have efficiently computable Jacobian determinants.

23.2.1 Affine flows

A simple choice is to use an affine transformation $\mathbf{x} = \mathbf{f}(\mathbf{u}) = \mathbf{A}\mathbf{u} + \mathbf{b}$. This is a bijection if and only if \mathbf{A} is an invertible square matrix. The Jacobian determinant of \mathbf{f} is $\det \mathbf{A}$, and its inverse is $\mathbf{u} = \mathbf{f}^{-1}(\mathbf{x}) = \mathbf{A}^{-1}(\mathbf{x} - \mathbf{b})$. A flow consisting of affine bijections is called an **affine flow**, or a **linear flow** if we ignore \mathbf{b} .

On their own, affine flows are limited in their expressive power. For example, suppose the base distribution is Gaussian, $p(\mathbf{u}) = \mathcal{N}(\mathbf{u}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Then the pushforward distribution after an affine bijection is still Gaussian, $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^\top)$. However, affine bijections are useful building blocks when composed with the non-affine bijections we discuss later, as they encourage “mixing” of dimensions through the flow.

For practical reasons, we need to ensure the Jacobian determinant and the inverse of the flow are fast to compute. In general, computing $\det \mathbf{A}$ and \mathbf{A}^{-1} explicitly takes $O(D^3)$ time. To reduce the cost, we can add structure to \mathbf{A} . If \mathbf{A} is diagonal, the cost becomes $O(D)$. If \mathbf{A} is triangular, the Jacobian determinant is the product of the diagonal elements, so it takes $O(D)$ time; inverting the flow requires solving the triangular system $\mathbf{A}\mathbf{u} = \mathbf{x} - \mathbf{b}$, which can be done with backsubstitution in $O(D^2)$ time.

The result of a triangular transformation depends on the ordering of the dimensions. To reduce sensitivity to this, and to encourage “mixing” of dimensions, we can multiply \mathbf{A} with a permutation matrix, which has an absolute determinant of 1. We often use a permutation that reverses the indices at each layer or that randomly shuffles them. However, usually the permutation at each layer is fixed rather than learned.

For spatially structured data (such as images), we can define \mathbf{A} to be a convolution matrix. For example, GLOW [KD18b] uses 1×1 convolution; this is equivalent to pointwise linear transformation across feature dimensions, but regular convolution across spatial dimensions. Two more general methods for modeling $d \times d$ convolutions are presented in [HBW19], one based on stacking autoregressive convolutions, and the other on carrying out the convolution in the Fourier domain.

23.2.2 Elementwise flows

Let $h : \mathbb{R} \rightarrow \mathbb{R}$ be a scalar-valued bijection. We can create a vector-valued bijection $\mathbf{f} : \mathbb{R}^D \rightarrow \mathbb{R}^D$ by applying h elementwise, that is, $\mathbf{f}(\mathbf{u}) = (h(u_1), \dots, h(u_D))$. The function \mathbf{f} is invertible, and its Jacobian determinant is given by $\prod_{i=1}^D \frac{dh}{du_i}$. A flow composed of such bijections is known as an **elementwise flow**.

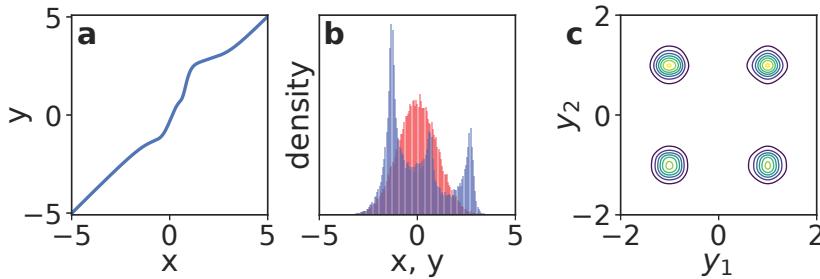


Figure 23.1: Non-linear squared flow (NLSq). Left: an invertible mapping consisting of 4 NLSq layers. Middle: red is the base distribution (Gaussian), blue is the distribution induced by the mapping on the left. Right: density of a 5-layer autoregressive flow using NLSq transformations and a Gaussian base density, trained on a mixture of 4 Gaussians. From Figure 5 of [ZR19b]. Used with kind permission of Zachary Ziegler.

On their own, elementwise flows are limited, since they do not model dependencies between the elements. However, they are useful building blocks for more complex flows, such as coupling flows (Section 23.2.3) and autoregressive flows (Section 23.2.4), as we will see later. In this section, we discuss techniques for constructing scalar-valued bijections $h : \mathbb{R} \rightarrow \mathbb{R}$ for use in elementwise flows.

23.2.2.1 Affine scalar bijection

An **affine scalar bijection** has the form $h(u; \theta) = au + b$, where $\theta = (a, b) \in \mathbb{R}^2$. (This is a scalar version of an affine flow.) Its derivative $\frac{dh}{du}$ is equal to a . It is invertible if and only if $a \neq 0$. In practice, we often parameterize a to be positive, for example by making it the exponential or the softplus of an unconstrained parameter. When $a = 1$, $h(u; \theta) = u + b$ is often called an **additive scalar bijection**.

23.2.2.2 Higher-order perturbations

The affine scalar bijection is simple to use, but limited. We can make it more flexible by adding higher-order perturbations, under the constraint that invertibility is preserved. For example, Ziegler and Rush [ZR19b] propose the following, which they term **non-linear squared flow**:

$$h(u; \theta) = au + b + \frac{c}{1 + (du + e)^2}, \quad (23.9)$$

where $\theta = (a, b, c, d, e) \in \mathbb{R}^5$. When $c = 0$, this reduces to the affine case. When $c \neq 0$, it adds an inverse-quadratic perturbation, which can induce multimodality as shown in Figure 23.1. Under the constraints $a > \frac{9}{8\sqrt{3}}cd$ and $d > 0$ the function becomes invertible, and its inverse can be computed analytically by solving a quadratic polynomial.

23.2.2.3 Combinations of strictly monotonic scalar functions

A strictly monotonic scalar function is one that is always increasing (has positive derivative everywhere) or always decreasing (has negative derivative everywhere). Such functions are invertible. Many

activation functions, such as the logistic sigmoid $\sigma(u) = 1/(1 + \exp(-u))$, are strictly monotonic.

Using such activation functions as a starting point, we can build more flexible monotonic functions via **conical combination** (linear combination with positive coefficients) and function composition. Suppose h_1, \dots, h_K are strictly increasing; then the following are also strictly increasing:

- $a_1 h_1 + \dots + a_K h_K + b$ with $a_k > 0$ (conical combination with a bias),
- $h_1 \circ \dots \circ h_K$ (function composition).

By repeating the above two constructions, we can build arbitrarily complex increasing functions. For example, a composition of conical combinations of logistic sigmoids is just an MLP where all weights are positive [Hua+18b].

The derivative of such a scalar bijection can be computed by repeatedly applying the chain rule, and in practice can be done with automatic differentiation. However, the inverse is not typically computable in closed form. In practice we can compute the inverse using bisection search, since the function is monotonic.

23.2.2.4 Scalar bijections from integration

A simple way to ensure a scalar function is strictly monotonic is to constrain its derivative to be positive. Let $h' = \frac{dh}{du}$ be this derivative. Wehenkel and Louppe [WL19] directly parameterize h' with a neural network whose output is made positive via an ELU activation function shifted up by 1. They then integrate the derivative numerically to get the bijection:

$$h(u) = \int_0^u h'(t) dt + b, \quad (23.10)$$

where b is a bias. They call this approach **unconstrained monotonic neural networks**.

The above integral is generally not computable in closed form. It can be, however, if h' is constrained appropriately. For example, Jaini, Selby, and Yu [JSY19] take h' to be a sum of K squared polynomials of degree L :

$$h'(u) = \sum_{k=1}^K \left(\sum_{\ell=0}^L a_{k\ell} u^\ell \right)^2. \quad (23.11)$$

This makes h' a non-negative polynomial of degree $2L$. The integral is analytically tractable, and makes h an increasing polynomial of degree $2L + 1$. For $L = 0$, h' is constant, so h reduces to an affine scalar bijection.

In these approaches, the derivative of the bijection can just be read off. However, the inverse is not analytically computable in general. In practice, we can use bisection search to compute the inverse numerically.

23.2.2.5 Splines

Another way to construct monotonic scalar functions is using **splines**. These are piecewise-polynomial or piecewise-rational functions, parameterized in terms of $K + 1$ **knots** (u_k, x_k) through which the spline passes. That is, we set $h(u_k) = x_k$, and define h on the interval (u_{k-1}, u_k) by interpolating

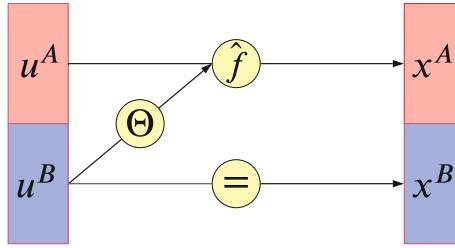


Figure 23.2: Illustration of a coupling layer $\mathbf{x} = f(\mathbf{u})$. A bijection, with parameters determined by \mathbf{u}^B , is applied to \mathbf{u}^A to generate \mathbf{x}^A ; meanwhile $\mathbf{x}^B = \mathbf{u}^B$ is passed through unchanged, so the mapping can be inverted. From Figure 3 of [KPB19]. Used with kind permission of Ivan Kobyzev.

from x_{k-1} to x_k with a polynomial or rational function (ratio of two polynomials). By increasing the number of knots we can create arbitrarily flexible monotonic functions.

Different ways to interpolate between knots give different types of spline. A simple choice is to interpolate linearly [Mül+19a]. However, this makes the derivative discontinuous at the knots. Interpolating with quadratic polynomials [Mül+19a] gives enough flexibility to make the derivative continuous. Interpolating with cubic polynomials [Dur+19], ratios of linear polynomials [DEL20], or ratios of quadratic polynomials [DBP19] allows the derivatives at the knots to be arbitrary parameters.

The spline is strictly increasing if we take $u_{k-1} < u_k$, $x_{k-1} < x_k$, and make sure the interpolation between knots is itself increasing. Depending on the flexibility on the interpolating function, more than one interpolation may exist; in practice we choose one that is guaranteed to be always increasing (see references above for details).

An advantage of splines is that they can be inverted analytically if the interpolating functions only contain low-degree polynomials. In this case, we compute $u = h^{-1}(x)$ as follows: first, we use binary search to locate the interval (x_{k-1}, x_k) in which x lies; then, we analytically solve the resulting low-degree polynomial for u .

23.2.3 Coupling flows

In this section we describe coupling flows, which allow us to model dependencies between dimensions using arbitrary non-linear functions (such as deep neural networks). Consider a partition of the input $\mathbf{u} \in \mathbb{R}^D$ into two subspaces, $(\mathbf{u}^A, \mathbf{u}^B) \in \mathbb{R}^d \times \mathbb{R}^{D-d}$, where d is an integer between 1 and $D - 1$. Assume a bijection $\hat{\mathbf{f}}(\cdot; \boldsymbol{\theta}) : \mathbb{R}^d \rightarrow \mathbb{R}^d$ parameterized by $\boldsymbol{\theta}$ and acting on the subspace \mathbb{R}^d . We define the function $\mathbf{f} : \mathbb{R}^D \rightarrow \mathbb{R}^D$ given by $\mathbf{x} = \mathbf{f}(\mathbf{u})$ as follows:

$$\mathbf{x}^A = \hat{\mathbf{f}}(\mathbf{u}^A; \boldsymbol{\theta}(\mathbf{u}^B)) \tag{23.12}$$

$$\mathbf{x}^B = \mathbf{u}^B. \tag{23.13}$$

See Figure 23.2 for an illustration. The function \mathbf{f} is called a **coupling layer** [DKB15; DSDB17], because it “couples” \mathbf{u}^A and \mathbf{u}^B together though $\hat{\mathbf{f}}$ and $\boldsymbol{\theta}$. We refer to flows consisting of coupling layers as **coupling flows**.

The parameters of $\hat{\mathbf{f}}$ are computed by $\boldsymbol{\theta} = \Theta(\mathbf{u}^B)$, where Θ is an *arbitrary* function called the **conditioner**. Unlike affine flows, which mix dimensions linearly, and elementwise flows, which do not mix dimensions at all, coupling flows can mix dimensions with a flexible non-linear conditioner Θ . In practice we often implement Θ as a deep neural network; any architecture can be used, including MLPs, CNNs, ResNets, etc.

The coupling layer \mathbf{f} is *invertible*, and its inverse is given by $\mathbf{u} = \mathbf{f}^{-1}(\mathbf{x})$, where

$$\mathbf{u}^A = \hat{\mathbf{f}}^{-1}(\mathbf{x}^A; \Theta(\mathbf{x}^B)) \quad (23.14)$$

$$\mathbf{u}^B = \mathbf{x}^B. \quad (23.15)$$

That is, \mathbf{f}^{-1} is given by simply replacing $\hat{\mathbf{f}}$ with $\hat{\mathbf{f}}^{-1}$. Because \mathbf{x}^B does not depend on \mathbf{u}^A , the Jacobian of \mathbf{f} is block triangular:

$$\mathbf{J}(\mathbf{f}) = \begin{pmatrix} \partial \mathbf{x}^A / \partial \mathbf{u}^A & \partial \mathbf{x}^A / \partial \mathbf{u}^B \\ \partial \mathbf{x}^B / \partial \mathbf{u}^A & \partial \mathbf{x}^B / \partial \mathbf{u}^B \end{pmatrix} = \begin{pmatrix} \mathbf{J}(\hat{\mathbf{f}}) & \partial \mathbf{x}^A / \partial \mathbf{u}^B \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \quad (23.16)$$

Thus, $\det \mathbf{J}(\mathbf{f})$ is equal to $\det \mathbf{J}(\hat{\mathbf{f}})$.

We often define $\hat{\mathbf{f}}$ to be an elementwise bijection, so that $\hat{\mathbf{f}}^{-1}$ and $\det \mathbf{J}(\hat{\mathbf{f}})$ are easy to compute. That is, we define:

$$\hat{\mathbf{f}}(\mathbf{u}^A; \boldsymbol{\theta}) = (h(u_1^A; \boldsymbol{\theta}_1), \dots, h(u_d^A; \boldsymbol{\theta}_d)), \quad (23.17)$$

where $h(\cdot; \boldsymbol{\theta}_i)$ is a scalar bijection parameterized by $\boldsymbol{\theta}_i$. Any of the scalar bijections described in Section 23.2.2 can be used here. For example, $h(\cdot; \boldsymbol{\theta}_i)$ can be an affine bijection with $\boldsymbol{\theta}_i$ its scale and shift parameters (Section 23.2.2.1); or it can be a monotonic MLP with $\boldsymbol{\theta}_i$ its weights and biases (Section 23.2.2.3); or it can be a monotonic spline with $\boldsymbol{\theta}_i$ its knot coordinates (Section 23.2.2.5).

There are many ways to define the partition of \mathbf{u} into $(\mathbf{u}^A, \mathbf{u}^B)$. A simple way is just to partition \mathbf{u} into two halves. We can also exploit spatial structure in the partitioning. For example, if \mathbf{u} is an image, we can partition its pixels using a ‘‘checkerboard’’ pattern, where pixels in ‘‘black squares’’ are in \mathbf{u}^A and pixels in ‘‘white squares’’ are in \mathbf{u}^B [DSDB17]. Since only part of the input is transformed by each coupling layer, in practice we typically employ different partitions along a coupling flow, to ensure all variables get transformed and are given the opportunity to interact.

Finally, if $\hat{\mathbf{f}}$ is an elementwise bijection, we can implement arbitrary partitions easily using a binary mask \mathbf{b} as follows:

$$\mathbf{x} = \mathbf{b} \odot \mathbf{u} + (1 - \mathbf{b}) \odot \hat{\mathbf{f}}(\mathbf{u}; \Theta(\mathbf{b} \odot \mathbf{u})), \quad (23.18)$$

where \odot denotes elementwise multiplication. A value of 0 in \mathbf{b} indicates that the corresponding element in \mathbf{u} is transformed (belongs to \mathbf{u}^A); a value of 1 indicates that it remains unchanged (belongs to \mathbf{u}^B).

As an example, we fit a masked coupling flow, created from piecewise rational quadratic splines, to the two moons dataset. Samples from each layer of the fitted model are shown in Figure 23.3.

23.2.4 Autoregressive flows

In this section we discuss **autoregressive flows**, which are flows composed of autoregressive bijections. Like coupling flows, autoregressive flows allow us to model dependencies between variables with arbitrary non-linear functions, such as deep neural networks.

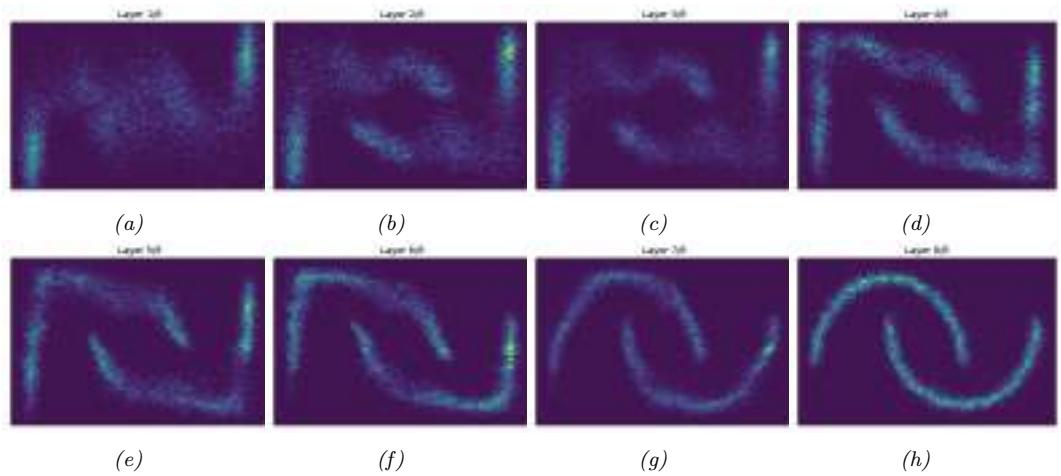


Figure 23.3: (a) Two moons dataset. (b) Samples from a normalizing flow fit to this dataset. Generated by [two_moons_nsf_normalizing_flow.ipynb](#).

Suppose the input \mathbf{u} contains D scalar elements, that is, $\mathbf{u} = (u_1, \dots, u_D) \in \mathbb{R}^D$. We define an **autoregressive bijection** $\mathbf{f} : \mathbb{R}^D \rightarrow \mathbb{R}^D$, its output denoted by $\mathbf{x} = (x_1, \dots, x_D) \in \mathbb{R}^D$, as follows:

$$x_i = h(u_i; \Theta_i(\mathbf{x}_{1:i-1})), \quad i = 1, \dots, D. \quad (23.19)$$

Each output x_i depends on the corresponding input u_i and all previous outputs $\mathbf{x}_{1:i-1} = (x_1, \dots, x_{i-1})$. The function $h(\cdot; \boldsymbol{\theta}) : \mathbb{R} \rightarrow \mathbb{R}$ is a scalar bijection (for example, one of those described in Section 23.2.2), and is parameterized by $\boldsymbol{\theta}$. The function Θ_i is a conditioner that outputs the parameters $\boldsymbol{\theta}_i$ that yield x_i , given all previous outputs $\mathbf{x}_{1:i-1}$. Like in coupling flows, Θ_i can be an arbitrary non-linear function, and is often parameterized as a deep neural network.

Because h is invertible, \mathbf{f} is also invertible, and its inverse is given by:

$$u_i = h^{-1}(x_i; \Theta_i(\mathbf{x}_{1:i-1})), \quad i = 1, \dots, D. \quad (23.20)$$

An important property of \mathbf{f} is that each output x_i depends on $\mathbf{u}_{1:i} = (u_1, \dots, u_i)$, but not on $\mathbf{u}_{i+1:D} = (u_{i+1}, \dots, u_D)$; as a result, the partial derivative $\partial x_i / \partial u_j$ is identically zero whenever $j > i$. Therefore, the Jacobian matrix $\mathbf{J}(\mathbf{f})$ is triangular, and its determinant is simply the product of its diagonal entries:

$$\det \mathbf{J}(\mathbf{f}) = \prod_{i=1}^D \frac{\partial x_i}{\partial u_i} = \prod_{i=1}^D \frac{dh}{du_i}. \quad (23.21)$$

In other words, the autoregressive structure of \mathbf{f} leads to a Jacobian determinant that can be computed efficiently in $O(D)$ time.

Although invertible, autoregressive bijections are computationally asymmetric: evaluating \mathbf{f} is inherently sequential, whereas evaluating \mathbf{f}^{-1} is inherently parallel. That is because we need $\mathbf{x}_{1:i-1}$ to

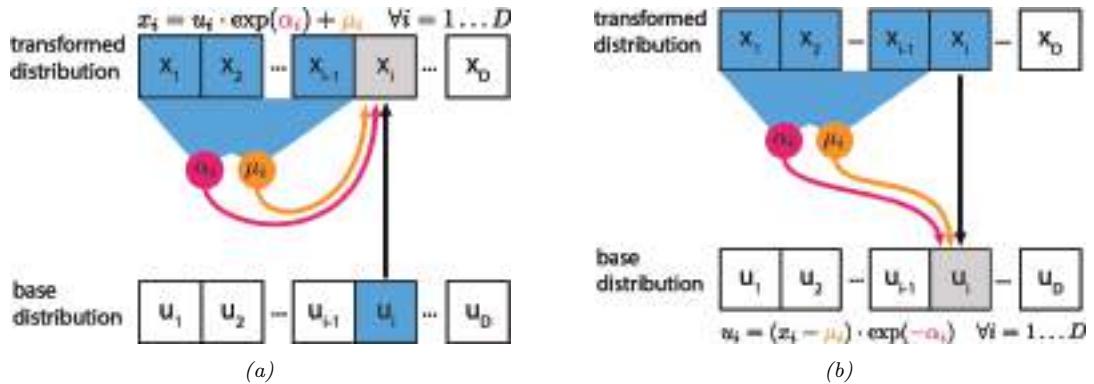


Figure 23.4: (a) Affine autoregressive flow with one layer. In this figure, \mathbf{u} is the input to the flow (sample from the base distribution) and \mathbf{x} is its output (sample from the transformed distribution). (b) Inverse of the above. From [Jan18]. Used with kind permission of Eric Jang.

compute x_i ; therefore, computing the components of \mathbf{x} must be done sequentially, by first computing x_1 , then using it to compute x_2 , then using x_1 and x_2 to compute x_3 , and so on. On the other hand, computing the inverse can be done in parallel for each u_i , since \mathbf{u} does not appear on the right-hand side of Equation (23.20). Hence, in practice it is often faster to compute \mathbf{f}^{-1} than to compute \mathbf{f} , assuming h and h^{-1} have similar computational cost.

23.2.4.1 Affine autoregressive flows

For a concrete example, we can take h to be an affine scalar bijection (Section 23.2.2.1) parameterized by a log scale α and a bias μ . Such autoregressive flows are known as **affine autoregressive flows**. The parameters of the i 'th component, α_i and μ_i , are functions of $\mathbf{x}_{1:i-1}$, so \mathbf{f} takes the following form:

$$x_i = u_i \exp(\alpha_i(\mathbf{x}_{1:i-1})) + \mu_i(\mathbf{x}_{1:i-1}). \quad (23.22)$$

This is illustrated in Figure 23.4(a). We can invert this by

$$u_i = (x_i - \mu_i(\mathbf{x}_{1:i-1})) \exp(-\alpha_i(\mathbf{x}_{1:i-1})). \quad (23.23)$$

This is illustrated in Figure 23.4(b). Finally, we can calculate the log absolute Jacobian determinant by

$$\log |\det \mathbf{J}(\mathbf{f})| = \log \left| \prod_{i=1}^D \exp(\alpha_i(\mathbf{x}_{1:i-1})) \right| = \sum_{i=1}^D \alpha_i(\mathbf{x}_{1:i-1}). \quad (23.24)$$

Let us look at an example of an affine autoregressive flow on a 2d density estimation problem. Consider an affine autoregressive flow $\mathbf{x} = (x_1, x_2) = \mathbf{f}(\mathbf{u})$, where $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and \mathbf{f} is a single autoregressive bijection. Since x_1 is an affine transformation of $u_1 \sim \mathcal{N}(0, 1)$, it is Gaussian with mean μ_1 and standard deviation $\sigma_1 = \exp \alpha_1$. Similarly, if we consider x_1 fixed, x_2 is an affine

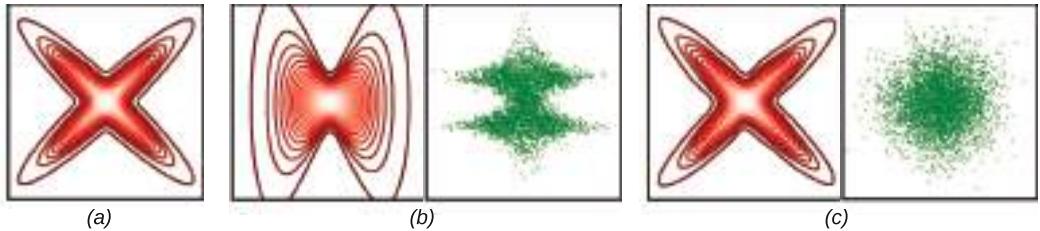


Figure 23.5: Density estimation with affine autoregressive flows, using a Gaussian base distribution. (a) True density. (b) Estimated density using a single autoregressive layer with ordering (x_1, x_2) . On the left (contour plot) we show $p(\mathbf{x})$. On the right (green dots) we show samples of $\mathbf{u} = \mathbf{f}^{-1}(\mathbf{x})$, where \mathbf{x} is sampled from the true density. (c) Same as (b), but using 5 autoregressive layers and reversing the variable ordering after each layer. Adapted from Figure 1 of [PPM17]. Used with kind permission of Iain Murray.

transformation of $u_2 \sim \mathcal{N}(0, 1)$, so it is *conditionally* Gaussian with mean $\mu_2(x_1)$ and standard deviation $\sigma_2(x_1) = \exp \alpha_2(x_1)$. Thus, a single affine autoregressive bijection will always produce a distribution with Gaussian conditionals, that is, a distribution of the following form:

$$p(x_1, x_2) = p(x_1) p(x_2|x_1) = \mathcal{N}(x_1|\mu_1, \sigma_1^2) \mathcal{N}(x_2|\mu_2(x_1), \sigma_2(x_1)^2) \quad (23.25)$$

This result generalizes to an arbitrary number of dimensions D .

A single affine bijection is not very powerful, regardless of how flexible the functions $\alpha_2(x_1)$ and $\mu_2(x_1)$ are. For example, suppose we want to fit the cross-shaped density shown in Figure 23.5(a) with such a flow. The resulting maximum-likelihood fit is shown in Figure 23.5(b). The red contours show the predictive distribution, $\hat{p}(\mathbf{x})$, which clearly fails to capture the true distribution. The green dots show transformed versions of the data samples, $p(\mathbf{u})$; we see that this is far from the Gaussian base distribution.

Fortunately, we can obtain a better fit by composing multiple autoregressive bijections (layers), and reversing the order of the variables after each layer. For example, Figure 23.5(c) shows the results of an affine autoregressive flow with 5 layers applied to the same problem. The red contours show that we have matched the empirical distribution, and the green dots show we have matched the Gaussian base distribution.

Note that another way to obtain a better fit is to replace the affine bijection h with a more flexible one, such as a monotonic MLP (Section 23.2.2.3) or a monotonic spline (Section 23.2.2.5).

23.2.4.2 Masked autoregressive flows

As we have seen, the conditioners Θ_i can be arbitrary non-linear functions. The most straightforward way to parameterize them is separately for each i , for example by using D separate neural networks. However, this can be parameter-inefficient for large D .

In practice, we often share parameters between conditioners by combining them into a single model Θ that takes in \mathbf{x} and outputs $(\theta_1, \dots, \theta_D)$. For the bijection to remain autoregressive, we must constrain Θ so that θ_i depends only on $\mathbf{x}_{1:i-1}$ and not on $\mathbf{x}_{i:D}$. One way to achieve this is to start with an arbitrary neural network (an MLP, a CNN, a ResNet, etc.), and drop connections (for example, by zeroing out weights) until θ_i is only a function of $\mathbf{x}_{1:i-1}$.

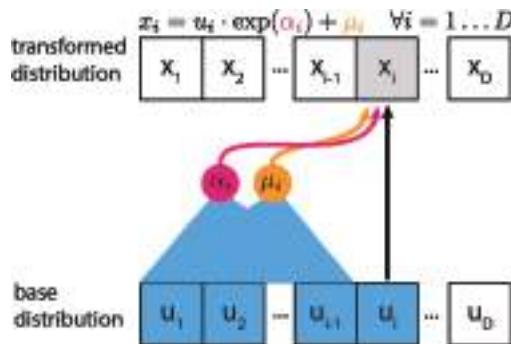


Figure 23.6: Inverse autoregressive flow that uses affine scalar bijections. In this figure, \mathbf{u} is the input to the flow (sample from the base distribution) and \mathbf{x} is its output (sample from the transformed distribution) From [Jan18]. Used with kind permission of Eric Jang.

An example of this approach is the **masked autoregressive flow (MAF)** model of [PPM17]. This model is an affine autoregressive flow combined with permutation layers, as we described in Section 23.2.4.1. MAF implements the combined conditioner Θ as follows: it starts with an MLP, and then multiplies (elementwise) the weight matrix of each layer with a binary mask of the same size (different masks are used for different layers). The masks are constructed using the method of [Ger+15]. This ensures that all computational paths from x_j to θ_i are zeroed out whenever $j > i$, effectively making θ_i only a function of $\mathbf{x}_{1:i-1}$. Still, evaluating the masked conditioner Θ has the same computational cost as evaluating the original (unmasked) MLP.

The key advantage of MAF (and of related models) is that, given \mathbf{x} , all parameters $(\theta_1, \dots, \theta_D)$ can be computed efficiently with one neural network evaluation, so the computation of the inverse \mathbf{f}^{-1} is fast. Thus, we can efficiently evaluate the probability density of the flow model for arbitrary datapoints. However, in order to compute \mathbf{f} , the conditioner Θ must be called a total of D times, since not all entries of \mathbf{x} are available to start with. Thus, generating new samples from the flow is D times more expensive than evaluating its probability density function. This makes MAF suitable for density estimation, but less so for data generation.

23.2.4.3 Inverse autoregressive flows

As we have seen, the parameters θ_i that yield the i 'th output x_i are functions of the previous outputs $\mathbf{x}_{1:i-1}$. This ensures that the Jacobian $\mathbf{J}(\mathbf{f})$ is triangular, and so its determinant is efficient to compute.

However, there is another possibility: we can make θ_i a function of the previous *inputs* instead, that is, a function of $\mathbf{u}_{1:i-1}$. This leads to the following bijection, which is known as **inverse autoregressive**:

$$x_i = h(u_i; \theta_i(\mathbf{u}_{1:i-1})), \quad i = 1, \dots, D. \quad (23.26)$$

Like its autoregressive counterpart, this bijection has a triangular Jacobian whose determinant is also given by $\det \mathbf{J}(\mathbf{f}) = \prod_{i=1}^D \frac{dh}{du_i}$. Figure 23.6 illustrates an inverse autoregressive flow, for the case where h is affine.

To see why this bijection is called “inverse autoregressive”, compare Equation (23.26) with Equation (23.20). The two formulas differ only notationally: we can get from one to the other by swapping \mathbf{u} with \mathbf{x} and h with h^{-1} . In other words, the inverse autoregressive bijection corresponds to a direct parameterization of the inverse of an autoregressive bijection.

Since inverse autoregressive bijections swap the forwards and inverse directions of their autoregressive counterparts, they also swap their computational properties. This means that the forward direction \mathbf{f} of an inverse autoregressive flow is inherently parallel and therefore fast, whereas its inverse direction \mathbf{f}^{-1} is inherently sequential and therefore slow.

An example of an inverse autoregressive flow is their namesake **IAF** model of [Kin+16]. IAF uses affine scalar bijections, masked conditioners, and permutation layers, so it is precisely the inverse of the MAF model described in Section 23.2.4.2. Using IAF, we can generate \mathbf{u} in parallel from the base distribution (using, for example, a diagonal Gaussian), and then sample each element of \mathbf{x} in parallel. However, evaluating $p(\mathbf{x})$ for an arbitrary datapoint \mathbf{x} is slow, because we have to evaluate each element of \mathbf{u} sequentially. Fortunately, evaluating the likelihood of samples generated from IAF (as opposed to externally provided samples) incurs no additional cost, since in this case the u_i terms will already have been computed.

Although not so suitable for density estimation or maximum-likelihood training, IAFs are well-suited for parameterizing variational posteriors in variational inference. This is because in order to estimate the variational lower bound (ELBO), we only need samples from the variational posterior and their associated probability densities, both of which are efficient to obtain. See Section 23.1.2.2 for details.

Another useful application of IAFs is training them to mimic models whose probability density is fast to evaluate but which are slow to sample from. A notable example is the **parallel wavenet** model of [Oor+18]. This model is an IAF p_s that is trained to mimic a pretrained wavenet model p_t by minimizing the KL divergence $D_{\text{KL}}(p_s \parallel p_t)$. This KL can be easily estimated by first sampling from p_s and then evaluating $\log p_s$ and $\log p_t$ at those samples, operations which are all efficient for these models. After training, we obtain an IAF that can generate audio of similar quality as the original wavenet, but can do so much faster.

23.2.4.4 Connection with autoregressive models

Autoregressive flows can be thought of as generalizing autoregressive models of continuous random variables, discussed in Section 22.1. Specifically, any continuous autoregressive model can be reparameterized as a one-layer autoregressive flow, as we describe below.

Consider a general autoregressive model over a continuous random variable $\mathbf{x} = (x_1, \dots, x_D) \in \mathbb{R}^D$ written as

$$p(\mathbf{x}) = \prod_{i=1}^D p_i(x_i | \boldsymbol{\theta}_i) \quad \text{where} \quad \boldsymbol{\theta}_i = \Theta_i(\mathbf{x}_{1:i-1}). \quad (23.27)$$

In the above expression, $p_i(x_i | \boldsymbol{\theta}_i)$ is the i 'th conditional distribution of the autoregressive model, whose parameters $\boldsymbol{\theta}_i$ are arbitrary functions of the previous variables $\mathbf{x}_{1:i-1}$. For example, $p_i(x_i | \boldsymbol{\theta}_i)$ can be a mixture of one-dimensional Gaussian distributions, with $\boldsymbol{\theta}_i$ representing the collection of its means, variances, and mixing coefficients.

Now consider sampling a vector \mathbf{x} from the autoregressive model, which can be done by sampling

one element at a time as follows:

$$x_i \sim p_i(x_i | \Theta_i(\mathbf{x}_{1:i-1})) \quad \text{for } i = 1, \dots, D. \quad (23.28)$$

Each conditional can be sampled from using inverse transform sampling (Section 11.3.1). Let $U(0, 1)$ be the uniform distribution on the interval $[0, 1]$, and let $\text{CDF}_i(x_i | \theta_i)$ be the cumulative distribution function of the i 'th conditional. Sampling can be written as:

$$x_i = \text{CDF}_i^{-1}(u_i | \Theta_i(\mathbf{x}_{1:i-1})) \quad \text{where } u_i \sim U(0, 1). \quad (23.29)$$

Comparing the above expression with the definition of an autoregressive bijection in Equation (23.19), we see that the autoregressive model has been expressed as a one-layer autoregressive flow whose base distribution is uniform on $[0, 1]^D$ and whose scalar bijections correspond to the inverse conditional cdf's. Viewing autoregressive models as flows this way has an important advantage, namely that it allows us to increase the flexibility of an autoregressive model by composing multiple instances of it in a flow, without sacrificing the overall tractability.

23.2.5 Residual flows

A residual network is a composition of **residual connections**, which are functions of the form $\mathbf{f}(\mathbf{u}) = \mathbf{u} + \mathbf{F}(\mathbf{u})$. The function $\mathbf{F} : \mathbb{R}^D \rightarrow \mathbb{R}^D$ is called the **residual block**, and it computes the difference between the output and the input, $\mathbf{f}(\mathbf{u}) - \mathbf{u}$.

Under certain conditions on \mathbf{F} , the residual connection \mathbf{f} becomes invertible. We will refer to flows composed of invertible residual connections as **residual flows**. In the following, we describe two ways the residual block \mathbf{F} can be constrained so that the residual connection \mathbf{f} is invertible.

23.2.5.1 Contractive residual blocks

One way to ensure the residual connection is invertible is to choose the residual block to be a contraction. A contraction is a function \mathbf{F} whose Lipschitz constant is less than 1; that is, there exists $0 \leq L < 1$ such that for all \mathbf{u}_1 and \mathbf{u}_2 we have:

$$\|\mathbf{F}(\mathbf{u}_1) - \mathbf{F}(\mathbf{u}_2)\| \leq L\|\mathbf{u}_1 - \mathbf{u}_2\|. \quad (23.30)$$

The invertibility of $\mathbf{f}(\mathbf{u}) = \mathbf{u} + \mathbf{F}(\mathbf{u})$ can be shown as follows. Consider the mapping $\mathbf{g}(\mathbf{u}) = \mathbf{x} - \mathbf{F}(\mathbf{u})$. Because \mathbf{F} is a contraction, \mathbf{g} is also a contraction. So, by Banach's fixed-point theorem, \mathbf{g} has a unique fixed point \mathbf{u}_* . Hence we have

$$\mathbf{u}_* = \mathbf{x} - \mathbf{F}(\mathbf{u}_*) \quad (23.31)$$

$$\Rightarrow \mathbf{u}_* + \mathbf{F}(\mathbf{u}_*) = \mathbf{x} \quad (23.32)$$

$$\Rightarrow \mathbf{f}(\mathbf{u}_*) = \mathbf{x}. \quad (23.33)$$

Because \mathbf{u}_* is unique, it follows that $\mathbf{u}_* = \mathbf{f}^{-1}(\mathbf{x})$.

An example of a residual flow with contractive residual blocks is the **iResNet** model of [Beh+19]. The residual blocks of iResNet are convolutional neural networks, that is, compositions of convolutional layers with non-linear activation functions. Because the Lipschitz constant of a composition is less or equal to the product of the Lipschitz constants of the individual functions, it is enough to ensure the

convolutions are contractive, and to use increasing activation functions with slope less or equal to 1. The iResNet model ensures the convolutions are contractive by applying spectral normalization to their weights [Miy+18a].

In general, there is no analytical expression for the inverse \mathbf{f}^{-1} . However, we can approximate $\mathbf{f}^{-1}(\mathbf{x})$ using the following iterative procedure:

$$\mathbf{u}_n = \mathbf{g}(\mathbf{u}_{n-1}) = \mathbf{x} - \mathbf{F}(\mathbf{u}_{n-1}). \quad (23.34)$$

Banach's fixed-point theorem guarantees that the sequence $\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \dots$ will converge to $\mathbf{u}_* = \mathbf{f}^{-1}(\mathbf{x})$ for any choice of \mathbf{u}_0 , and it will do so at a rate of $O(L^n)$, where L is the Lipschitz constant of \mathbf{g} (which is the same as the Lipschitz constant of \mathbf{F}). In practice, it is convenient to choose $\mathbf{u}_0 = \mathbf{x}$.

In addition, there is no analytical expression for the Jacobian determinant, whose exact computation costs $O(D^3)$. However, there is a computationally efficient stochastic estimator of the log Jacobian determinant. The idea is to express the log Jacobian determinant as a power series. Using the fact that $\mathbf{f}(\mathbf{x}) = \mathbf{x} + \mathbf{F}(\mathbf{x})$, we have

$$\log |\det \mathbf{J}(\mathbf{f})| = \log |\det(\mathbf{I} + \mathbf{J}(\mathbf{F}))| = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \text{tr}[\mathbf{J}(\mathbf{F})^k]. \quad (23.35)$$

This power series converges when the matrix norm of $\mathbf{J}(\mathbf{F})$ is less than 1, which here is guaranteed exactly because \mathbf{F} is a contraction. The trace of $\mathbf{J}(\mathbf{F})^k$ can be efficiently approximated using Jacobian-vector products via the **Hutchinson trace estimator** [Ski89; Hut89; Mey+21]:

$$\text{tr}[\mathbf{J}(\mathbf{F})^k] \approx \mathbf{v}^\top \mathbf{J}(\mathbf{F})^k \mathbf{v}, \quad (23.36)$$

where \mathbf{v} is a sample from a distribution with zero mean and unit covariance, such as $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Finally, the infinite series can be approximated by a finite one either by truncation [Beh+19], which unfortunately yields a biased estimator, or by employing the **Russian roulette estimator** [Che+19], which is unbiased.

23.2.5.2 Residual blocks with low-rank Jacobian

There is an efficient way of computing the determinant of a matrix which is a low-rank perturbation of an identity matrix. Suppose \mathbf{A} and \mathbf{B} are matrices, where \mathbf{A} is $D \times M$ and \mathbf{B} is $M \times D$. The following formula is known as the **Weinstein-Aronszajn identity**², and is a special case of the more general **matrix determinant lemma**:

$$\det(\mathbf{I}_D + \mathbf{AB}) = \det(\mathbf{I}_M + \mathbf{BA}). \quad (23.37)$$

We write \mathbf{I}_D and \mathbf{I}_M for the $D \times D$ and $M \times M$ identity matrices respectively. The significance of this formula is that it turns a $D \times D$ determinant that costs $O(D^3)$ into an $M \times M$ determinant that costs $O(M^3)$. If M is smaller than D , this saves computation.

With some restrictions on the residual block $\mathbf{F} : \mathbb{R}^D \rightarrow \mathbb{R}^D$, we can apply this formula to compute the determinant of a residual connection efficiently. The trick is to create a bottleneck inside \mathbf{F} . We do that by defining $\mathbf{F} = \mathbf{F}_2 \circ \mathbf{F}_1$, where $\mathbf{F}_1 : \mathbb{R}^D \rightarrow \mathbb{R}^M$, $\mathbf{F}_2 : \mathbb{R}^M \rightarrow \mathbb{R}^D$ and $M \ll D$. The chain

2. See https://en.wikipedia.org/wiki/Weinstein-Aronszajn_identity.

rule gives $\mathbf{J}(\mathbf{F}) = \mathbf{J}(\mathbf{F}_2)\mathbf{J}(\mathbf{F}_1)$, where $\mathbf{J}(\mathbf{F}_2)$ is $D \times M$ and $\mathbf{J}(\mathbf{F}_1)$ is $M \times D$. Now we can apply our determinant formula as follows:

$$\det \mathbf{J}(\mathbf{f}) = \det(\mathbf{I}_D + \mathbf{J}(\mathbf{F})) = \det(\mathbf{I}_D + \mathbf{J}(\mathbf{F}_2)\mathbf{J}(\mathbf{F}_1)) = \det(\mathbf{I}_M + \mathbf{J}(\mathbf{F}_1)\mathbf{J}(\mathbf{F}_2)). \quad (23.38)$$

Since the final determinant costs $O(M^3)$, we can make the Jacobian determinant efficient by reducing M , that is, by narrowing the bottleneck.

An example of the above is the **planar flow** of [RM15]. In this model, each residual block is an MLP with one hidden layer and one hidden unit. That is,

$$\mathbf{f}(\mathbf{u}) = \mathbf{u} + \mathbf{v}\sigma(\mathbf{w}^\top \mathbf{u} + b), \quad (23.39)$$

where $\mathbf{v} \in \mathbb{R}^D$, $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$ are the parameters, and σ is the activation function. The residual block is the composition of $\mathbf{F}_1(\mathbf{u}) = \mathbf{w}^\top \mathbf{u} + b$ and $\mathbf{F}_2(z) = \mathbf{v}\sigma(z)$, so $M = 1$. Their Jacobians are $\mathbf{J}(\mathbf{F}_1)(\mathbf{u}) = \mathbf{w}^\top$ and $\mathbf{J}(\mathbf{F}_2)(z) = \mathbf{v}\sigma'(z)$. Substituting these in the formula for the Jacobian determinant we obtain:

$$\det \mathbf{J}(\mathbf{f})(\mathbf{u}) = 1 + \mathbf{w}^\top \mathbf{v}\sigma'(\mathbf{w}^\top \mathbf{u} + b), \quad (23.40)$$

which can be computed efficiently in $O(D)$. Other examples include the **circular flow** of [RM15] and the **Sylvester flow** of [Ber+18].

This technique gives an efficient way of computing determinants of residual connections with bottlenecks, but in general there is no guarantee that such functions are invertible. This means that invertibility must be satisfied on a case-by-case basis. For example, the planar flow is invertible when σ is the hyperbolic tangent and $\mathbf{w}^\top \mathbf{v} > -1$, but otherwise it may not be.

23.2.6 Continuous-time flows

So far we have discussed flows that consist of a sequence of bijections $\mathbf{f}_1, \dots, \mathbf{f}_N$. Starting from some input $\mathbf{x}_0 = \mathbf{u}$, this creates a sequence of outputs $\mathbf{x}_1, \dots, \mathbf{x}_N$ where $\mathbf{x}_n = \mathbf{f}_n(\mathbf{x}_{n-1})$. However, we can also have flows where the input is transformed into the final output in a continuous way. That is, we start from $\mathbf{x}_0 = \mathbf{x}(0)$, create a continuously-indexed sequence $\mathbf{x}(t)$ for $t \in [0, T]$ with some fixed T , and take $\mathbf{x}(T)$ to be the final output. Thinking of t as analogous to time, we refer to these as **continuous-time flows**.

The sequence $\mathbf{x}(t)$ is defined as the solution to a first-order ordinary differential equation (ODE) of the form:

$$\frac{d\mathbf{x}}{dt}(t) = \mathbf{F}(\mathbf{x}(t), t). \quad (23.41)$$

The function $\mathbf{F} : \mathbb{R}^D \times [0, T] \rightarrow \mathbb{R}^D$ is a time-dependent vector field that parameterizes the ODE. If we think of $\mathbf{x}(t)$ as the position of a particle in D dimensions, the vector $\mathbf{F}(\mathbf{x}(t), t)$ determines the particle's velocity at time t .

The flow (for time T) is a function $\mathbf{f} : \mathbb{R}^D \rightarrow \mathbb{R}^D$ that takes in an input \mathbf{x}_0 , solves the ODE with initial condition $\mathbf{x}(0) = \mathbf{x}_0$, and returns $\mathbf{x}(T)$. The function \mathbf{f} is a well-defined bijection if the solution to the ODE exists for all $t \in [0, T]$ and is unique. These conditions are not generally satisfied for arbitrary \mathbf{F} , but they are if $\mathbf{F}(\cdot, t)$ is Lipschitz continuous with a Lipschitz constant that does not

depend on t . That is, \mathbf{f} is a well-defined bijection if there exists a constant L such that for all $\mathbf{x}_1, \mathbf{x}_2$ and $t \in [0, T]$ we have:

$$\|\mathbf{F}(\mathbf{x}_1, t) - \mathbf{F}(\mathbf{x}_2, t)\| \leq L\|\mathbf{x}_1 - \mathbf{x}_2\|. \quad (23.42)$$

This result is a consequence of the **Picard-Lindelöf theorem** for ODEs.³ In practice, we can parameterize \mathbf{F} using any choice of model, provided the Lipschitz condition is met.

Usually the ODE cannot be solved analytically, but we can solve it approximately by discretizing it. A simple example is **Euler's method**, which corresponds to the following discretization for some small step size $\epsilon > 0$:

$$\mathbf{x}(t + \epsilon) = \mathbf{x}(t) + \epsilon\mathbf{F}(\mathbf{x}(t), t). \quad (23.43)$$

This is equivalent to a residual connection with residual block $\epsilon\mathbf{F}(\cdot, t)$, so the ODE solver can be thought of as a deep residual network with $O(T/\epsilon)$ layers. A smaller step size leads to a more accurate solution, but also to more computation. There are several other solution methods varying in accuracy and sophistication, such as those in the broader Runge-Kutta family, some of which use adaptive step sizes.

The inverse of \mathbf{f} can be easily computed by solving the ODE in reverse. That is, to compute $\mathbf{f}^{-1}(\mathbf{x}_T)$ we solve the ODE with initial condition $\mathbf{x}(T) = \mathbf{x}_T$, and return $\mathbf{x}(0)$. Unlike some other flows (such as autoregressive flows) which are more expensive to compute in one direction than in the other, continuous-time flows require the same amount of computation in either direction.

In general, there is no analytical expression for the Jacobian determinant of \mathbf{f} . However, we can express it as the solution to a separate ODE, which we can then solve numerically. First, we define $\mathbf{f}_t : \mathbb{R}^D \rightarrow \mathbb{R}^D$ to be the flow for time t , that is, the function that takes \mathbf{x}_0 , solves the ODE with initial condition $\mathbf{x}(0) = \mathbf{x}_0$ and returns $\mathbf{x}(t)$. Clearly, \mathbf{f}_0 is the identity function and $\mathbf{f}_T = \mathbf{f}$. Let us define $L(t) = \log |\det \mathbf{J}(\mathbf{f}_t)(\mathbf{x}_0)|$. Because \mathbf{f}_0 is the identity function, $L(0) = 0$, and because $\mathbf{f}_T = \mathbf{f}$, $L(T)$ gives the Jacobian determinant of \mathbf{f} that we are interested in. It can be shown that L satisfies the following ODE:

$$\frac{dL}{dt}(t) = \text{tr}[\mathbf{J}(\mathbf{F}(\cdot, t))(\mathbf{x}(t))]. \quad (23.44)$$

That is, the rate of change of L at time t is equal to the Jacobian trace of $\mathbf{F}(\cdot, t)$ evaluated at $\mathbf{x}(t)$. So we can compute $L(T)$ by solving the above ODE with initial condition $L(0) = 0$. Moreover, we can compute $\mathbf{x}(T)$ and $L(T)$ simultaneously, by combining their two ODEs into a single ODE operating on the extended space (\mathbf{x}, L) .

An example of a continuous-time flow is the **neural ODE** model of [Che+18c], which uses a neural network to parameterize \mathbf{F} . To avoid backpropagating gradients through the ODE solver, which can be computationally demanding, they use the **adjoint sensitivity method** to express the time evolution of the gradient with respect to $\mathbf{x}(t)$ as a separate ODE. Solving this ODE gives the required gradients, and can be thought of as the continuous-time analog of backpropagation.

Another example is the **FFJORD** model of [Gra+19]. This is similar to the neural ODE model, except that it uses the Hutchinson trace estimator to approximate the Jacobian trace of $\mathbf{F}(\cdot, t)$. This usage of the Hutchinson trace estimator is analogous to that in contractive residual flows (Section 23.2.5.1), and it speeds up computation in exchange for a stochastic (but unbiased) estimate.

See also Section 25.4.4, where we discuss continuous time diffusion models.

³ See https://en.wikipedia.org/wiki/Picard-Lindelöf_theorem

23.3 Applications

In this section, we highlight some applications of flows for canonical probabilistic machine learning tasks.

23.3.1 Density estimation

Flow models allow exact density computation and can be used to fit multi-modal densities to observed data. (see Figure 23.3 for an example). An early example is Gaussianization [CG00] who applied this idea to fit low-dimensional densities. Tabak and Vanden-Eijnden [TVE10] and Tabak and Turner [TT13] introduced the modern idea of flows (including the term ‘normalizing flows’), describing a flow as a composition of simpler maps. Deep density models [RA13] was one of the first to use neural networks for flows to parameterize high-dimensional densities. There has been a rich line of follow-up work including NICE [DKB15] and Real NVP [DSDB17]. (NVP stands for “non-volume-preserving”, which refers to the fact that the Jacobian of the transform is not unity.) Masked autoregressive flows (Section 23.2.4.2) further improved performance on unconditional and conditional density estimation tasks.

Flows can be used for *hybrid models* which model the joint density of inputs and targets $p(\mathbf{x}, y)$, as opposed to discriminative classification models which just model the conditional $p(y|\mathbf{x})$ and density models which just model the marginal $p(\mathbf{x})$. Nalisnick et al. [Nal+19b] proposed a flow-based hybrid model using invertible mappings for representation learning and showed that the joint density $p(\mathbf{x}, y)$ can be computed efficiently, which can be useful for downstream tasks such as anomaly detection, semi-supervised learning and selective classification. Flow-based hybrid models are memory-efficient since most of the parameters are in the invertible representation which are shared between the discriminative and generative models; furthermore, the density $p(\mathbf{x}, y)$ can be computed in a single forwards pass leading to computational savings. Residual flows [Che+19] use invertible residual mappings [Beh+19] for hybrid modeling which further improves performance. Flows have also been used to fit densities to embeddings [Zha+20b; CZG20] for anomaly detection tasks.

23.3.2 Generative modeling

Another task is generation, which involves generating novel samples from a fitted model $p^*(\mathbf{x})$. Generation is a popular downstream task for normalizing flows, which have been applied for different data modalities including images, video, audio, text, and structured objects such as graphs and point clouds. Images are arguably the most popular modality for deep generative models: GLOW [KD18b] was one of the first flow-based models to generate compelling high-dimensional images, and has been extended to video to produce RGB frames [Kum+19b]; residual flows [Che+19] have also been shown to produce sharp images.

Oord et al. [Oor+18] used flows for audio synthesis by distilling WaveNet into an IAF (Section 23.2.4.3), which enables faster sampling than WaveNet. Other flow models for audio include WaveFLOW [PVC19] and FlowWaveNet [Kim+19], which directly speed up WaveNet using coupling layers.

Flows have been also used for text. Tran et al. [Tra+19] define a discrete flow over a vocabulary for language-modeling tasks. Another popular approach is to define a latent variable model with discrete observation space but a continuous latent space. For example, Ziegler and Rush [ZR19a] use

normalizing flows in latent space for language modeling.

23.3.3 Inference

Normalizing flows have been used for probabilistic inference. Rezende and Mohamed [RM15] popularized normalizing flows in machine learning, and showed how they can be used for modeling variational posterior distributions in latent variable models. Various extensions such as Householder flows [TW16], inverse autoregressive flows [Kin+16], multiplicative normalizing flows [LW17], and Sylvester flows [Ber+18] have been proposed for modeling the variational posterior for latent variable models, as well as posteriors for Bayesian neural networks.

Flows have been used as complex proposal distributions for importance sampling; examples include neural importance sampling [Mül+19b] and Boltzmann generators [Noé+19]. Hoffman et al. [Hof+19] used flows to improve the performance of Hamiltonian Monte Carlo (Section 12.5) by defining bijective transformations to transform random variables to simpler distributions and performing HMC in that space instead.

Finally, flows can be used in the context of simulation-based inference, where the likelihood function of the parameters is not available, but simulating data from the model is possible. The main idea is to train a flow on data simulated from the model in order to approximate the posterior distribution or the likelihood function. The flow model can also be used to guide simulations in order to make inference more efficient [PSM19; GNM19]. This approach has been used for inference of simulation models in cosmology [Als+19] and computational neuroscience [Gon+20].

24 Energy-based models

This chapter is co-authored with Yang Song and Durk Kingma.

24.1 Introduction

We have now seen several ways of defining deep generative models, including VAEs (Chapter 21), autoregressive models (Chapter 22), and normalizing flows (Chapter 23). All of the above models can be formulated in terms of directed graphical models (Chapter 4), where we generate the data one step at a time, using locally normalized distributions. In some cases, it is easier to specify a distribution in terms of a set of constraints that valid samples must satisfy, rather than a generative process. This can be done using an undirected graphical model (Chapter 4).

Energy-based models or **EBM** can be written as a Gibbs distribution as follows:

$$p_{\theta}(\mathbf{x}) = \frac{\exp(-\mathcal{E}_{\theta}(\mathbf{x}))}{Z_{\theta}} \quad (24.1)$$

where $\mathcal{E}_{\theta}(\mathbf{x}) \geq 0$ is known as the **energy function** with parameters θ , and Z_{θ} is the **partition function**:

$$Z_{\theta} = \int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) \, d\mathbf{x} \quad (24.2)$$

This is constant wrt \mathbf{x} but is a function of θ . Since EBMs do not usually make any Markov assumptions (unlike graphical models), evaluating this integral is usually intractable. Consequently we usually need to use approximate methods, such as annealed importance sampling, discussed in Section 11.5.4.1.

The advantage of an EBM over other generative models is that the energy function can be any kind of function that returns a non-negative scalar; it does not need to integrate to 1. This allows one to use a variety of neural network architectures for defining the energy. As such, EBMs have found wide applications in many fields of machine learning, including image generation [Ngi+11; Xie+16; DM19b], discriminative learning [Gra+20b], natural processing [Mik+13; Den+20], density estimation [Wen+19a; Son+19], and reinforcement learning [Haa+17; Haa+18a], to list a few. (More examples can be found at <https://github.com/yataobian/awesome-ebm>.)

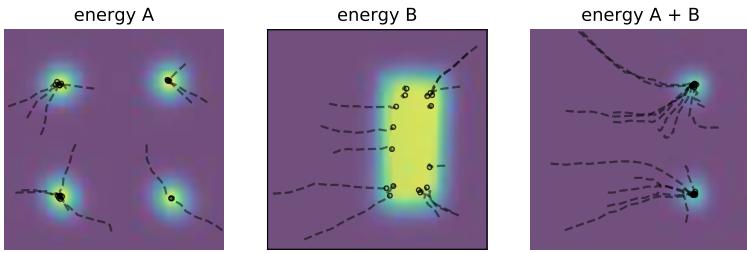


Figure 24.1: Combining two energy functions in 2d by summation, which is equivalent to multiplying the corresponding probability densities. We also illustrate some sampled trajectories towards high probability (low energy) regions. From Figure 14 of [DM19a]. Used with kind permission of Yilun Du.

24.1.1 Example: products of experts (PoE)

As an example of why energy based models are useful, suppose we want to create a generative model of proteins that are thermally stable at room temperature, and which bind to the COVID-19 spike receptor. Suppose $p_1(\mathbf{x})$ can generate stable proteins and $p_2(\mathbf{x})$ can generate proteins that bind. (For example, both of these models could be autoregressive sequence models, trained on different datasets.) We can view each of these models as “experts” about a particular aspect of the data. On their own, they are not an adequate model of the data that we have (or want to have), but we can then combine them, to represent the **conjunction of features**, by computing a **product of experts** (PoE) [Hin02]:

$$p_{12}(\mathbf{x}) = \frac{1}{Z_{12}} p_1(\mathbf{x}) p_2(\mathbf{x}) \quad (24.3)$$

This will assign high probability to proteins that are stable and which bind, and low probability to all others. By contrast, a **mixture of experts** would either generate from p_1 or from p_2 , but would not combine features from both.

If the experts are represented as energy based models (EBM), then the PoE model is also an EBM, with an energy given by

$$\mathcal{E}_{12}(\mathbf{x}) = \mathcal{E}_1(\mathbf{x}) + \mathcal{E}_2(\mathbf{x}) \quad (24.4)$$

Intuitively, we can think of each component of energy as a “soft constraint” on the data. This idea is illustrated in Figure 24.1.

24.1.2 Computational difficulties

Although the flexibility of EBMs can provide significant modeling advantages, computation of the likelihood and drawing samples from the model are generally intractable. In this chapter, we will discuss a variety of approximate methods to solve these problems.

24.2 Maximum likelihood training

The de facto standard for learning probabilistic models from iid data is maximum likelihood estimation (MLE). Let $p_{\theta}(\mathbf{x})$ be a probabilistic model parameterized by θ , and $p_{\mathcal{D}}(\mathbf{x})$ be the underlying data distribution of a dataset. We can fit $p_{\theta}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by maximizing the expected log-likelihood function over the data distribution, defined by

$$\ell(\theta) = \mathbb{E}_{\mathbf{x} \sim p_{\mathcal{D}}(\mathbf{x})} [\log p_{\theta}(\mathbf{x})] \quad (24.5)$$

as a function of θ . Here the expectation can be easily estimated with samples from the dataset. Maximizing likelihood is equivalent to minimizing the KL divergence between $p_{\mathcal{D}}(\mathbf{x})$ and $p_{\theta}(\mathbf{x})$, because

$$\ell(\theta) = -D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\theta}(\mathbf{x})) + \text{const} \quad (24.6)$$

where the constant is equal to $\mathbb{E}_{\mathbf{x} \sim p_{\mathcal{D}}(\mathbf{x})} [\log p_{\mathcal{D}}(\mathbf{x})]$ which does not depend on θ .

We cannot usually compute the likelihood of an EBM because the normalizing constant Z_{θ} is often intractable. Nevertheless, we can still estimate the gradient of the log-likelihood with MCMC approaches, allowing for likelihood maximization with stochastic gradient ascent [You99]. In particular, the gradient of the log-probability of an EBM decomposes as a sum of two terms:

$$\nabla_{\theta} \log p_{\theta}(\mathbf{x}) = -\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x}) - \nabla_{\theta} \log Z_{\theta}. \quad (24.7)$$

The first gradient term, $-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})$, is straightforward to evaluate with automatic differentiation. The challenge is in approximating the second gradient term, $\nabla_{\theta} \log Z_{\theta}$, which is intractable to compute exactly. This gradient term can be rewritten as the following expectation:

$$\nabla_{\theta} \log Z_{\theta} = \nabla_{\theta} \log \int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.8)$$

$$\stackrel{(i)}{=} \left(\int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \right)^{-1} \nabla_{\theta} \int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.9)$$

$$= \left(\int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \right)^{-1} \int \nabla_{\theta} \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.10)$$

$$\stackrel{(ii)}{=} \left(\int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \right)^{-1} \int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) (-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.11)$$

$$= \int \left(\int \exp(-\mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \right)^{-1} \exp(-\mathcal{E}_{\theta}(\mathbf{x})) (-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.12)$$

$$\stackrel{(iii)}{=} \int \frac{1}{Z_{\theta}} \exp(-\mathcal{E}_{\theta}(\mathbf{x})) (-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.13)$$

$$\stackrel{(iv)}{=} \int p_{\theta}(\mathbf{x}) (-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})) d\mathbf{x} \quad (24.14)$$

$$= \mathbb{E}_{\mathbf{x} \sim p_{\theta}(\mathbf{x})} [-\nabla_{\theta} \mathcal{E}_{\theta}(\mathbf{x})], \quad (24.15)$$

where steps (i) and (ii) are due to the chain rule of gradients, and (iii) and (iv) are from definitions in Equations (24.1) and (24.2). Thus, we can obtain an unbiased Monte Carlo estimate of the log-likelihood gradient by using

$$\nabla_{\theta} \log Z_{\theta} \simeq -\frac{1}{S} \sum_{s=1}^S \nabla_{\theta} \mathcal{E}_{\theta}(\tilde{\mathbf{x}}_s), \quad (24.16)$$

where $\tilde{\mathbf{x}}_s \sim p_{\theta}(\mathbf{x})$, i.e., a random sample from the distribution over \mathbf{x} given by the EBM. Therefore, as long as we can draw random samples from the model, we have access to an unbiased Monte Carlo estimate of the log-likelihood gradient, allowing us to optimize the parameters with stochastic gradient ascent.

Much of the literature has focused on methods for efficient MCMC sampling from EBMs. We discuss some of these methods below.

24.2.1 Gradient-based MCMC methods

Some efficient MCMC methods, such as **Langevin MCMC** (Section 12.5.6) or Hamiltonian Monte Carlo (Section 12.5), make use of the fact that the gradient of the log-probability wrt \mathbf{x} (known as the **Hyvärinen score function**, named after [Hyv05a] to distinguish it from the standard score function in Equation (3.39)) is equal to the (negative) gradient of the energy, and is therefore easy to calculate:

$$\nabla_{\mathbf{x}} \log p_{\theta}(\mathbf{x}) = -\nabla_{\mathbf{x}} \mathcal{E}_{\theta}(\mathbf{x}) - \underbrace{\nabla_{\mathbf{x}} \log Z_{\theta}}_{=0} = -\nabla_{\mathbf{x}} \mathcal{E}_{\theta}(\mathbf{x}). \quad (24.17)$$

For example, when using Langevin MCMC to sample from $p_{\theta}(\mathbf{x})$, we first draw an initial sample \mathbf{x}^0 from a simple prior distribution, and then simulate an overdamped Langevin diffusion process for K steps with step size $\epsilon > 0$:

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \frac{\epsilon^2}{2} \underbrace{\nabla_{\mathbf{x}} \log p_{\theta}(\mathbf{x}^k)}_{=-\nabla_{\mathbf{x}} \mathcal{E}_{\theta}(\mathbf{x})} + \epsilon \mathbf{z}^k, \quad k = 0, 1, \dots, K-1. \quad (24.18)$$

where $\mathbf{z}^k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is a Gaussian noise term. We show an example of this process in Figure 25.5d.

When $\epsilon \rightarrow 0$ and $K \rightarrow \infty$, \mathbf{x}^K is guaranteed to distribute as $p_{\theta}(\mathbf{x})$ under some regularity conditions. In practice we have to use a small finite ϵ , but the discretization error is typically negligible, or can be corrected with a Metropolis-Hastings step (Section 12.2), leading to the Metropolis-adjusted Langevin algorithm (Section 12.5.6).

24.2.2 Contrastive divergence

Running MCMC till convergence to obtain a sample $\mathbf{x} \sim p_{\theta}(\mathbf{x})$ can be computationally expensive. Therefore we typically need approximations to make MCMC-based learning of EBMs practical. One popular method for doing so is **contrastive divergence** (CD) [Hin02]. In CD, one initializes the MCMC chain from the datapoint \mathbf{x} , and proceeds to perform MCMC for a fixed number of steps. One can show that T steps of CD minimizes the following objective:

$$\text{CD}_T = D_{\text{KL}}(p_0 \parallel p_{\infty}) - D_{\text{KL}}(p_T \parallel p_{\infty}) \quad (24.19)$$

where p_T is the distribution over \mathbf{x} after T MCMC updates, and p_0 is the data distribution. Typically we can get good results with a small value of T , sometimes just $T = 1$. We give the details below.

24.2.2.1 Fitting RBMs with CD

CD was initially developed to fit a special kind of latent variable EBM known as a restricted Boltzmann machine (Section 4.3.3.2). This model was specifically designed to support fast block Gibbs sampling, which is required by CD (and can also be exploited by standard MCMC-based learning methods [AHS85].)

For simplicity, we will assume the hidden and visible nodes are binary, and we use 1-step contrastive divergence. As discussed in Supplementary Section 4.3.1, the binary RBM has the following energy function:

$$\mathcal{E}(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta}) = \sum_{d=1}^D \sum_{k=1}^K x_d z_k W_{dk} + \sum_{d=1}^D x_d b_d + \sum_{k=1}^K z_k c_k \quad (24.20)$$

(Henceforth we will drop the unary (bias) terms, which can be emulated by clamping $z_k = 1$ or $x_d = 1$.) This is a loglinear model where we have one binary feature per edge. Thus from Equation (4.135) the gradient of the log-likelihood is given by the clamped expectations minus the unclamped expectations:

$$\frac{\partial \ell}{\partial w_{dk}} = \frac{1}{N} \sum_{n=1}^N \mathbb{E}[x_d z_k | \mathbf{x}_n, \boldsymbol{\theta}] - \mathbb{E}[x_d z_k | \boldsymbol{\theta}] \quad (24.21)$$

We can rewrite the above gradient in matrix-vector form as follows:

$$\nabla_{\mathbf{w}} \ell = \mathbb{E}_{p_D(\mathbf{x}) p(\mathbf{z} | \mathbf{x}, \boldsymbol{\theta})} [\mathbf{x} \mathbf{z}^T] - \mathbb{E}_{p(\mathbf{z}, \mathbf{x} | \boldsymbol{\theta})} [\mathbf{x} \mathbf{z}^T] \quad (24.22)$$

(We can derive a similar expression for the gradient of the bias terms by setting $x_d = 1$ or $z_k = 1$.)

The first term in the expression for the gradient in Equation (24.21), when \mathbf{x} is fixed to a data case, is sometimes called the **clamped phase**, and the second term, when \mathbf{x} is free, is sometimes called the **unclamped phase**. When the model expectations match the empirical expectations, the two terms cancel out, the gradient becomes zero and learning stops.

We can also make a connection to the principle of **Hebbian learning** in neuroscience. In particular, Hebb's rule says that the strength of connection between two neurons that are simultaneously active should be increased. (This theory is often summarized as "Cells that fire together wire together".¹) The first term in Equation (24.21) is therefore considered a Hebbian term, and the second term an anti-Hebbian term, due to the sign change.

We can leverage the Markov structure of the bipartite graph to approximate the expectations as follows:

$$\mathbf{z}_n \sim p(\mathbf{z} | \mathbf{x}_n, \boldsymbol{\theta}) \quad (24.23)$$

$$\mathbf{x}'_n \sim p(\mathbf{x} | \mathbf{z}_n, \boldsymbol{\theta}) \quad (24.24)$$

$$\mathbf{z}'_n \sim p(\mathbf{z} | \mathbf{x}'_n, \boldsymbol{\theta}) \quad (24.25)$$

1. See https://en.wikipedia.org/wiki/Hebbian_theory.

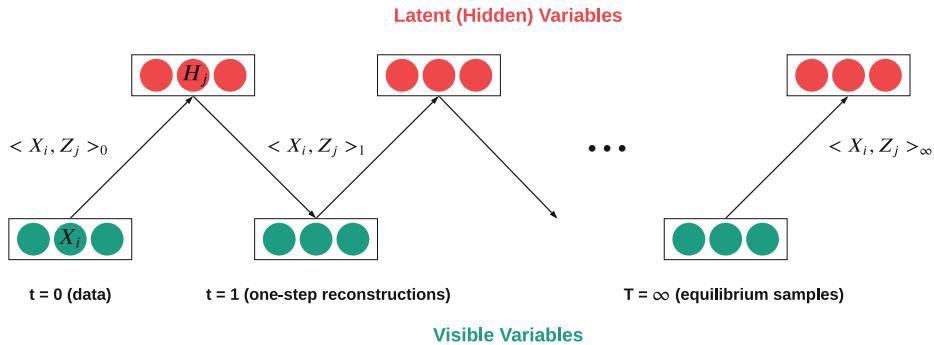


Figure 24.2: Illustration of contrastive divergence sampling for an RBM. The visible nodes are initialized at an example drawn from the dataset. Then we sample a hidden vector, then another visible vector, etc. Eventually (at “infinity”) we will be producing samples from the joint distribution $p(\mathbf{x}, \mathbf{z}|\theta)$.

We can think of \mathbf{x}'_n as the model’s best attempt at reconstructing \mathbf{x}_n after being encoded and then decoded by the model. Such samples are sometimes called **fantasy data**. See Figure 24.2 for an illustration. Given these samples, we then make the approximation

$$\mathbb{E}_{p(\cdot|\theta)} [\mathbf{x}\mathbf{z}^\top] \approx \mathbf{x}_n(\mathbf{z}'_n)^\top \quad (24.26)$$

In practice, it is common to use $\mathbb{E}[\mathbf{z}|\mathbf{x}'_n]$ instead of a sampled value \mathbf{z}'_n in the above expression, since this reduces the variance. However, it is not valid to use $\mathbb{E}[\mathbf{z}|\mathbf{x}_n]$ instead of sampling $\mathbf{z}_n \sim p(\mathbf{z}|\mathbf{x}_n)$ in Equation (24.23), because then each hidden unit would be able to pass more than 1 bit of information, so it would not act as much of a bottleneck.

The whole procedure is summarized in Algorithm 24.1. For more details, see [Hin10; Swe+10].

Algorithm 24.1: CD-1 training for an RBM with binary hidden and visible units

```

1 Initialize weights  $\mathbf{W} \in \mathbb{R}^{D \times K}$  randomly
2 for  $t = 1, 2, \dots$  do
3   for each minibatch of size  $B$  do
4     Set minibatch gradient to zero,  $\mathbf{g} := \mathbf{0}$ 
5     for each case  $\mathbf{x}_n$  in the minibatch do
6       Compute  $\boldsymbol{\mu}_n = \mathbb{E}[\mathbf{z}|\mathbf{x}_n, \mathbf{W}]$ 
7       Sample  $\mathbf{z}_n \sim p(\mathbf{z}|\mathbf{x}_n, \mathbf{W})$ 
8       Sample  $\mathbf{x}'_n \sim p(\mathbf{x}|\mathbf{z}_n, \mathbf{W})$ 
9       Compute  $\boldsymbol{\mu}'_n = \mathbb{E}[\mathbf{z}|\mathbf{x}'_n, \mathbf{W}]$ 
10      Compute gradient  $\nabla_{\mathbf{W}} = (\mathbf{x}_n)(\boldsymbol{\mu}_n)^\top - (\mathbf{x}'_n)(\boldsymbol{\mu}'_n)^\top$ 
11      Accumulate  $\mathbf{g} := \mathbf{g} + \nabla_{\mathbf{W}}$ 
12   Update parameters  $\mathbf{W} := \mathbf{W} + \eta_t \frac{1}{B} \mathbf{g}$ 

```

24.2.2.2 Persistent CD

One variant of CD that sometimes performs better is **persistent contrastive divergence** (PCD) [Tie08; TH09; You99]. In this approach, a single MCMC chain with a persistent state is employed to sample from the EBM. In PCD, we do not restart the MCMC chain when training on a new datapoint; rather, we carry over the state of the previous MCMC chain and use it to initialize a new MCMC chain for the next training step. See Algorithm 12 for some pseudocode. Hence there are two dynamical processes running at different time scales: the states \mathbf{x} change quickly, and the parameters $\boldsymbol{\theta}$ change slowly.

Algorithm 24.2: Persistent MCMC-SGD for fitting an EBM

```

1 Initialize parameters  $\boldsymbol{\theta}$  randomly
2 Initialize chains  $\tilde{\mathbf{x}}_{1:S}$  randomly
3 Initialize learning rate  $\eta$ 
4 for  $t = 1, 2, \dots$  do
5   for  $\mathbf{x}_b$  in minibatch of size  $B$  do
6      $\mathbf{g}_b = \nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}_b)$ 
7     for sample  $s = 1 : S$  do
8       Sample  $\tilde{\mathbf{x}}_s \sim \text{MCMC}(\text{target} = p(\cdot | \boldsymbol{\theta}), \text{init} = \tilde{\mathbf{x}}_s, \text{nsteps} = N)$ 
9        $\tilde{\mathbf{g}}_s = \nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}_s)$ 
10       $\mathbf{g}_t = -(\frac{1}{B} \sum_{b=1}^B \mathbf{g}_b) - (\frac{1}{S} \sum_{s=1}^S \tilde{\mathbf{g}}_s)$ 
11       $\boldsymbol{\theta} := \boldsymbol{\theta} + \eta \mathbf{g}_t$ 
12    Decrease step size  $\eta$ 

```

A theoretical justification for this was given in [You89], who showed that we can start the MCMC chain at its previous value, and just take a few steps, because $p(\mathbf{x} | \boldsymbol{\theta}_t)$ is likely to be close to $p(\mathbf{x} | \boldsymbol{\theta}_{t-1})$, since we only changed the parameters by a small amount in the intervening SGD step.

24.2.2.3 Other methods

PCD can be further improved by keeping multiple historical states of the MCMC chain in a replay buffer and initialize new MCMC chains by randomly sampling from it [DM19b]. Other variants of CD include mean field CD [WH02], and multi-grid CD [Gao+18].

EBMs trained with CD may not capture the data distribution faithfully, since truncated MCMC can lead to biased gradient updates that hurt the learning dynamics [SMB10; FI10; Nij+19]. There are several methods that focus on removing this bias for improved MCMC training. For example, one line of work proposes unbiased estimators of the gradient through coupled MCMC [JOA17; QZW19]; and Du et al. [Du+20] propose to reduce the bias by differentiating through the MCMC sampling algorithm and estimating an entropy correction term.

24.3 Score matching (SM)

If two continuously differentiable real-valued functions $f(\mathbf{x})$ and $g(\mathbf{x})$ have equal first derivatives everywhere, then $f(\mathbf{x}) \equiv g(\mathbf{x}) + \text{constant}$. When $f(\mathbf{x})$ and $g(\mathbf{x})$ are log probability density functions (pdf's) with equal first derivatives, the normalization requirement (Equation (24.1)) implies that $\int \exp(f(\mathbf{x})) d\mathbf{x} = \int \exp(g(\mathbf{x})) d\mathbf{x} = 1$, and therefore $f(\mathbf{x}) \equiv g(\mathbf{x})$. As a result, one can learn an EBM by (approximately) matching the first derivatives of its log-pdf to the first derivatives of the log pdf of the data distribution. If they match, then the EBM captures the data distribution exactly. The first-order gradient function of a log pdf wrt its input, $\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$, is called the (Stein) **score** function. (This is distinct from the Fisher score, $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$.) For training EBMs, it is useful to transform the equivalence of distributions to the equivalence of scores, because the score of an EBM can be easily obtained as follows:

$$\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x}) \triangleq \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) = -\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) \quad (24.27)$$

We see that this does not involve the typically intractable normalizing constant $Z_{\boldsymbol{\theta}}$.

Let $p_{\mathcal{D}}(\mathbf{x})$ be the underlying data distribution, from which we have a finite number of iid samples but do not know its pdf. The **score matching** objective [Hyv05b] minimizes a discrepancy between two distributions called the **Fisher divergence**:

$$D_F(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\frac{1}{2} \|\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x}) - \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})\|^2 \right]. \quad (24.28)$$

The expectation wrt $p_{\mathcal{D}}(\mathbf{x})$, in this objective and its variants below, admits a trivial unbiased Monte Carlo estimator using the empirical mean of samples $\mathbf{x} \sim p_{\mathcal{D}}(\mathbf{x})$. However, the second term of Equation (24.28), $\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x})$, is generally impractical to calculate since it requires knowing the pdf of $p_{\mathcal{D}}(\mathbf{x})$. We discuss a solution to this below.

24.3.1 Basic score matching

Hyvärinen [Hyv05b] shows that, under certain regularity conditions, the Fisher divergence can be rewritten using integration by parts, with second derivatives of $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$ replacing the unknown first derivatives of $p_{\mathcal{D}}(\mathbf{x})$:

$$D_F(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} \right)^2 - \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i^2} \right] + \text{constant} \quad (24.29)$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\frac{1}{2} \|\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x})\|^2 + \text{tr}(\mathbf{J}_{\mathbf{x}} \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x})) \right] + \text{constant} \quad (24.30)$$

where d is the dimensionality of \mathbf{x} , and $\mathbf{J}_{\mathbf{x}} \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x})$ is the Jacobian of the score function. The constant does not affect optimization and thus can be dropped for training. It is shown by [Hyv05b] that estimators based on score matching are consistent under some regularity conditions, meaning that the parameter estimator obtained by minimizing Equation (24.28) converges to the true parameters in the limit of infinite data. See Figure 25.5 for an example.

An important downside of the objective Equation (24.30) is that it takes $O(d^2)$ time to compute the trace of the Jacobian. For this reason, the implicit SM formulation of Equation (24.30) has only

24.3. Score matching (SM)

been applied to relatively simple energy functions where computation of the second derivatives is tractable.

Score Matching assumes a continuous data distribution with positive density over the space, but it can be generalized to discrete or bounded data distributions [Hyv07b; Lyu12]. It is also possible to consider higher-order gradients of log pdf's beyond first derivatives [PDL+12].

24.3.2 Denoising score matching (DSM)

The Score Matching objective in Equation (24.30) requires several regularity conditions for $\log p_D(\mathbf{x})$, e.g., it should be continuously differentiable and finite everywhere. However, these conditions may not always hold in practice. For example, a distribution of digital images is typically discrete and bounded, because the values of pixels are restricted to the range $\{0, 1, \dots, 255\}$. Therefore, $\log p_D(\mathbf{x})$ in this case is discontinuous and is negative infinity outside the range, and thus SM is not directly applicable.

To alleviate this, one can add a bit of noise to each datapoint: $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon$. As long as the noise distribution $p(\epsilon)$ is smooth, the resulting noisy data distribution $q(\tilde{\mathbf{x}}) = \int q(\tilde{\mathbf{x}} | \mathbf{x}) p_D(\mathbf{x}) d\mathbf{x}$ is also smooth, and thus the Fisher divergence $D_F(q(\tilde{\mathbf{x}}) \| p_{\theta}(\tilde{\mathbf{x}}))$ is a proper objective. [KL10] showed that the objective with noisy data can be approximated by the noiseless Score Matching objective of Equation (24.30) plus a regularization term; this regularization makes Score Matching applicable to a wider range of data distributions, but still requires expensive second-order derivatives.

[Vin11] proposed an elegant and scalable solution to the above difficulty, by showing that:

$$D_F(q(\tilde{\mathbf{x}}) \| p_{\theta}(\tilde{\mathbf{x}})) = \mathbb{E}_{q(\tilde{\mathbf{x}})} \left[\frac{1}{2} \|\nabla_{\tilde{\mathbf{x}}} \log p_{\theta}(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \log q(\tilde{\mathbf{x}})\|_2^2 \right] \quad (24.31)$$

$$= \mathbb{E}_{q(\mathbf{x}, \tilde{\mathbf{x}})} \left[\frac{1}{2} \|\nabla_{\tilde{\mathbf{x}}} \log p_{\theta}(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \log q(\tilde{\mathbf{x}} | \mathbf{x})\|_2^2 \right] + \text{constant} \quad (24.32)$$

$$= \frac{1}{2} \mathbb{E}_{q(\mathbf{x}, \tilde{\mathbf{x}})} \left[\left\| s_{\theta}(\tilde{\mathbf{x}}) - \frac{(\mathbf{x} - \tilde{\mathbf{x}})}{\sigma^2} \right\|_2^2 \right] \quad (24.33)$$

where $s_{\theta}(\tilde{\mathbf{x}}) = \nabla_{\tilde{\mathbf{x}}} \log p_{\theta}(\tilde{\mathbf{x}})$ is the estimated score function, and

$$\nabla_{\mathbf{x}} \log q(\tilde{\mathbf{x}} | \mathbf{x}) = \nabla_{\mathbf{x}} \log \mathcal{N}(\tilde{\mathbf{x}} | \mathbf{x}, \sigma^2 \mathbf{I}) = \frac{-(\tilde{\mathbf{x}} - \mathbf{x})}{\sigma^2} + \text{const} \quad (24.34)$$

The directional term $\mathbf{x} - \tilde{\mathbf{x}}$ corresponds to moving from the noisy input towards the clean input, and we want the score function to approximate this denoising operation. (We will see this idea again in Section 25.3, where we discuss diffusion models.)

To compute the expectation in Equation (24.33), we can sample from $p_D(\mathbf{x})$ and then sample the noise term $\tilde{\mathbf{x}}$. (The constant term does not affect optimization and can be ignored without changing the optimal solution.)

This estimation method is called **denoising score matching** (DSM) by [Vin11]. Similar formulations were also explored by Raphan and Simoncelli [RS07; RS11] and can be traced back to Tweedie's formula (Supplementary Section 3.3) and Stein's unbiased risk estimation [Ste81].

24.3.2.1 Difficulties

The major drawback of adding noise to data arises when $p_{\mathcal{D}}(\mathbf{x})$ is already a well-behaved distribution that satisfies the regularity conditions required by score matching. In this case, $D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})) \neq D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}))$, and DSM is not a consistent objective because the optimal EBM matches the noisy distribution $q(\tilde{\mathbf{x}})$, not $p_{\mathcal{D}}(\mathbf{x})$. This inconsistency becomes non-negligible when $q(\tilde{\mathbf{x}})$ significantly differs from $p_{\mathcal{D}}(\mathbf{x})$.

One way to attenuate the inconsistency of DSM is to choose $q \approx p_{\mathcal{D}}$, i.e., use a small noise perturbation. However, this often significantly increases the variance of objective values and hinders optimization. As an example, suppose $q(\tilde{\mathbf{x}} | \mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}} | \mathbf{x}, \sigma^2 I)$ and $\sigma \approx 0$. The corresponding DSM objective is

$$\begin{aligned} D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})) &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(0, I)} \left[\frac{1}{2} \left\| \frac{\mathbf{z}}{\sigma} + \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x} + \sigma \mathbf{z}) \right\|_2^2 \right] \\ &\simeq \frac{1}{2N} \sum_{i=1}^N \left\| \frac{\mathbf{z}^{(i)}}{\sigma} + \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} + \sigma \mathbf{z}^{(i)}) \right\|_2^2, \end{aligned} \quad (24.35)$$

where $\{\mathbf{x}^{(i)}\}_{i=1}^N \stackrel{\text{i.i.d.}}{\sim} p_{\mathcal{D}}(\mathbf{x})$, and $\{\mathbf{z}^{(i)}\}_{i=1}^N \stackrel{\text{i.i.d.}}{\sim} \mathcal{N}(\mathbf{0}, I)$. When $\sigma \rightarrow 0$, we can leverage Taylor series expansion to rewrite the Monte Carlo estimator in Equation (24.35) to

$$\frac{1}{2N} \sum_{i=1}^N \left[\frac{2}{\sigma} (\mathbf{z}^{(i)})^\top \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) + \frac{\|\mathbf{z}^{(i)}\|_2^2}{\sigma^2} \right] + \text{constant}. \quad (24.36)$$

When estimating the above expectation with samples, the variances of $(\mathbf{z}^{(i)})^\top \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})/\sigma$ and $\|\mathbf{z}^{(i)}\|_2^2/\sigma^2$ will both grow unbounded as $\sigma \rightarrow 0$ due to division by σ and σ^2 . This enlarges the variance of DSM and makes optimization challenging. Various methods have been proposed to reduce this variance (see e.g., [Wan+20d]).

24.3.3 Sliced score matching (SSM)

By adding noise to data, DSM avoids the expensive computation of second-order derivatives. However, as mentioned before, the optimal EBM that minimizes the DSM objective corresponds to the distribution of noise-perturbed data $q(\tilde{\mathbf{x}})$, not the original noise-free data distribution $p_{\mathcal{D}}(\mathbf{x})$. In other words, DSM does not give a consistent estimator of the data distribution, i.e., one cannot directly obtain an EBM that exactly matches the data distribution even with unlimited data.

Sliced score matching (SSM) [Son+19] is one alternative to Denoising Score Matching that is both consistent and computationally efficient. Instead of minimizing the Fisher divergence between two vector-valued scores, SSM randomly samples a projection vector \mathbf{v} , takes the inner product between \mathbf{v} and the two scores, and then compares the resulting two scalars. More specifically, sliced score matching minimizes the following divergence called the **sliced Fisher divergence**:

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{p(\mathbf{v})} \left[\frac{1}{2} (\mathbf{v}^\top \nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x}) - \mathbf{v}^\top \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}))^2 \right], \quad (24.37)$$

where $p(\mathbf{v})$ denotes a projection distribution such that $\mathbb{E}_{p(\mathbf{v})}[\mathbf{v}\mathbf{v}^\top]$ is positive definite. Similar to Fisher divergence, sliced Fisher divergence has an implicit form that does not involve the unknown

24.3. Score matching (SM)

$\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x})$, which is given by

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{p(\mathbf{v})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} v_i \right)^2 + \sum_{i=1}^d \sum_{j=1}^d \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j} v_i v_j \right] + C. \quad (24.38)$$

All expectations in the above objective can be estimated with empirical means, and again the constant term C can be removed without affecting training. The second term involves second-order derivatives of $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$, but contrary to SM, it can be computed efficiently with a cost linear in the dimensionality d . This is because

$$\sum_{i=1}^d \sum_{j=1}^d \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j} v_i v_j = \sum_{i=1}^d \frac{\partial}{\partial x_i} \underbrace{\left(\sum_{j=1}^d \frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_j} v_j \right)}_{:=f(\mathbf{x})} v_i, \quad (24.39)$$

where $f(\mathbf{x})$ is the same for different values of i . Therefore, we only need to compute it once with $O(d)$ computation, *plus* another $O(d)$ computation for the outer sum to evaluate Equation (24.39), whereas the original SM objective requires $O(d^2)$ computation.

For many choices of $p(\mathbf{v})$, part of the SSM objective (Equation (24.38)) can be evaluated in closed form, potentially leading to lower variance. For example, when $p(\mathbf{v}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, we have

$$\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{p(\mathbf{v})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} v_i \right)^2 \right] = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} \right)^2 \right] \quad (24.40)$$

and as a result,

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} \right)^2 + \sum_{i=1}^d \sum_{j=1}^d \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j} v_i v_j \right] + C \quad (24.41)$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \left[\frac{1}{2} (\mathbf{v}^\top \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x}))^2 + \mathbf{v}^\top [\mathbf{J}\mathbf{v}] \right] \quad (24.42)$$

where $\mathbf{J} = \mathbf{J}_x \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x})$. (Note that $\mathbf{J}\mathbf{v}$ can be computed using a Jacobian vector product operation.)

The above objective Equation (24.41) can also be obtained by approximating the sum of second-order gradients in the standard SM objective (Equation (24.30)) with the Hutchinson trace estimator [Ski89; Hut89; Mey+21]. It often (but not always) has lower variance than Equation (24.38), and can perform better in some applications [Son+19].

24.3.4 Connection to contrastive divergence

Though score matching and contrastive divergence (Section 24.2.2) are seemingly very different approaches, they are closely connected to each other. In fact, score matching can be viewed as a special instance of contrastive divergence in the limit of a particular MCMC sampler [Hyv07a]. Moreover, the Fisher divergence optimized by Score Matching is related to the derivative of KL divergence [Cov99], which is the underlying objective of Contrastive Divergence.

Contrastive divergence requires sampling from the EBM $\mathcal{E}_\theta(\mathbf{x})$, and one popular method for doing so is Langevin MCMC. Recall from Section 24.2.1 that given any initial datapoint \mathbf{x}^0 , the Langevin MCMC method executes the following

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k - \frac{\epsilon}{2} \nabla_{\mathbf{x}} \mathcal{E}_\theta(\mathbf{x}^k) + \sqrt{\epsilon} \mathbf{z}^k, \quad (24.43)$$

iteratively for $k = 0, 1, \dots, K - 1$, where $\mathbf{z}^k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\epsilon > 0$ is the step size.

Suppose we only run one-step Langevin MCMC for contrastive divergence. In this case, the gradient of the log-likelihood is given by

$$\begin{aligned} \mathbb{E}_{p_D(\mathbf{x})} [\nabla_\theta \log p_\theta(\mathbf{x})] &= -\mathbb{E}_{p_D(\mathbf{x})} [\nabla_\theta \mathcal{E}_\theta(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim p_\theta(\mathbf{x})} [\nabla_\theta \mathcal{E}_\theta(\mathbf{x})] \\ &\simeq -\mathbb{E}_{p_D(\mathbf{x})} [\nabla_\theta \mathcal{E}_\theta(\mathbf{x})] + \mathbb{E}_{p_\theta(\mathbf{x}), \mathbf{z} \sim \mathcal{N}(0, \mathbf{I})} \left[\nabla_\theta \mathcal{E}_\theta \left(\mathbf{x} - \frac{\epsilon^2}{2} \nabla_{\mathbf{x}} E_{\theta'}(\mathbf{x}) + \epsilon \mathbf{z} \right) \Big|_{\theta'=\theta} \right]. \end{aligned} \quad (24.44)$$

After Taylor series expansion with respect to ϵ followed by some algebraic manipulations, the above equation can be transformed to the following [Hyv07a]:

$$\frac{\epsilon^2}{2} \nabla_\theta D_F(p_D(\mathbf{x}) \parallel p_\theta(\mathbf{x})) + o(\epsilon^2). \quad (24.45)$$

When ϵ is sufficiently small, it corresponds to the re-scaled gradient of the score matching objective.

In general, score matching minimizes the Fisher divergence $D_F(p_D(\mathbf{x}) \parallel p_\theta(\mathbf{x}))$, whereas Contrastive Divergence minimizes an objective related to the KL divergence $D_{KL}(p_D(\mathbf{x}) \parallel p_\theta(\mathbf{x}))$, as shown in Equation (24.19). The above connection of score matching and Contrastive Divergence is a natural consequence of the connection between those two statistical divergences, as characterized by *de Bruijin's identity* [Cov99; Lyu12]:

$$\frac{d}{dt} D_{KL}(q_t(\tilde{\mathbf{x}}) \parallel p_{\theta,t}(\tilde{\mathbf{x}})) = -\frac{1}{2} D_F(q_t(\tilde{\mathbf{x}}) \parallel p_{\theta,t}(\tilde{\mathbf{x}})).$$

Here $q_t(\tilde{\mathbf{x}})$ and $p_{\theta,t}(\tilde{\mathbf{x}})$ denote smoothed versions of $p_D(\mathbf{x})$ and $p_\theta(\mathbf{x})$, resulting from adding Gaussian noise to \mathbf{x} with variance t ; i.e., $\tilde{\mathbf{x}} \sim \mathcal{N}(\mathbf{x}, t\mathbf{I})$.

24.3.5 Score-based generative models

We have seen how to use score matching to fit EBMs by learning the scalar energy function $\mathcal{E}_\theta(\mathbf{x})$. We can alternatively directly learn the score function, $s_\theta(\mathbf{x}) = \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x})$; this is called a score-based generative model, and is discussed in Section 25.3. Such unconstrained score models are not guaranteed to output a conservative vector field, meaning they do not correspond to the gradient of any function. However, both methods seem to give comparable results [SH21].

24.4 Noise contrastive estimation

Another principle for learning the parameters of EBMs is **Noise contrastive estimation** (NCE), introduced by [GH10]. It is based on the idea that we can learn an EBM by contrasting it with another distribution with known density.

Let $p_{\mathcal{D}}(\mathbf{x})$ be our data distribution, and let $p_n(\mathbf{x})$ be a chosen distribution with known density, called a noise distribution. This noise distribution is usually simple and has a tractable pdf, like $\mathcal{N}(\mathbf{0}, \mathbf{I})$, such that we can compute the pdf and generate samples from it efficiently. Strategies exist to learn the noise distribution, as referenced below. Furthermore, let y be a binary variable with Bernoulli distribution, which we use to define a mixture distribution of noise and data: $p_{n,\text{data}}(\mathbf{x}) = p(y=0)p_n(\mathbf{x}) + p(y=1)p_{\mathcal{D}}(\mathbf{x})$. According to Bayes' rule, given a sample \mathbf{x} from this mixture, the posterior probability of $y=0$ is

$$p_{n,\text{data}}(y=0 | \mathbf{x}) = \frac{p_{n,\text{data}}(\mathbf{x} | y=0)p(y=0)}{p_{n,\text{data}}(\mathbf{x})} = \frac{p_n(\mathbf{x})}{p_n(\mathbf{x}) + \nu p_{\mathcal{D}}(\mathbf{x})} \quad (24.46)$$

where $\nu = p(y=1)/p(y=0)$.

Let our EBM $p_{\theta}(\mathbf{x})$ be defined as:

$$p_{\theta}(\mathbf{x}) = \exp(-\mathcal{E}_{\theta}(\mathbf{x}))/Z_{\theta} \quad (24.47)$$

Contrary to most other EBMs, Z_{θ} is treated as a learnable (scalar) parameter in NCE. Given this model, similar to the mixture of noise and data above, we can define a mixture of noise and the model distribution: $p_{n,\theta}(\mathbf{x}) = p(y=0)p_n(\mathbf{x}) + p(y=1)p_{\theta}(\mathbf{x})$. The posterior probability of $y=0$ given this noise/model mixture is:

$$p_{n,\theta}(y=0 | \mathbf{x}) = \frac{p_n(\mathbf{x})}{p_n(\mathbf{x}) + \nu p_{\theta}(\mathbf{x})} \quad (24.48)$$

In NCE, we indirectly fit $p_{\theta}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by fitting $p_{n,\theta}(y | \mathbf{x})$ to $p_{n,\text{data}}(y | \mathbf{x})$ through a standard conditional maximum likelihood objective:

$$\boldsymbol{\theta}^* = \underset{\theta}{\operatorname{argmin}} \mathbb{E}_{p_{n,\text{data}}(\mathbf{x})} [D_{KL}(p_{n,\text{data}}(y | \mathbf{x}) \| p_{n,\theta}(y | \mathbf{x}))] \quad (24.49)$$

$$= \underset{\theta}{\operatorname{argmax}} \mathbb{E}_{p_{n,\text{data}}(\mathbf{x}, y)} [\log p_{n,\theta}(y | \mathbf{x})], \quad (24.50)$$

which can be solved using stochastic gradient ascent. Just like any other deep classifier, when the model is sufficiently powerful, $p_{n,\theta^*}(y | \mathbf{x})$ will match $p_{n,\text{data}}(y | \mathbf{x})$ at the optimum. In that case:

$$p_{n,\theta^*}(y=0 | \mathbf{x}) \equiv p_{n,\text{data}}(y=0 | \mathbf{x}) \quad (24.51)$$

$$\iff \frac{p_n(\mathbf{x})}{p_n(\mathbf{x}) + \nu p_{\theta^*}(\mathbf{x})} \equiv \frac{p_n(\mathbf{x})}{p_n(\mathbf{x}) + \nu p_{\mathcal{D}}(\mathbf{x})} \quad (24.52)$$

$$\iff p_{\theta^*}(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x}) \quad (24.53)$$

Consequently, $E_{\theta^*}(\mathbf{x})$ is an unnormalized energy function that matches the data distribution $p_{\mathcal{D}}(\mathbf{x})$, and Z_{θ^*} is the corresponding normalizing constant.

As one unique feature that contrastive divergence and score matching do not have, NCE provides the normalizing constant of an Energy-Based Model as a by-product of its training procedure. When the EBM is very expressive, e.g., a deep neural network with many parameters, we can assume it is able to approximate a normalized probability density and absorb Z_{θ} into the parameters of $\mathcal{E}_{\theta}(\mathbf{x})$ [MT12], or equivalently, fixing $Z_{\theta} = 1$. The resulting EBM trained with NCE will be self-normalized, i.e., having a normalizing constant close to 1.

In practice, choosing the right noise distribution $p_n(\mathbf{x})$ is critical to the success of NCE, especially for structured and high-dimensional data. As argued in Gutmann and Hirayama [GH12], NCE works the best when the noise distribution is close to the data distribution (but not exactly the same). Many methods have been proposed to automatically tune the noise distribution, such as Adversarial Contrastive Estimation [BLC18], Conditional NCE [CG18] and Flow Contrastive Estimation [Gao+20]. NCE can be further generalized using Bregman divergences (Section 5.1.10), where the formulation introduced here reduces to a special case.

24.4.1 Connection to score matching

Noise contrastive estimation provides a family of objectives that vary for different $p_n(\mathbf{x})$ and ν . This flexibility may allow adaptation to special properties of a task with hand-tuned $p_n(\mathbf{x})$ and ν , and may also give a unified perspective for different approaches. In particular, when using an appropriate $p_n(\mathbf{x})$ and a slightly different parameterization of $p_{n,\theta}(y | \mathbf{x})$, we can recover score matching from NCE [GH12].

Specifically, we choose the noise distribution $p_n(\mathbf{x})$ to be a perturbed data distribution: given a small (deterministic) vector \mathbf{v} , let $p_n(\mathbf{x}) = p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})$. It is efficient to sample from this $p_n(\mathbf{x})$, since we can first draw any datapoint $\mathbf{x}' \sim p_{\mathcal{D}}(\mathbf{x}')$ and then compute $\mathbf{x} = \mathbf{x}' + \mathbf{v}$. It is, however, difficult to evaluate the density of $p_n(\mathbf{x})$ because $p_{\mathcal{D}}(\mathbf{x})$ is unknown. Since the original parameterization of $p_{n,\theta}(y | \mathbf{x})$ in NCE (Equation (24.48)) depends on the pdf of $p_n(\mathbf{x})$, we cannot directly apply the standard NCE objective. Instead, we replace $p_n(\mathbf{x})$ with $p_{\theta}(\mathbf{x} - \mathbf{v})$ and parameterize $p_{n,\theta}(y = 0 | \mathbf{x})$ with the following form

$$p_{n,\theta}(y = 0 | \mathbf{x}) := \frac{p_{\theta}(\mathbf{x} - \mathbf{v})}{p_{\theta}(\mathbf{x}) + p_{\theta}(\mathbf{x} - \mathbf{v})} \quad (24.54)$$

In this case, the NCE objective (Equation (24.50)) reduces to:

$$\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\log(1 + \exp(\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x} - \mathbf{v})) + \log(1 + \exp(\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x} + \mathbf{v})))] \quad (24.55)$$

At $\boldsymbol{\theta}^*$, we have a solution where:

$$p_{n,\boldsymbol{\theta}^*}(y = 0 | \mathbf{x}) \equiv p_{n,\text{data}}(y = 0 | \mathbf{x}) \quad (24.56)$$

$$\implies \frac{p_{\boldsymbol{\theta}^*}(\mathbf{x} - \mathbf{v})}{p_{\boldsymbol{\theta}^*}(\mathbf{x}) + p_{\boldsymbol{\theta}^*}(\mathbf{x} - \mathbf{v})} \equiv \frac{p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})}{p_{\mathcal{D}}(\mathbf{x}) + p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})} \quad (24.57)$$

which implies that $p_{\boldsymbol{\theta}^*}(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x})$, i.e., our model matches the data distribution.

As noted in Gutmann and Hirayama [GH12] and Song et al. [Son+19], when $\|\mathbf{v}\|_2 \approx 0$, the NCE objective Equation (24.50) has the following equivalent form by Taylor expansion

$$\operatorname{argmin}_{\boldsymbol{\theta}} \frac{1}{4} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\frac{1}{2} \sum_{i=1}^d \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} v_i \right)^2 + \sum_{i=1}^d \sum_{j=1}^d \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j} v_i v_j \right] + 2 \log 2 + o(\|\mathbf{v}\|_2^2). \quad (24.58)$$

Comparing against Equation (24.38), we immediately see that the above objective equals that of SSM, if we ignore small additional terms hidden in $o(\|\mathbf{v}\|_2^2)$ and take the expectation with respect to \mathbf{v} over a user-specified distribution $p(\mathbf{v})$.

24.5 Other methods

Aside from MCMC-based training, score matching and noise contrastive estimation, there are also other methods for learning EBMs. Below we briefly survey some examples of them. Interested readers can learn more details from references therein.

24.5.1 Minimizing Differences/Derivatives of KL Divergences

The overarching strategy for learning probabilistic models from data is to minimize the KL divergence between data and model distributions. However, because the normalizing constants of EBMs are typically intractable, it is hard to directly evaluate the KL divergence when the model is an EBM (see the discussion in Section 24.2.1). One generic idea that has frequently circumvented this difficulty is to consider differences/derivatives of KL divergences. It turns out that the unknown partition functions of EBMs are often cancelled out after taking the difference of two closely related KL divergences, or computing the derivatives.

Typical examples of this strategy include minimum velocity learning [Mov08; Wan+20d], minimum probability flow [SDBD11], and minimum KL contraction [Lyu11], to name a few. In minimum velocity learning and minimum probability flow, a Markov chain is designed such that it starts from the data distribution $p_{\mathcal{D}}(\mathbf{x})$ and converges to the EBM distribution $p_{\theta}(\mathbf{x}) = e^{-\mathcal{E}_{\theta}(\mathbf{x})}/Z_{\theta}$. Specifically, the Markov chain satisfies $p_0(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x})$ and $p_{\infty}(\mathbf{x}) \equiv p_{\theta}(\mathbf{x})$, where we denote by $p_t(\mathbf{x})$ the state distribution at time $t \geq 0$.

This Markov chain will evolve towards $p_{\theta}(\mathbf{x})$ unless $p_{\mathcal{D}}(\mathbf{x}) \equiv p_{\theta}(\mathbf{x})$. Therefore, we can fit the EBM distribution $p_{\theta}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by minimizing the modulus of the “velocity” of this evolution, defined by

$$\frac{d}{dt} D_{\text{KL}}(p_t(\mathbf{x}) \parallel p_{\theta}(\mathbf{x})) \Big|_{t=0} \quad \text{or} \quad \frac{d}{dt} D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_t(\mathbf{x})) \Big|_{t=0} \quad (24.59)$$

in minimum velocity learning and minimum probability flow respectively. These objectives typically do not require computing the normalizing constant Z_{θ} .

In minimum KL contraction [Lyu11], a distribution transformation Φ is chosen such that

$$D_{\text{KL}}(p(\mathbf{x}) \parallel q(\mathbf{x})) \geq D_{\text{KL}}(\Phi\{p(\mathbf{x})\} \parallel \Phi\{q(\mathbf{x})\}) \quad (24.60)$$

with equality if and only if $p(\mathbf{x}) = q(\mathbf{x})$. We can leverage this Φ to train an EBM, by minimizing

$$D_{\text{KL}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\theta}(\mathbf{x})) - D_{\text{KL}}(\Phi\{p_{\mathcal{D}}(\mathbf{x})\} \parallel \Phi\{p_{\theta}(\mathbf{x})\}). \quad (24.61)$$

This objective does not require computing the partition function Z_{θ} whenever Φ is linear.

Minimum velocity learning, minimum probability flow, and minimum KL contraction can all be viewed as generalizations to score matching and noise contrastive estimation [Mov08; SDBD11; Lyu11].

24.5.2 Minimizing the Stein discrepancy

We can train EBMs by minimizing the **Stein discrepancy**, defined by

$$D_{\text{Stein}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\theta}(\mathbf{x})) := \sup_{\mathbf{f} \in \mathcal{F}} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\nabla_{\mathbf{x}} \log p_{\theta}(\mathbf{x})^T \mathbf{f}(\mathbf{x}) + \text{trace}(\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}))], \quad (24.62)$$

where \mathcal{F} is a family of vector-valued functions, and $\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x})$ denotes the Jacobian of $\mathbf{f}(\mathbf{x})$. (See [Ana+23] for a recent review of Stein’s method.) With some regularity conditions [GM15; LLJ16; CSG16], we have $D_S(p_{\mathcal{D}}(\mathbf{x}) \| p_{\theta}(\mathbf{x})) \geq 0$, where the equality holds if and only if $p_{\mathcal{D}}(\mathbf{x}) \equiv p_{\theta}(\mathbf{x})$. Similar to score matching (Equation (24.30)), the objective Equation (24.62) only involves the score function of $p_{\theta}(\mathbf{x})$, and does not require computing the EBM’s partition function. Still, the trace term in Equation (24.62) may demand expensive computation, and does not scale well to high dimensional data.

There are two common methods that sidestep this difficulty. [CSG16] and [LLJ16] discovered that when \mathcal{F} is a unit ball in a reproducing kernel Hilbert space (RKHS) with a fixed kernel, the Stein discrepancy becomes **kernelized Stein discrepancy**, where the trace term is a constant and does not affect optimization. Otherwise, $\text{trace}(\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}))$ can be approximated with the Skilling-Hutchinson trace estimator [Ski89; Hut89; Gra+20c].

24.5.3 Adversarial training

Recall from Section 24.2.1 that when training EBMs with maximum likelihood estimation (MLE), we need to sample from the EBM per training iteration. However, sampling using multiple MCMC steps is expensive and requires careful tuning of the Markov chain. One way to avoid this difficulty is to use non-MLE methods that do not need sampling, such as score matching and noise contrastive estimation. Here we introduce another family of methods that sidestep costly MCMC sampling by learning an auxiliary model through adversarial training, which allows fast sampling.

Using the definition of EBMs, we can rewrite the maximum likelihood objective by introducing a variational distribution $q_{\phi}(\mathbf{x})$ parameterized by ϕ :

$$\begin{aligned} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\log p_{\theta}(\mathbf{x})] &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - \log Z_{\theta} \\ &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - \log \int e^{-\mathcal{E}_{\theta}(\mathbf{x})} d\mathbf{x} \\ &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - \log \int q_{\phi}(\mathbf{x}) \frac{e^{-\mathcal{E}_{\theta}(\mathbf{x})}}{q_{\phi}(\mathbf{x})} d\mathbf{x} \\ &\stackrel{(i)}{\leq} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - \int q_{\phi}(\mathbf{x}) \log \frac{e^{-\mathcal{E}_{\theta}(\mathbf{x})}}{q_{\phi}(\mathbf{x})} d\mathbf{x} \\ &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - \mathbb{E}_{q_{\phi}(\mathbf{x})}[-\mathcal{E}_{\theta}(\mathbf{x})] - H(q_{\phi}(\mathbf{x})), \end{aligned} \quad (24.63)$$

where $H(q_{\phi}(\mathbf{x}))$ denotes the entropy of $q_{\phi}(\mathbf{x})$. Step (i) is due to Jensen’s inequality. Equation (24.63) provides an upper bound to the expected log-likelihood. For EBM training, we can first minimize the upper bound Equation (24.63) with respect to $q_{\phi}(\mathbf{x})$ so that it is closer to the likelihood objective, and then maximize Equation (24.63) with respect to $\mathcal{E}_{\theta}(\mathbf{x})$ as a surrogate for maximizing likelihood. This amounts to using the following maximin objective

$$\max_{\theta} \min_{\phi} \mathbb{E}_{q_{\phi}(\mathbf{x})}[\mathcal{E}_{\theta}(\mathbf{x})] - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\mathcal{E}_{\theta}(\mathbf{x})] - H(q_{\phi}(\mathbf{x})). \quad (24.64)$$

Optimizing the above objective is similar to training GANs (Chapter 26), and can be achieved by adversarial training. The variational distribution $q_{\phi}(\mathbf{x})$ should allow both fast sampling and efficient entropy evaluation to make Equation (24.64) tractable. This limits the model family of $q_{\phi}(\mathbf{x})$, and

usually restricts our choice to invertible probabilistic models, such as inverse autoregressive flow (Section 23.2.4.3). See Dai et al. [Dai+19b] for an example on designing $q_\phi(\mathbf{x})$ and training EBMs with Equation (24.64).

Kim and Bengio [KB16] and Zhai et al. [Zha+16] propose to represent $q_\phi(\mathbf{x})$ with neural samplers, like the generator of GANs. A neural sampler is a deterministic mapping g_ϕ that maps a random Gaussian noise $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ directly to a sample $\mathbf{x} = g_\phi(\mathbf{z})$. When using a neural sampler as $q_\phi(\mathbf{x})$, it is efficient to draw samples through the deterministic mapping, but $H(q_\phi(\mathbf{x}))$ is intractable since the density of $q_\phi(\mathbf{x})$ is unknown. Kim and Bengio [KB16] and Zhai et al. [Zha+16] propose several heuristics to approximate this entropy function. Kumar et al. [Kum+19c] propose to estimate the entropy through its connection to mutual information: $H(q_\phi(\mathbf{z})) = I(g_\phi(\mathbf{z}), \mathbf{z})$, which can be estimated from samples with variational lower bounds [NWJ10b; NCT16b]. Dai et al. [Dai+19a] noticed that when defining $p_\theta(\mathbf{x}) = p_0(\mathbf{x})e^{-\mathcal{E}_\theta(\mathbf{x})}/Z_\theta$, with $p_0(\mathbf{x})$ being a fixed base distribution, the entropy term $-H(q_\phi(\mathbf{x}))$ in Equation (24.64) can be replaced by $D_{\text{KL}}(q_\phi(\mathbf{x}) \parallel p_0(\mathbf{x}))$, which can also be approximated with variational lower bounds using samples from $q_\phi(\mathbf{x})$ and $p_0(\mathbf{x})$, without requiring the density of $q_\phi(\mathbf{x})$.

Grathwohl et al. [Gra+20a] represent $q_\phi(\mathbf{x})$ as a noisy neural sampler, where samples are obtained via $g_\phi(\mathbf{z}) + \sigma\epsilon$, assuming $\mathbf{z}, \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. With a noisy neural sampler, $\nabla_\phi H(q_\phi(\mathbf{x}))$ becomes particularly easy to estimate, which allows gradient-based optimization for the minimax objective in Equation (24.63). A related approach is proposed in Xie et al. [Xie+18], where authors train a noisy neural sampler with samples obtained from MCMC, and initialize new MCMC chains with samples generated from the neural sampler. This cooperative sampling scheme improves the convergence of MCMC, but may still require multiple MCMC steps for sample generation. It does not optimize the objective in Equation (24.63).

When using both adversarial training and MCMC sampling, Yu et al. [Yu+20] noticed that EBMs can be trained with an arbitrary f -divergence, including KL, reverse KL, total variation, Hellinger, etc. The method proposed by Yu et al. [Yu+20] allows us to explore the trade-offs and inductive bias of different statistical divergences for more flexible EBM training.

