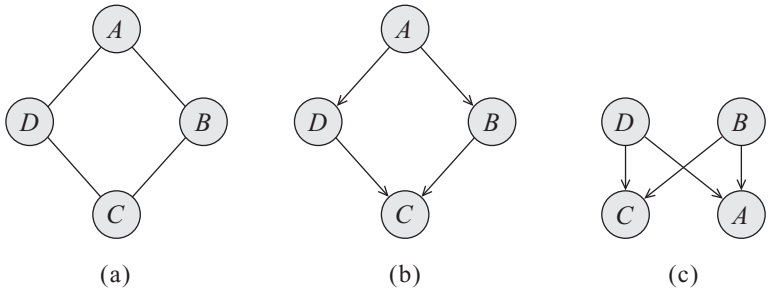


Figure 4.37: A UPGM and two failed attempts to represent it as a DPGM. From Figure 3.10 of [KF09a]. Used with kind permission of Daphne Koller.

the arrows, we get $A - C - B$, which asserts $A \perp B|C$ and $A \not\perp B$, which is not consistent with the independence statements encoded by the DPGM. In fact, there is no UPGM that can precisely represent all and only the two CI statements encoded by a v-structure. In general, CI properties in UPGMs are monotonic, in the following sense: if $A \perp B|C$, then $A \perp B|(C \cup D)$. But in DPGMs, CI properties can be non-monotonic, since conditioning on extra variables can eliminate conditional independencies due to explaining away.

As an example of some CI relationships that can be perfectly modeled by a UPGM but not a DPGM, consider the 4-cycle shown in Figure 4.37(a). One attempt to model this with a DPGM is shown in Figure 4.37(b). This correctly asserts that $A \perp C|B, D$. However, it incorrectly asserts that $B \perp D|A$. Figure 4.37(c) is another incorrect DPGM: it correctly encodes $A \perp C|B, D$, but incorrectly encodes $B \perp D$. In fact there is no DPGM that can precisely represent all and only the CI statements encoded by this UPGM.

Some distributions can be perfectly modeled by either a DPGM or a UPGM; the resulting graphs are called **decomposable** or **chordal**. Roughly speaking, this means the following: if we collapse together all the variables in each maximal clique, to make "mega-variables", the resulting graph will be a tree. Of course, if the graph is already a tree (which includes chains as a special case), it will already be chordal.

4.5.2 Converting between a directed and undirected model

Although DPGMs and UPGMs are not in general equivalent, if we are willing to allow the graph to encode fewer CI properties than may strictly hold, then we can safely convert one to the other, as we explain below.

4.5.2.1 Converting a DPGM to a UPGM

We can easily convert a DPGM to a UPGM as follows. First, any "unmarried" parents that share a child must get "married", by adding an edge between them; this process is known as **moralization**. Then we can drop the arrows, resulting in an undirected graph. The reason we need to do this is to ensure that the CI properties of the UGM match those of the DGM, as explained in Section 4.3.6.2. It also ensures there is a clique that can "store" the CPDs of each family.

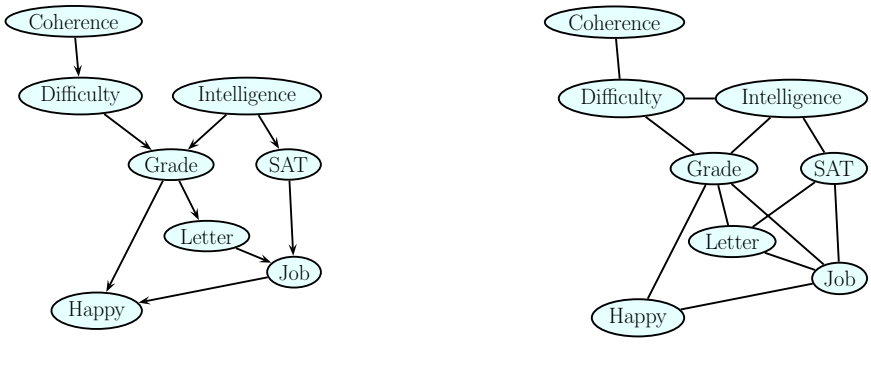


Figure 4.38: Left: The full student DPGM. Right: the equivalent UPGM. We add moralization arcs D-I, G-J and L-S. Adapted from Figure 9.8 of [KF09a].

Let us consider an example from [KF09a]. We will use the (full version of the student network shown in Figure 4.38(a). The corresponding joint has the following form:

$$P(C, D, I, G, S, L, J, H) \tag{4.149}$$
$$= P(C)P(D|C)P(I)P(G|I, D)P(S|I)P(L|G)P(J|L, S)P(H|G, J) \tag{4.150}$$

Next, we define a potential or factor for every CPD, yielding

$$p(C, D, I, G, S, L, J, H) = \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, I, D) \tag{4.151}$$
$$\psi_S(S, I)\psi_L(L, G)\psi_J(J, L, S)\psi_H(H, G, J) \tag{4.152}$$

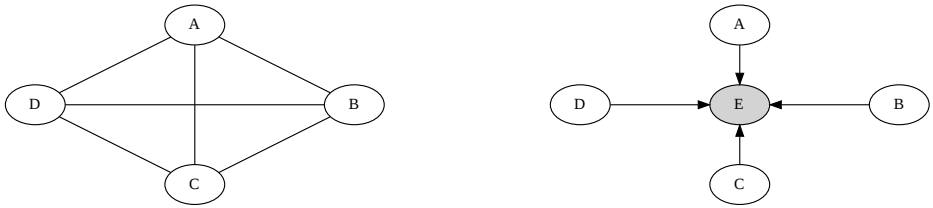
All the potentials are **locally normalized**, since they are CPDs, there is no need for a global normalization constant, so $Z = 1$. The corresponding undirected graph is shown in Figure 4.38(b). We see the that we have added D-I, G-J, and L-S moralization edges.[11]

4.5.2.2 Converting a UPGM to a DPGM

To convert a UPGM to a DPGM, we proceed as follows. For each potential function $\psi_c(\boldsymbol{x}_c; \boldsymbol{\theta}_c)$, we create a "dummy node", call it Y_c, which is "clamped" to a special observed state, call it y_c^*. We then define $p(Y_c = y_c^*|\boldsymbol{x}_c) = \psi_c(\boldsymbol{x}_c; \boldsymbol{\theta}_c)$. This "local evidence" CPD encodes the same factor as in the DGM. The overall joint has the form $p_{\text{undir}}(\boldsymbol{x}) \propto p_{\text{dir}}(\boldsymbol{x}, \boldsymbol{y}^*)$.

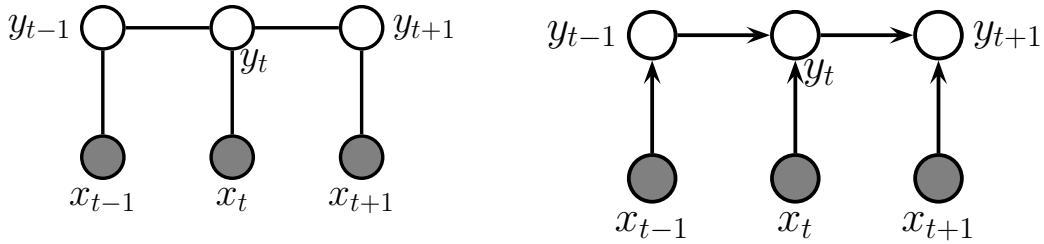
As an example, consider the UPGM in Figure 4.39(a), which defines the joint $p(A, B, C, D) = \psi(A, B, C, D)/Z$. We can represent this as a DPGM by adding a dummy E node, which is a child of all the other nodes. We set $E = 1$ and define the CPD $p(E = 1|A, B, C, D) \propto \psi(A, B, C, D)$. By conditioning on this observed child, all the parents become dependent, as in the UGM.

11. We will see this example again in Section 9.4, where we use it to illustrate the variable elimination inference algorithm.



(a) *(b)*

Figure 4.39: (a) An undirected graphical model. (b) A directed equivalent, obtained by adding a dummy observed child node.



(a) *(b)*

Figure 4.40: Two discriminative models for sequential data. (a) An undirected model (CRF). (b) A directed model (MEMM).

4.5.3 Conditional directed vs undirected PGMs and the label bias problem

Directed and undirected models behave somewhat differently in the conditional (discriminative) setting. As an example of this, let us compare the 1d undirected CRF in Figure 4.40a with the directed Markov chain in Section 4.5.3. (This latter model is called a maximum entropy Markov model (MEMM), which is a reference to the connection with maxent models discussed in Section 4.3.4.) The MEMM suffers from a subtle problem compared to the CRF known (rather obscurely) as the **label bias** problem [LMP01]. The problem is that local features at time t do not influence states prior to time t. That is, $y_{t-1} \perp \boldsymbol{x}_t | y_t$, thus blocking information flow backwards in time.

To understand what this means in practice, consider the part of speech tagging task which we discussed in Section 4.4.1.1. Suppose we see the word "banks"; this could be a verb (as in "he banks at Chase"), or a noun (as in "the river banks were overflowing"). Locally the part of speech tag for the word is ambiguous. However, suppose that later in the sentence, we see the word "fishing"; this gives us enough context to infer that the sense of "banks" is "river banks" and not "financial banks". However, in an MEMM the "fishing" evidence will not flow backwards, so we will not be able to infer the correct label for "banks". The CRF does not have this problem.

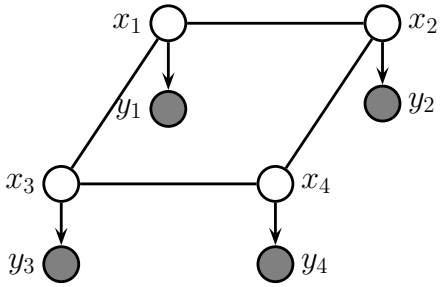


Figure 4.41: A grid-structured MRF with hidden nodes x_i and local evidence nodes y_i. The prior $p(\boldsymbol{x})$ is an undirected Ising model, and the likelihood $p(\boldsymbol{y}|\boldsymbol{x}) = \prod_i p(y_i|x_i)$ is a directed fully factored model.

The label bias problem in MEMMs occurs because directed models are **locally normalized**, meaning each CPD sums to 1. By contrast, MRFs and CRFs are **globally normalized**, which means that local factors do not need to sum to 1, since the partition function Z, which sums over all joint configurations, will ensure the model defines a valid distribution.

However, this solution comes at a price: in a CRF, we do not get a valid probability distribution over $\boldsymbol{y}_{1:T}$ until we have seen the whole sentence, since only then can we normalize over all configurations. Consequently, CRFs are not as useful as directed probabilistic graphical models (DPGM) for online or real-time inference. Furthermore, the fact that Z is a function of all the parameters makes CRFs less modular and much slower to train than DPGM's, as we discuss in Section 4.4.3.

4.5.4 Combining directed and undirected graphs

We can also define graphical models that contain directed and undirected edges. We discuss a few examples below.

4.5.4.1 Chain graphs

A **chain graph** is a PGM which may have both directed and undirected edges, but without any directed cycles. A simple example is shown in Figure 4.41, which defines the following joint model:

$$p(\boldsymbol{x}_{1:D}, \boldsymbol{y}_{1:D}) = p(\boldsymbol{x}_{1:D})p(\boldsymbol{y}_{1:D}|\boldsymbol{x}_{1:D}) = \left[\frac{1}{Z}\prod_{i\sim j}\psi_{ij}(x_i, x_j)\right]\left[\prod_{i=1}^{D} p(y_i|x_i)\right] \tag{4.153}$$

In this example, the prior $p(\boldsymbol{x})$ is specified by a UPGM, and the likelihood $p(\boldsymbol{y}|\boldsymbol{x})$ is specified as a fully factorized DPGM.

More generally, a chain graph can be defined in terms of a **partially directed acyclic graph** (**PDAG**). This is a graph which can be decomposed into a directed graph of **chain components**, where the nodes within each chain component are connected with each other only with undirected edges. See Figure 4.42 for an example.

We can use a PDAG to define a joint distribution using $\prod_i p(C_i|\text{pa}(C_i))$, where each C_i is a chain component, and each CPD is a conditional random field. For example, referring to Figure 4.42, we

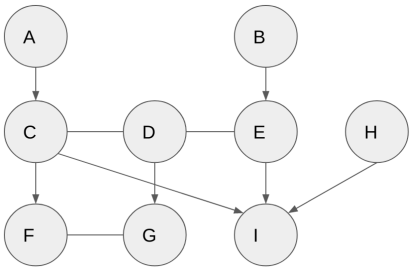


Figure 4.42: A partially directed acyclic graph (PDAG). The chain components are $\{A\}$, $\{B\}$, $\{C, D, E\}$, $\{F, G\}$, $\{H\}$ and $\{I\}$. Adapted from Figure 4.15 of [KF09a].

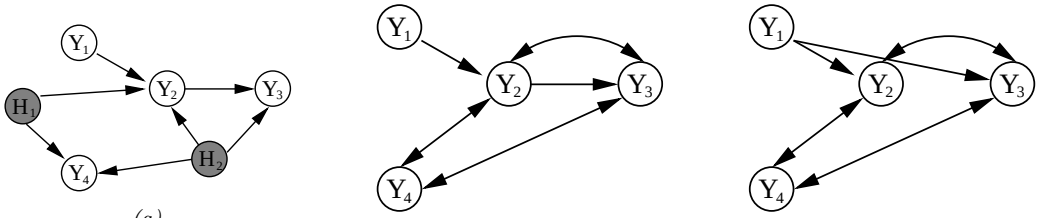


Figure 4.43: (a) A DAG with two hidden variables (shaded). (b) The corresponding ADMG. The bidirected edges reflect correlation due to the hidden variable. (c) A Markov equivalent ADMG. From Figure 3 of [SG09]. Used with kind permission of Ricardo Silva.

have

$$p(A, B, ..., I) = p(A)p(B)p(C, D, E|A, B)p(F, G|C, D)p(H)p(I|C, E, H) \tag{4.154}$$

$$p(C, D, E|A, B) = \frac{1}{Z(A, B)}\phi(A, C)\phi(B, E)\phi(C, D)\phi(D, E) \tag{4.155}$$

$$p(F, G|C, D) = \frac{1}{Z(C, D)}\phi(F, C)\phi(G, D)\phi(F, G) \tag{4.156}$$

For more details, see e.g., [KF09a, Sec 4.6.2].

4.5.4.2 Acyclic directed mixed graphs

One can show [Pea09b, p51] that every latent variable DPGM can be rewritten in a way such that every latent variable is a root node with exactly two observed children. This is called the **projection** of the latent variable PGM, and is observationally indistinguishable from the original model.

Each such latent variable root node induces a dependence between its two children. We can represent this with a directed arc. The resulting graph is called an **acyclic directed mixed graph** or **ADMG**. See Figure 4.43 for an example. (A **mixed graph** is one with undirected, unidirected, and bidirected edges.)

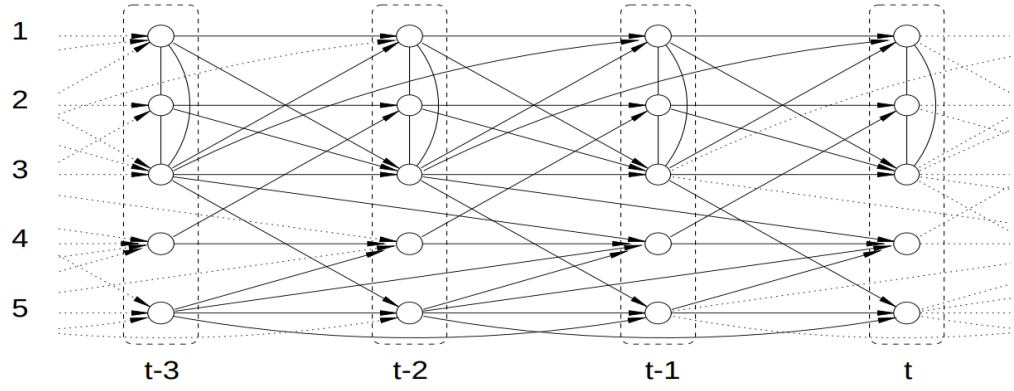


Figure 4.44: A VAR(2) process represented as a dynamic chain graph. From [DE00]. Used with kind permission of Rainer Dahlhaus.

One can determine CI properties of ADMGs using a technique called **m-separation** [Ric03]. This is equivalent to d-separation in a graph where every bidirected edge $Y_i \leftrightarrow Y_j$ is replaced by $Y_i \leftarrow X_{ij} \rightarrow Y_j$, where X_{ij} is a hidden variable for that edge.

The most common example of ADMGs is when everything is linear-Gaussian. This is known as a structural equation model and is discussed in Section 4.7.2.

4.5.5 Comparing directed and undirected Gaussian PGMs

In this section, we compare directed and undirected Gaussian graphical models. In Section 4.2.3, we saw that directed GGMs correspond to sparse regression matrices. In Section 4.3.5, we saw that undirected GGMs correspond to sparse precision matrices.

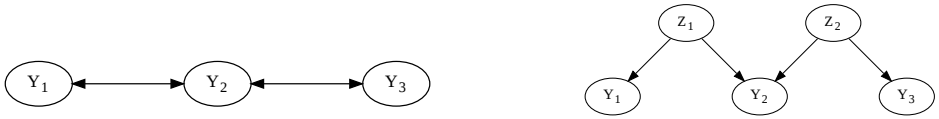
The advantage of the DAG formulation is that we can make the regression weights $\mathbf{W}$, and hence $\mathbf{\Sigma}$, be conditional on covariate information [Pou04], without worrying about positive definite constraints. The disadavantage of the DAG formulation is its dependence on the order, although in certain domains, such as time series, there is already a natural ordering of the variables.

It is actually possible to combine both directed and undirected representations, resulting in a model known as a (Gaussian) **chain graph**. For example, consider a discrete-time, second-order Markov chain in which the observations are continuous, $\boldsymbol{x}_t \in \mathbb{R}^D$. The transition function can be represented as a (vector-valued) linear-Gaussian CPD:

$$p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{x}_t|\mathbf{A}_1\boldsymbol{x}_{t-1} + \mathbf{A}_2\boldsymbol{x}_{t-2}, \mathbf{\Sigma}) \tag{4.157}$$

This is called **vector auto-regressive** or **VAR** process of order 2. Such models are widely used in econometrics for time-series forecasting.

The time series aspect is most naturally modeled using a DPGM. However, if $\mathbf{\Sigma}^{-1}$ is sparse, then the correlation amongst the components within a time slice is most naturally modeled using a UPGM.



(a) *(b)*

Figure 4.45: (a) A bi-directed graph. (b) The equivalent DAG. Here the z nodes are latent confounders. Adapted from Figures 5.12-5.13 of [Cho11].

For example, suppose we have

$$\mathbf{A}_1 = \begin{pmatrix} \frac{3}{5} & 0 & \frac{1}{5} & 0 & 0 \\ 0 & \frac{3}{5} & 0 & -\frac{1}{5} & 0 \\ \frac{2}{5} & \frac{1}{3} & \frac{3}{5} & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{5} \\ 0 & 0 & \frac{1}{5} & 0 & \frac{2}{5} \end{pmatrix}, \; \mathbf{A}_2 = \begin{pmatrix} 0 & 0 & -\frac{1}{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{5} \end{pmatrix} \tag{4.158}$$

and

$$\mathbf{\Sigma} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & 0 & 0 \\ \frac{1}{2} & 1 & -\frac{1}{3} & 0 & 0 \\ \frac{1}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \; \mathbf{\Sigma}^{-1} = \begin{pmatrix} 2.13 & -1.47 & -1.2 & 0 & 0 \\ -1.47 & 2.13 & 1.2 & 0 & 0 \\ -1.2 & 1.2 & 1.8 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.159}$$

The resulting graphical model is illustrated in Figure 4.44. Zeros in the transition matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ correspond to absent directed arcs from $\boldsymbol{x}_{t-1}$ and $\boldsymbol{x}_{t-2}$ into $\boldsymbol{x}_t$. Zeros in the precision matrix $\mathbf{\Sigma}^{-1}$ correspond to absent undirected arcs between nodes in $\boldsymbol{x}_t$.

4.5.5.1 Covariance graphs

Sometimes we have a sparse covariance matrix rather than a sparse precision matrix. This can be represented using a **bi-directed graph**, where each edge has arrows in both directions, as in Figure 4.45(a). Here nodes that are not connected are unconditionally independent. For example in Figure 4.45(a) we see that $Y_1 \perp Y_3$. In the Gaussian case, this means $\Sigma_{1,3} = \Sigma_{3,1} = 0$. (A graph representing a sparse covariance matrix is called a **covariance graph**, see e.g., [Pen13]). By contrast, if this were an undirected model, we would have that $Y_1 \perp Y_3|Y_2$, and $\Lambda_{1,3} = \Lambda_{3,1} = 0$, where $\mathbf{\Lambda} = \mathbf{\Sigma}^{-1}$.

A bidirected graph can be converted to a DAG with latent variables, where each bidirected edge is replaced with a hidden variable representing a hidden common cause, or **confounder**, as illustrated in Figure 4.45(b). The relevant CI properties can then be determined using d-separation.

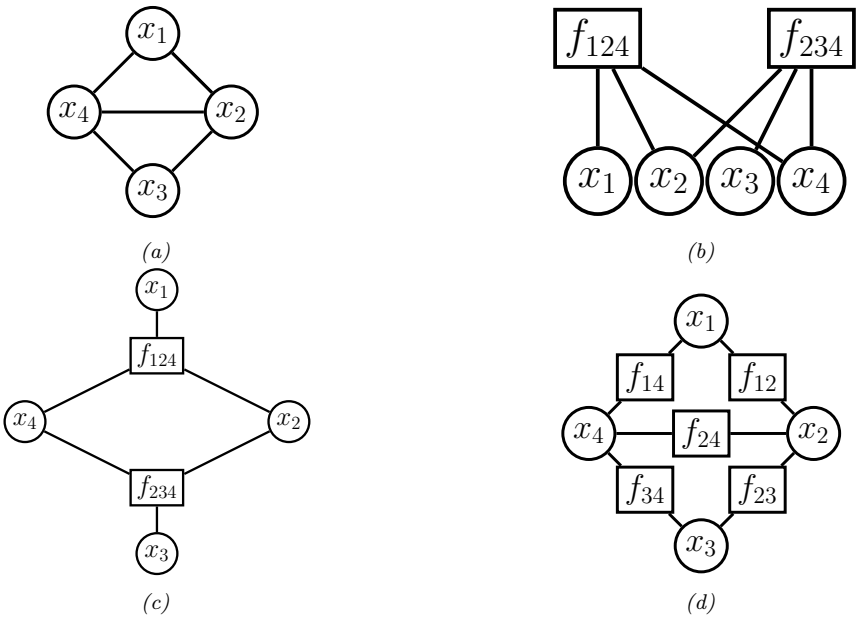


Figure 4.46: (a) A simple UPGM. (b) A factor graph representation assuming one potential per maximal clique. (c) Same as (b), but graph is visualized differently. (d) A factor graph representation assuming one potential per edge.

4.6 PGM extensions

In this section, we discuss some extensions of the basic PGM framework.

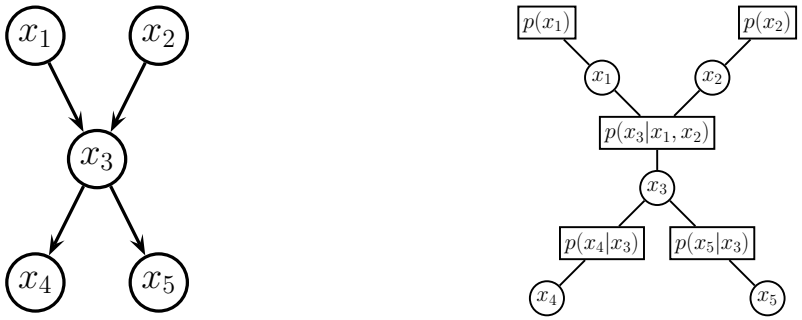
4.6.1 Factor graphs

A **factor graph** [KFL01; Loe04] is a graphical representation that unifies directed and undirected models. They come in two main "flavors". The original version uses a bipartite graph, where we have nodes for random variables and nodes for factors, as we discuss in Section 4.6.1.1. An alternative form, known as a **Forney factor graphs** [For01] just has nodes for factors, and the variables are associated with edges, as we explain in Section 4.6.1.2.

4.6.1.1 Bipartite factor graphs

A **factor graph** is an undirected bipartite graph with two kinds of nodes. Round nodes represent variables, square nodes represent factors, and there is an edge from each variable to every factor that mentions it. For example, consider the MRF in Figure 4.46(a). If we assume one potential per maximal clique, we get the factor graph in Figure 4.46(b), which represents the function

$$f(x_1, x_2, x_3, x_4) = f_{124}(x_1, x_2, x_4) f_{234}(x_2, x_3, x_4) \tag{4.160}$$



(a) *(b)*

Figure 4.47: (a) A simple DPGM. (b) Its corresponding factor graph.

We can represent this in a topologically equivalent way as in Figure 4.46(c).

One advantage of factor graphs over UPGM diagrams is that they are more fine-grained. For example, suppose we associate one potential per edge, rather than per clique. In this case, we get the factor graph in Figure 4.46(d), which represents the function

$$f(x_1, x_2, x_3, x_4) = f_{14}(x_1, x_4) f_{12}(x_1, x_2) f_{34}(x_3, x_4) f_{23}(x_2, x_3) f_{24}(x_2, x_4) \tag{4.161}$$

We can also convert a DPGM to a factor graph: just create one factor per CPD, and connect that factor to all the variables that use that CPD. For example, Figure 4.47 represents the following factorization:

$$f(x_1, x_2, x_3, x_4, x_5) = f_1(x_1) f_2(x_2) f_{123}(x_1, x_2, x_3) f_{34}(x_3, x_4) f_{35}(x_3, x_5) \tag{4.162}$$

where we define $f_{123}(x_1, x_2, x_3) = p(x_3|x_1, x_2)$, etc. If each node has at most one parent (and hence the graph is a chain or simple tree), then there will be one factor per edge (root nodes can have their prior CPDs absorbed into their children's factors). Such models are equivalent to pairwise MRFs.

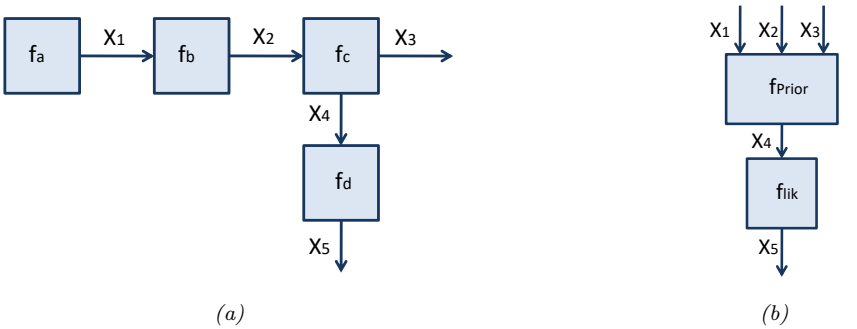
4.6.1.2 Forney factor graphs

A **Forney factor graph** (**FFG**), also called a **normal factor graph**, is a graph in which nodes represent factors, and edges represent variables [For01; Loe04; Loe+07; CLV19]. This is more similar to standard neural network diagrams, and electrical engineering diagrams, where signals (represented as electronic pulses, or tensors, or probability distributions) propagate along wires and are modified by functions represented as nodes.

For example, consider the following factorized function:

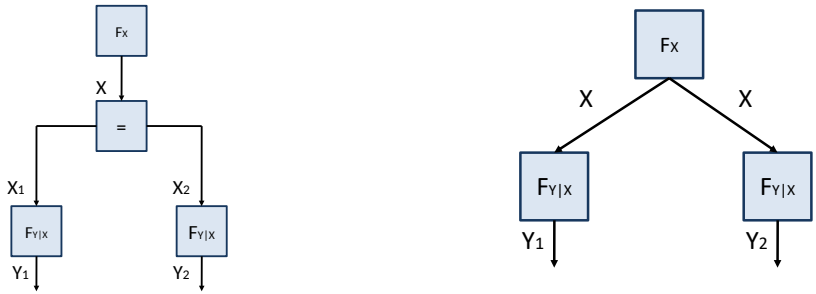
$$f(x_1, \ldots, x_5) = f_a(x_1) f_b(x_1, x_2) f_c(x_2, x_3, x_4) f_d(x_4, x_5) \tag{4.163}$$

We can visualize this as an FFG as in Figure 4.48a. The edge labeled x_3 is called a **half-edge**, since it is only connected to one node; this is because x_3 only participates in one factor. (Similarly for x_5.)



(a) (b)

Figure 4.48: A Forney factor graph. (a) Directed version. (b) Hierarchical version.



(a) (b)

Figure 4.49: An FFG with an equality constraint node (left) and its corresponding simplified form (right).

The directionality associated with the edges is a useful mnemonic device if there is a natural order in which the variables are generated. In addition, associating directions with each edge allows us to uniquely name "messages" that are sent along each edge, which will prove useful when we discuss inference algorithms in Section 9.2.

In addition to being more similar to neural network diagrams, FFGs have the advantage over bipartite FGs in that they support hierarchical (compositional) construction, in which complex dependency structure between variables can be represented as a blackbox, with the input/output interface being represented by edges corresponding to the variables exposed by the blackbox. See Figure 4.48b for an example, which represents the function

$$f(x_1, \ldots, x_5) = f_{\text{prior}}(x_1, x_2, x_3, x_4) f_{\text{lik}}(x_4, x_5) \tag{4.164}$$

The factor f_{prior} represents a (potentially complex) joint distribution $p(x_1, x_2, x_3, x_4)$, and the factor f_{lik} represents the likelihood term $p(x_5|x_4)$. Such models are widely used to build error-correcting codes (see Section 9.3.8).

To allow for variables to participate in more than 2 factors, equality constraint nodes are introduced, as illustrated in Figure 4.49(a). Formally, this is a factor defined as follows:

$$f_{=}(x, x_1, x_2) = \delta(x - x_1)\delta(x - x_2) \tag{4.165}$$

where $\delta(u)$ is a Dirac delta if u is continuous, and a Kronecker delta if u is discrete. The effect of this factor is to ensure all the variables connected to the factor have the same value; intuitively, this factor acts like a "wire splitter". Thus the function represented in Figure 4.49(a) is equivalent to the following:

$$f(x, y_1, y_2) = f_x(x) f_{y|x}(y_1, x) f_{y|x}(y_2, x) \tag{4.166}$$

This simplified form is represented in Figure 4.49(b), where we reuse the x variable across multiple edges. We have chosen the edge orientations to reflect our interpretation of the factors $f_{y|x}(y, x)$ as likelihood terms, $p(y|x)$. We have also chosen to reuse the same $f_{y|x}$ factor for both y variables; this is an example of **parameter tying**.

4.6.2 Probabilistic circuits

A **probabilistic circuit** is a kind of graphical model that supports efficient exact inference. It includes **arithmetic circuits** [Dar03; Dar09], **sum-product networks** (SPNs) [PD11; SCPD22]. and other kinds of model.

Here we briefly describe SPNs. An SPN is a probabilistic model, based on a directed tree-structured graph, in which terminal nodes represent univariate probability distributions and non-terminal nodes represent convex combinations (weighted sums) and products of probability functions. SPNs are similar to deep mixture models, in which we combine together dimensions. SPNs leverage context-specific independence to reduce the complexity of exact inference to time that is proportional to the number of links in the graph, as opposed to the treewidth of the graph (see Section 9.4.2).

SPNs are particularly useful for tasks such as missing data imputation of tabular data (see e.g., [Cla20; Ver+19]). A recent extension of SPNs, known as **einsum networks**, is proposed in [Peh+20] (see Section 9.6.1 for details on the connection between einstein summation and PGM inference).

4.6.3 Directed relational PGMs

A Bayesian network defines a joint probability distribution over a fixed number of random variables. By using plate notation (Section 4.2.8), we can define models with certain kinds of repetitive structure, and tied parameters, but many models are not expressible in this way. For example, it is not possible to represent even a simple HMM using plate notation (see Figure 29.12). Various notational extensions of plates have been proposed to handle repeated structure (see e.g., [HMK04; Die10]) but have not been widely adopted. The problem becomes worse when we have more complex domains, involving multiple objects which interact via multiple relationships.[12] Such models are called **relational probability models** or **RPMs**. In this section, we focus on directed RPMs; see Section 4.6.4 for the undirected case.

As in first order logic, RPMs have constant symbols (representing objects), function symbols (mapping one set of constants to another), and predicate symbols (representing relations between objects). We will assume that each function has a **type signature**. To illustrate this, consider an example from [RN19, Sec 15.1], which concerns online book reviews on sites such as Amazon. Suppose

12. See e.g., this blog post from Rob Zinkov: https://www.zinkov.com/posts/2013-07-28-stop-using-plates.

there are two types of objects, Book and Customer, and the following functions and predicates:

$$\text{Honest} : \text{Customer} \to \{\text{True}, \text{False}\} \tag{4.167}$$
$$\text{Kindess} : \text{Customer} \to \{1, 2, 3, 4, 5\} \tag{4.168}$$
$$\text{Quality} : \text{Book} \to \{1, 2, 3, 4, 5\} \tag{4.169}$$
$$\text{Recommendation} : \text{Customer} \times \text{Book} \to \{1, 2, 3, 4, 5\} \tag{4.170}$$

The constant symbols refer to specific objects. To keep things simple, we assume there are two books, B_1 and B_2, and two customers, C_1 and C_2. The **basic random variables** are obtained by instantiating each function with each possible combination of objects to create a set of **ground terms**. In this example, these variables are $H(C_1)$, $Q(B_1)$, $R(C_1, B_2)$, etc. (We use the abbreviations H, K, Q and R for the functions Honest, Kindness, Quality and Recommendation.[13])

We now need to specify the (conditional) distribution over these random variables. We define these distributions in terms of the generic indexed form of the variables, rather than the specific ground form. For example, we may use the following priors for the root nodes (variables with no parents):

$$H(c) \sim \text{Cat}(0.99, 0.01) \tag{4.171}$$
$$K(c) \sim \text{Cat}(0.1, 0.1, 0.2, 0.3, 0.3) \tag{4.172}$$
$$Q(b) \sim \text{Cat}(0.05, 0.2, 0.4, 0.2, 0.15) \tag{4.173}$$

For the recommendation nodes, we need to define a conditional distribution of the form

$$R(c, b) \sim \text{RecCPD}(H(c), K(c), Q(b)) \tag{4.174}$$

where RecCPD is the conditional probability distribution (CPD) for the recommendation node. If represented as a conditional probability table (CPT), this has $2 \times 5 \times 5 = 50$ rows, each with 5 entries. This table can encode our assumptions about what kind of ratings a book receives based on the quality of the book, but also properties of the reviewer, such as their honest and kindness. (More sophisticated models of human raters in the context of crowd-sourced data collection can be found in e.g., [LRC19].)

We can convert the above formulae into a a graphical model "**template**", as shown in Figure 4.50a. Given a set of objects, we can "**unroll**" the template to create a "**ground network**", as shown in Figure 4.50b. There are $C \times B + 2C + B$ random variables, with a corresonding joint state space (set of **possible worlds**) of size $2^C 5^{C+B+BC}$, which can get quite large. However, if we are only interested in answering specific queries, we can dynamically unroll small pieces of the network that are relevant to that query [GC90; Bre92].

Let us assume that only a subset of the $R(c, b)$ entries are observed, and we would like to predict the missing entries of this matrix. This is essentially a simplified **recommender system**. (Unfortunately it ignores key aspects of the problem, such as the content/topic of the books, and the interests/preferences of the customers.) We can use standard probabilistic inference methods for graphical models (which we discuss in Chapter 9) to solve this problem.

Things get more interesting when we don't know which objects are being referred to. For example, customer C_1 might write a review of a book called "Probabilistic Machine Learning", but do they

13. A unary function of an object that returns a basic type, such as Boolean or an integer, is often called an **attribute** of that object.

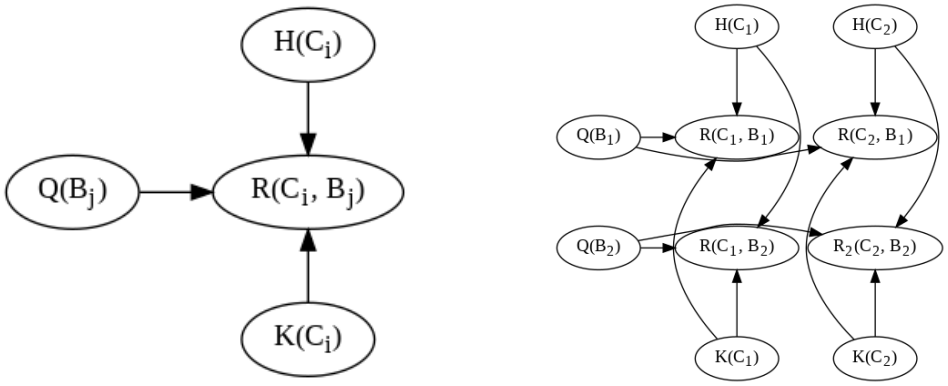


Figure 4.50: RPM for the book review domain. (a) Template for a generic customer C_i and book B_j pair. R is rating, Q is quality, H is honesty, and K is kindness. (b) Unrolled model for 2 books and 2 customers.

mean edition 1 (B_1) or edition 2 (B_2)? To handle this kind of **relational uncertainty**, we can add all possible referents as parents to each relation. This is illustrated in Figure 4.51, where now $Q(B_1)$ and $Q(B_2)$ are both parents of $R(C_1, B_1)$. This is necessary because their review score might either depend on $Q(B_1)$ or $Q(B_2)$, depending on which edition they are writing about. To disambiguate this, we create a new variable, $L(C_i)$, which specifies which version number of each book customer i is referring to. The new CPD for the recommendation node, $p(R(c,b)|H(c), K(c), Q(1:B), L(c))$, has the form

$$R(c,b) \sim \text{RecCPT}(H(c), K(c), Q(b')) \text{ where } b' = L(c) \tag{4.175}$$

This CPD acts like a **multiplexer**, where the $L(c)$ node specifies which of the parents $Q(1:B)$ to actually use.

Although the above problem may seem contrived, **identity uncertainty** is a widespread problem in many areas, such as citation analysis, credit card histories, and object tracking (see Section 4.6.5). In particular, the problem of **entity resolution** or **record linkage** — which refers to the task of mapping particular strings (such as names) to particular objects (such as people) — is a whole field of research (see e.g., https://en.wikipedia.org/wiki/Record_linkage for an overview and [SHF15] for a Bayesian approach).

4.6.4 Undirected relational PGMs

We can create **relational UGMs** in a manner which is analogous to relational DGMs (Section 4.6.3). This is particularly useful in the discriminative setting, for the same reasons that undirected CRFs are preferable to conditional DGMs (see Section 4.4).

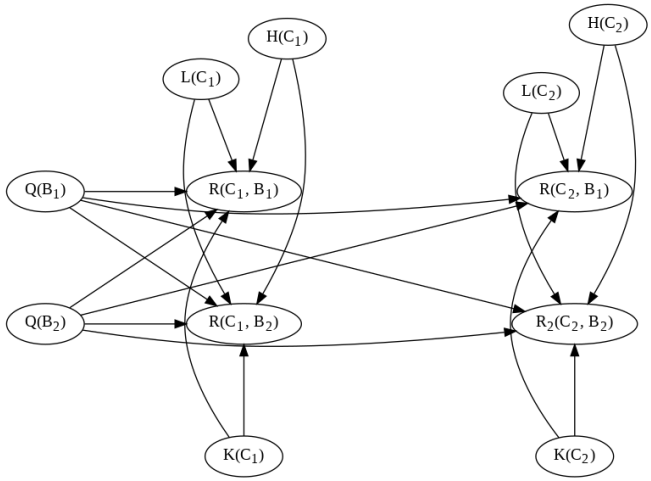


Figure 4.51: An extension of the book review RPM to handle identity uncertainty about which book a given customer is actually reviewing. The $R(c, b)$ node now depends on all books, since we don't know which one is being referred to. We can select one of these parents based on the mapping specified by the user's library, $L(c)$.

4.6.4.1 Collective classification

As an example of a relational UGM, suppose we are interested in the problem of classifying web pages of a university into types (e.g., student, professor, admin, etc.) Obviously we can do this based on the contents of the page (e.g., words, pictures, layout, etc.) However, we might also suppose there is information in the hyper-link structure itself. For example, it might be likely for students to cite professors, and professors to cite other professors, but there may be no links between admin pages and students / professors. When faced with a web page whose label is ambiguous, we can bias our estimate based on the estimated labels of its neighbors, as in a CRF. This process is known as **collective classification** (see e.g., [Sen+08]). To specify the CRF structure for a web-graph of arbitrary size and shape, we just specify a template graph and potential functions, and then unroll the template appropriately to match the topology of the web, making use of parameter tying.

4.6.4.2 Markov logic networks

One particularly popular way of specifying relational UGMs is to use **first-order logic** rather than a graphical description of the template. The result is known as a **Markov logic network** [RD06; Dom+06; DL09].

For example, consider the sentences "Smoking causes cancer" and "If two people are friends, and one smokes, then so does the other". We can write these sentences in first-order logic as follows:

$$\forall x. Sm(x) \implies Ca(x) \tag{4.176}$$

$$\forall x. \forall y. Fr(x, y) \wedge Sm(x) \implies Sm(y) \tag{4.177}$$

where Sm and Ca are predicates, and Fr is a relation.

Fr(A,A)	Fr(B,B)	Fr(B,A)	Fr(A,B)	Sm(A)	Sm(B)	Ca(A)	Ca(B)
1	1	0	1	1	1	1	1
1	1	0	1	1	0	0	0
1	1	0	1	1	1	0	1

Table 4.5: Some possible joint instantiations of the 8 variables in the smoking example.

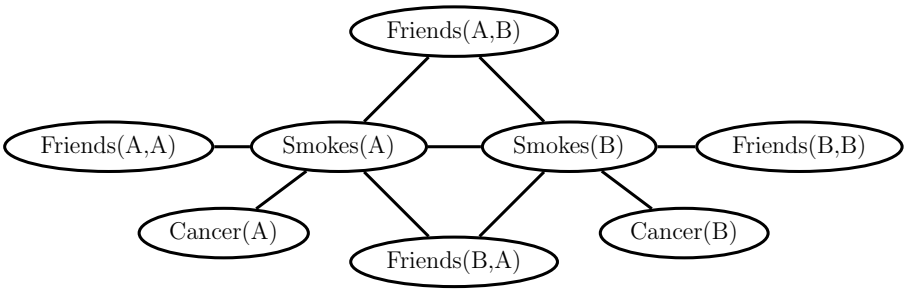


Figure 4.52: An example of a ground Markov logic network represented as a pairwise MRF for 2 people. Adapted from Figure 2.1 from [DL09]. Used with kind permission of Pedro Domingos.

It is convenient to write all formulas in **conjunctive normal form** (CNF), also known as **clausal form**. In this case, we get

$$\neg Sm(x) \vee Ca(x) \tag{4.178}$$

$$\neg Fr(x,y) \vee \neg Sm(x) \vee Sm(y) \tag{4.179}$$

The first clause can be read as "Either x does not smoke or he has cancer", which is logically equivalent to Equation (4.176). (Note that in a clause, any unbound variable, such as x, is assumed to be universally quantified.)

Suppose there are just two objects (people) in the world, Anna and Bob, which we will denote by **constant symbols** A and B. We can then create 8 binary random variables $Sm(x)$, $Ca(x)$, and $Fr(x,y)$ for $x, y \in \{A, B\}$. This defines 2^8 **possible worlds**, some of which are shown in Table 4.5.[14]

Our goal is to define a probability distribution over these joint assignments. We can do this by creating a UGM with these variables, and adding a potential function to capture each logical rule or constraint. For example, we can encode the rule $\neg Sm(x) \vee Ca(x)$ by creating a potential function

14. Note that we have not encoded the fact that Fr is a symmetric relation, so $Fr(A,B)$ and $Fr(B,A)$ might have different values. Similarly, we have the "degenerate" nodes $Fr(A)$ and $Fr(B)$, since we did not enforce $x \neq y$ in Equation (4.177). (If we add such constraints, then the model compiler, which generates the ground network, should avoid creating redundant nodes.)

$\Psi(Sm(x), Ca(x))$, where we define

$$\Psi(Sm(x), Ca(x)) = \begin{cases} 1 & \text{if } \neg Sm(x) \vee Ca(x) = T \\ 0 & \text{if } \neg Sm(x) \vee Ca(x) = F \end{cases} \tag{4.180}$$

The result is the UGM in Figure 4.52.

The above approach will assign non-zero probability to all logically valid worlds. However, logical rules may not always be true. For example, smoking does not always cause cancer. We can relax the hard constraints by using non-zero potential functions. In particular, we can associate a weight with each rule, and thus get potentials such as

$$\Psi(Sm(x), Ca(x)) = \begin{cases} e^w & \text{if } \neg Sm(x) \vee Ca(x) = T \\ e^0 & \text{if } \neg Sm(x) \vee Ca(x) = F \end{cases} \tag{4.181}$$

where the value of $w > 0$ controls strongly we want to enforce the corresponding rule.

The overall joint distribution has the form

$$p(\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{w})} \exp(\sum_i w_i n_i(\boldsymbol{x})) \tag{4.182}$$

where $n_i(\boldsymbol{x})$ is the number of instances of clause i which evaluate to true in assignment $\boldsymbol{x}$.

Given a grounded MLN model, we can then perform inference using standard methods. Of course, the ground models are often extremely large, so more efficient inference methods, which avoid creating the full ground model (known as **lifted inference**), must be used. See [DL09; KNP11] for details.

One way to gain tractability is to relax the discrete problem to a continuous one. This is the basic idea behind **hinge-loss MRFs** [Bac+15b], which support exact inference using scalable convex optimization. There is a template language for this model family known as **probabilistic soft logic**, which has a similar "flavor" to MLN, although it is not quite as expressive.

Recently MLNs have been combined with DL in various ways. For example, [Zha+20f] uses graph neural networks for inference. And [WP18] uses MLNs for evidence fusion, where the noisy predictions come from DNNs trained using weak supervision.

Finally, it is worth noting one subtlety which arises with undirected models, namely that the size of the unrolled model, which depends on the number of objects in the universe, can affect the results of inference, even if we have no data about the new objects. For example, consider an undirected chain of length T, with T hidden nodes z_t and T observed nodes y_t; call this model M_1. Now suppose we double the length of the chain to $2T$, without adding more evidence; call this model M_2. We find that $p(z_t|y_{1:T}, M_1) \neq p(z_t|y_{1:T}, M_2)$, for $t = 1 : T$, even though we have not added new information, due to the different partition functions. This does not happen with a directed chain, because the newly added nodes can be marginalized out without affecting the original nodes, since the model is locally normalized and therefore modular. See [JBB09; Poo+12] for further discussion.

4.6.5 Open-universe probability models

In Section 4.6.3, we discussed relational probability models, as well as the topic of identity uncertainty. However, we also implicitly made a **closed world assumption**, namely that the set of all objects is fixed and specified ahead of time. In many real world problems, this is an unrealistic assumption.

For example, in Section 29.9.3.5, we discuss the problem of tracking an unknown number of objects over time. As another example, consider the problem of enforcing the UN Comprehensive Nuclear Test Ban Treaty (CTBT). This requires monitoring seismic events, and determinining if they were caused by nature or man-made explosions. Thus the number of objects of each type, as well as their source, is uncertain [ARS13],

As another (more peaceful) example, suppose we want to perform **citation matching**, in which we want to know whether to cite an arxiv version of a paper or the version on some conference website. Are these the same object? It is often hard to tell, since the titles and author might be the same, yet the content may have been updated. It is often necessary to use subtle cues, such as the date stored in the meta-data, to infer if the two "textual measurements" refer to the same underlying object (paper) or not [Pas+02].

In problems such as these, the number of objects of each type, as well as their relationships, is uncertain. This requires the use of **open-universe probability models** or **OUPM**, which can generate new objects as well as their properties [Rus15; MR10; LB19]. The first formal language for OUPMs was **BLOG** [Mil+05], which stands for "Bayesian LOGic". This used a general purpose, but slow, MCMC inference scheme to sample over possible worlds of variable size and shape. [Las08; LLC20] describes another open-universe modeling language called **multi-entity Bayesian networks**.

Very recently, Facebook has released the **Bean Machine** library, available at https://beanmachine.org/, which supports more efficient inference in OUPMs. Details can be found in [Teh+20], as well as their blog post.[15]

4.6.6 Programs as probability models

OUPMs, discussed in Section 4.6.5, let us define probability models over complex dynamic state spaces of unbounded and variable size. The set of possible worlds correspond to objects and their attributes and relationships. Another approach is to use a **probabilistic programming language** or **PPL**, in which we define the set of possible words as the set of **execution traces** generated by the program when it is endowed with a random choice mechanism. (This is a **procedural approach** to the problem, whereas OUPMs are a **declarative approach**.)

The difference between a probabilistic programming language and a standard one was described in [Gor+14] as follows: "Probabilistic programs are usual functional or imperative programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the abiliy to condition values of variables in a program via observation". The former is a way to define $p(\boldsymbol{z}, \boldsymbol{y})$, and the latter is the same as standard Bayesian conditioning $p(\boldsymbol{z}|\boldsymbol{y})$.

Some recent examples of PPLs include **Gen** [CT+19], **Pyro** [Bin+19] and **Turing** [GXG18]. Inference in such models is often based on SMC, which we discuss in Chapter 13. For more details on PPLs, see e.g. [Mee+18].

4.7 Structural causal models

> While probabilities encode our beliefs about a static world, causality tells us whether and how probabilities change when the world changes, be it by intervention or by act of imagination. —

15. See https://tinyurl.com/2svy5tmh.

Judea Pearl [PM18b].

In this section, we discuss how we can use directed graphical model notation to represent **causal models**. We discuss causality in greater detail in Chapter 36, but we introduce some basic ideas and notation here, since it is foundational material that we will need in other parts of the book.

The core idea behind causal models is to create a mechanistic model of the world in which we can reason about the effects of local changes. The canonical example is an electronic circuit: we can predict the effects of any action, such as "knocking out" a particular transistor, or changing the resistance level of a wire, by modifying the circuit locally, and then "re-running" it from the same initial conditions.

We can generalize this idea to create a **structural causal models** or **SCM** [PGJ16], also called **functional causal model** [Sch19]. An SCM is a triple $\mathcal{M} = (\mathcal{U}, \mathcal{V}, \mathcal{F})$, where $\mathcal{U} = \{U_i : i = 1 : N\}$ is a set of unexplained or **exogenous** "noise" variables, which are passed as input to the model, $\mathcal{V} = \{V_i : i = 1 : N\}$ is a set of **endogeneous** variables that are part of the model itself, and $\mathcal{F} = \{f_i : i = 1 : N\}$ is a set of deterministic functions of the form $V_i = f_i(V_{\text{pa}_i}, U_i)$, where pa_i are the parents of variable i, and $U_i \in \mathcal{U}$ are the external inputs. We assume the equations can be structured in a **recursive** way, so the dependency graph of nodes given their parents is a DAG. Finally, we assume our model is **causally sufficient**, which means that $\mathcal{V}$ and $\mathcal{U}$ are all of the causally relevant factors (although they may not all be observed). This is called the "**causal Markov assumption**".

Of course, a model typically cannot represent all the variables that might influence observations or decisions. After all, models are *abstractions* of reality. The variables that we choose not to model explicitly in a functional way can be lumped into the unmodeled exogenous terms. To represent our ignorance about these terms, we can use a distribution $p(U)$ over their values. By "pushing" this external noise through the deterministic part of the model, we induce a distribution over the endogeneous variables, $p(\mathcal{V})$, as in a probabilistic graphical model. However, SCMs make stronger assumptions than PGMs.

We usually assume $p(U)$ is factorized (i.e., the U_i are independent); this is called a **Markovian SCM**. If the exogeneous noise terms are not independent, it would break the assumption that outcomes can be determined locally using deterministic functions. If there are believed to be dependencies between some of the U_i, we can add extra hidden parents to represent this; this is often depicted as a bidirected or undirected edge connecting the U_i, and is known as a **semi-Markovian SCM**.

4.7.1 Example: causal impact of education on wealth

We now give a simple example of an SCM, based on [PM18b, p276]. Suppose we are interested in the causal effect of education on wealth. Let X represent the level of education of a person (on some numeric scale, say 0 = high school, 1 = college, 2 = graduate school), and Y represent their wealth (at some moment in time). In some cases we might expect that increasing X would increase Y (although it of course depends on the nature of the degree, the nature of the job, etc). Thus we add an edge from X to Y. However, getting more education can cost a lot of money (in certain countries), which is a potentially confounding factor on wealth. Let Z be the debt incurred by a person based on their education. We add an edge from X to Z to reflect the fact that larger X means larger Z (in general), and we add an edge from Z to Y to reflect that larger Z means lower Y (in general).

We can represent our structural assumptions graphically as shown in Figure 4.53b(a). The

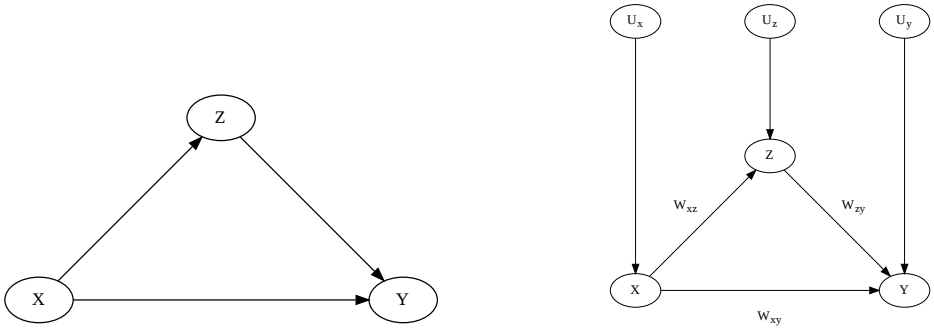


Figure 4.53: (a) PGM for modeling relationship between salary, education and debt. (b) Corresponding SCM.

corresponding SCM has the form:

$$X = f_X(U_x) \tag{4.183}$$
$$Z = f_Z(X, U_z) \tag{4.184}$$
$$Y = f_Y(X, Z, U_y) \tag{4.185}$$

for some set of functions f_x, f_y, f_z, and some prior distribution $p(U_x, U_y, U_z)$. We can also explicitly represent the exogeneous noise terms as shown in Figure 4.53b(b); this makes clear our assumption that the noise terms are a priori independent. (We return to this point later.)

4.7.2 Structural equation models

A **structural equation model** [Bol89; BP13], also known as a **path diagram**, is a special case of a structural causal model in which all the functional relationships are linear, and the prior on the noise terms is Gaussian. SEMs are widely used in economics and social science, due to the fact that they have a causal interpretation, yet they are computationally tractable.

For example, let us make an SEM version of our education example. We have

$$X = U_x \tag{4.186}$$
$$Z = c_z + w_{xz}X + U_z \tag{4.187}$$
$$Y = c_y + w_{xy}X + w_{zy}Z + U_y \tag{4.188}$$

If we assume $p(U_x) = \mathcal{N}(U_x|0, \sigma_x^2)$, $p(U_z) = \mathcal{N}(U_x|0, \sigma_z^2)$, and $p(U_y) = \mathcal{N}(U_x|0, \sigma_y^2)$, then the model can be converted to the following Gaussian DGM:

$$p(X) = \mathcal{N}(X|\mu_x, \sigma_x^2) \tag{4.189}$$
$$p(Z|X) = \mathcal{N}(Z|c_z + w_{xz}X, \sigma_z^2) \tag{4.190}$$
$$p(Y|X, Z) = \mathcal{N}(Y|c_y + w_{xy}X + w_{zy}Z, \sigma_y^2) \tag{4.191}$$

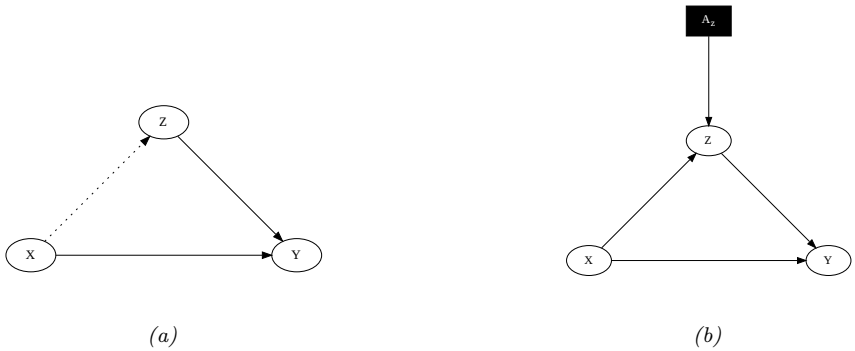


Figure 4.54: An SCM in which we intervene on Z. (a) Hard intervention, in which we clamp Z and thus cut its incoming edges (shown as dotted). (b) Soft intervention, in which we change Z's mechanism. The square node is an "action" node, using the influence diagram notation from Section 34.2.

We can relax the linearity assumption, to allow arbitrarily flexible functions, and relax the Gaussian assumption, to allow any noise distribution. The resulting "nonparametric SEMs" are equivalent to structural causal models. (For a more detailed comparison between SEMs and SCMs, see [Pea12; BP13; Shi00].)

4.7.3 Do operator and augmented DAGs

One of the main advantages of SCMs is that they let us predict the effect of **interventions**, which are actions that change one or more local mechanisms. A simple intervention is to force a variable to have a given value, e.g., we can force a gene to be "on" or "off". This is called a **perfect intervention** and is written as $\text{do}(X_i = x_i)$, where we have introduced new notation for the **"do" operator** (as in the verb "to do"). This notation means we actively clamp variable X_i to value x_i (as opposed to just observing that it has this value). Since the value of X_i is now independent of its usual parents, we should "cut" the incoming edges to node X_i in the graph. This is called the "**graph surgery**" operation.

In Figure 4.54a we illustrate this for our education SCM, where we force Z to have a given value. For example, we may set $Z = 0$, by paying off everyone's student debt. Note that $p(X|\text{do}(Z = z)) \neq p(X|Z = z)$, since the intervention changes the model. For example, if we see someone with a debt of 0, we may infer that they probably did not get higher education, i.e., $p(X \geq 1|Z = 0)$ is small; but if we pay off everyone's college loans, then observing someone with no debt in this modified world should not change our beliefs about whether they got higher education, i.e., $p(X \geq 1|\text{do}(Z = 0)) = p(X \geq 1)$.

In more realistic scenarios, we may not be able to set a variable to a specific value, but we may be able to change it from its current value in some way. For example, we may be able to reduce everyone's debt by some fixed amount, say $\Delta = -10,000$. Thus we replace $Z = f_Z(X, U_z)$ with $Z = f'_z(Z, U_z)$, where $f'_z(Z, U_z) = f_z(Z, U_z) + \Delta$. This is called an **additive intervention**.

To model this kind of scenario, we can add create an **augmented DAG**, in which every variable is augmented with an additional parent node, representing whether or not the variable's mechanism is

Level	Activity	Questions	Examples
1:Association. $p(Y\|a)$	Seeing	How would seeing A change my belief in Y?	Someone took aspirin, how likely is it their headache will be cured?
2:Intervention. $p(Y\|\text{do}(a))$	Doing	What if I do A?	If I take aspirin, will my headache be cured?
3:Counterfactuals. $p(Y^a\|\text{do}(a'), y')$	Imagining	Was it A that caused Y?	Would my headache be cured had I not taken aspirin?

Table 4.6: Pearl's causal hierarchy. Adapted from Table 1 of [Pea19].

changed in some way [Daw02; Daw15; CPD17]. These extra variables are represented by square nodes, and correspond to decision variables or actions, as in the influence diagram formalism (Section 34.2). The same formalism is used in MDPs for reinforcement learning (see Section 34.5).

We give an example of this in Figure 4.54b, where we add the $A_z \in \{0, 1\}$ node to specify whether we use the debt reduction policy or not. The modified mechanism for Z becomes

$$Z = f'_Z(X, U_x, A_z) = \begin{cases} f_Z(X, U_x) & \text{if } A_z = 0 \\ f_Z(X, U_x) + \Delta & \text{if } A_z = 1 \end{cases} \tag{4.192}$$

With this new definition, conditioning on the effects of an action can be performed using standard probabilistic inference. That is, $p(Q|\text{do}(A_z = a), E = e) = p(Q|A_z = a, E = e)$, where Q is the query (e.g., the event $X \geq 1$) and E are the (possibly empty) evidence variables. This is because the A_z node has no parents, so it has no incoming edges to cut when we clamp it.

Although the augmented DAG allows us to use standard notation (no explicit do operators) and inference machinery, the use of "surgical" interventions, which delete incoming edges to a node that is set to a value, results in a simpler graph, which can simplify many calculations, particularly in the non-parametric setting (see [Pea09b, p361] for a discussion). It is therefore a useful abstraction, even if it is less general than the augmented DAG approach.

4.7.4 Counterfactuals

So far we have been focused on predicting the **effects of causes**, so we can choose the optimal action (e.g., if I have a headache, I have to decide should I take an aspirin or not). This can be tackled using standard techniques from Bayesian decision theory, as we have seen (see [Daw00; Daw15; LR19; Roh21; DM22] for more details).

Now suppose we are interested in the **causes of effects**. For example, suppose I took the aspirin and my headache did go away. I might be interested in the **counterfactual question** "if I had not taken the aspirin, would my headache have gone away anyway?". This kind of reasoning is crucial for legal reasoning (see e.g., [DMM17]), as well as for tasks like explainability and fairness.

Counterfactual reasoning requires strictly more assumptions than reasoning about interventions (see e.g., [DM22]). Indeed, Judea Pearl has proposed what he calls the **causal hierarchy** [Pea09b; PGJ16; PM18b], which has three levels of analysis, each more powerful than the last, but each making stronger assumptions. See Table 4.6 for a summary.

In counterfactual reasoning, we want to answer questions of the type $p(Y^{a'}|\text{do}(a), y)$, which is read as: "what is the probability distribution over outcomes Y *if I were to do* a', given that I have already

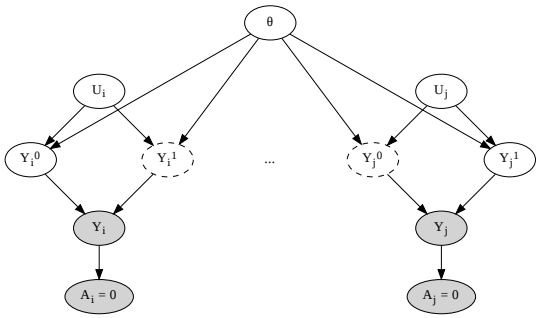


Figure 4.55: Illustration of the potential outcomes framework as a SCM. The nodes with dashed edges are unobserved. In this example, for unit 1, we select action $A_1 = 0$ and observe $Y_1 = Y_1^0 = y_1$, whereas for unit 2, we select action $A_2 = 1$ and observe $Y_2 = Y_2^1 = y_2$.

done a and observed outcome y". (We can also condition on any other evidencee that was observed, such as covariates $\boldsymbol{x}$.) The quantity $Y^{a'}$ is often called a **potential outcome** [Rub74], since it is the outcome that would occur in a hypothetical world in which you did a' instead of a. (Note that $p(Y^{a'} = y)$ is equivalent to $p(Y = y|\text{do}(a'))$, and is an interventional prediction, not a counterfactual one.)

The assumptions behind the potential outcomes framework can be clearly expressed using a structural causal model. We illustrate this in Figure 4.55 for a simple case where there are two possible actions. We see that we have a set of "**units**", such as individual patients, indexed by subscripts. Each unit is associated with a hidden exogeneous random noise source, U_i, that captures everything that is unique about that unit. This noise gets deterministically mapped to two potential outcomes, Y_i^0 and Y_i^1, depending on which action is taken. For any given unit, we only get to observe one of the outcomes, namely the one corresponding to the action that was actually chosen. In the figure, for unit 1, we chose action $A_1 = 0$, so we get to see $Y_1^0 = y_1$, whereas for unit 2, we chose action $A_2 = 1$, so we get to see $Y_2^1 = y_2$. The fact that we cannot simultaneously see both outcomes for the same unit is called the "**fundamental problem of causal inference**" [Hol86].

We will assume the noise sources are independent, which is known as the "stable unit treatment value assumption" or **SUTVA**. (This would not be true if the treatment on person j could somehow affect the outcome of person i, e.g., due to spreading disease or information between i and j.) We also assume that the determinsistic mechanisms that map noise to outcomes are the same across all units (represented by the shared parameter vector $\boldsymbol{\theta}$ in Figure 4.55). We need to make one final assumption, namely that the exogeneous noise is not affected by our actions. (This is a formalization of the assumption known as "all else being equal", or (in legal terms) "**ceteris paribus**".)

With the above assumptions, we can predict what the outcome *for an individual unit* would have been in the alternative universe where we picked the other action. The procedure is as follows. First we perform **abduction** using SCM G, to infer $p(U_i|A_i = a, Y_i = y_i)$, which is the posterior over the latent factors for unit i given the observed evidence in the actual world. Second we perform **intervention**, in which we modify the causal mechanisms of G by replacing $A_i = a$ with $A_i = a'$ to

get $G_{a'}$. Third we perform **prediction**, in which we propagate the distribution of the latent factors, $p(U_i|A_i = a, Y_i = y_i)$, through the modified SCM $G_{a'}$ to get $p(Y_i^{a'}|A_i = a, Y_i = y_i)$.

In Figure 4.55, we see that we have two copies of every possible outcome variable, to represent the set of possible worlds. Of course, we only get to see one such world, based on the actions that we actually took. More generally, a model in which we "clone" all the deterministic variables, with the noise being held constant between the two branches of the graph for the same unit, is called a **twin network** [Pea09b]. We will see a more practical example in Section 29.12.7, where we discuss assessing the counterfactual causal impact of an intervention in a time series. (See also [RR11; RR13], who propose a related formalism known as **single world intervention graph** or **SWIG**.)

We see from the above that the potential outcomes framework is mathematically equivalent to structural causal models, but does not use graphical model notation. This has led to heated debate between the founders of the two schools of thought.[16]. The SCM approach is more popular in computer science (see e.g., [PJS17; Sch19; Sch+21b]), and the PO approach is more popular in economics (see e.g. [AP09; Imb19]). Modern textbooks on causality usually use both formalisms (see e.g., [HR20a; Nea20]).

16. The potential outcomes framework is based on the work of Donald Rubin, and others, and is therefore sometimes called the **Rubin Causal Model** (see e.g., https://en.wikipedia.org/wiki/Rubin_causal_model). The structural causal models framework is based on the work of Judea Pearl and others. See e.g., http://causality.cs.ucla.edu/blog/index.php/2012/12/03/judea-pearl-on-potential-outcomes/ for a discussion of the two.

5 Information theory

Machine learning is fundamentally about **information processing**. But what do we mean by "information"? We discuss this in Section 5.1–Section 5.3. We then go on to briefly discuss two main applications of information theory. The first application is **data compression** or **source coding**, which is the problem of removing redundancy from data so it can be represented more compactly, either in a lossless way (e.g., ZIP files) or a lossy way (e.g., MP3 files). See Section 5.4 for details. The second application is **error correction** or **channel coding**, which means encoding data in such a way that it is robust to errors when sent over a noisy channel, such as a telephone line or a satellite link. See Section 5.5 for details.

It turns out that methods for data compression and error correction both rely on having an accurate probabilistic model of the data. For compression, a probabilistic model is needed so the sender can assign shorter **codewords** to data vectors which occur most often, and hence save space. For error correction, a probabilistic model is needed so the receiver can infer the most likely source message by combining the received noisy message with a prior over possible messages.

It is clear that probabilistic machine learning is useful for information theory. However, information theory is also useful for machine learning. Indeed, we have seen that Bayesian machine learning is about representing and reducing our uncertainty, and so is fundamentally about information. In Section 5.6.2, we explore this direction in more detail, where we discuss the information bottleneck.

For more information on information theory, see e.g., [Mac03; CT06].

5.1 KL divergence

This section is written with Alex Alemi.

To discuss information theory, we need some way to measure or quantify information itself. Let's say we start with some distribution describing our degrees of belief about a random variable, call it $q(x)$. We then want to update our degrees of belief to some new distribution $p(x)$, perhaps because we've taken some new measurements or merely thought about the problem a bit longer. What we seek is a mathematical way to quantify the magnitude of this update, which we'll denote $I[p\|q]$. What sort of criteria would be reasonable for such a measure? We discuss this issue below, and then define a quantity that satisfies these criteria.

5.1.1 Desiderata

For simplicity, imagine we are describing a distribution over N possible events. In this case, the probability distribution $q(\boldsymbol{x})$ consists of N non-negative real numbers that add up to 1. To be even more concrete, imagine we are describing the random variable representing the suit of the next card we'll draw from a deck: $S \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$. Imagine we initially believe the distributions over suits to be uniform: $q = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$. If our friend told us they removed all of the red cards we could update to: $q' = [\frac{1}{2}, \frac{1}{2}, 0, 0]$. Alternatively, we might believe some diamonds changed into clubs and want to update to $q'' = [\frac{3}{8}, \frac{2}{8}, \frac{2}{8}, \frac{1}{8}]$. Is there a good way to quantify *how much* we've updated our beliefs? Which is a larger update: $q \to q'$ or $q \to q''$?

It seems desireable that any useful such measure would satisfy the following properties:

1. *continuous* in its arguments: If we slightly perturb either our starting or ending distribution, it should similarly have a small effect on the magnitude of the update. For example: $I[p\|\frac{1}{4} + \epsilon, \frac{1}{4}, \frac{1}{4}, \frac{1}{4} - \epsilon]$ should be close to $I[p\|q]$ for small ϵ, where $q = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$.

2. *non-negative*: $I[p\|q] \geq 0$ for all $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$. The magnitude of our updates are non-negative.

3. *permutation invariant*: The magnitude of the update should not depend on the order we choose for the elements of $\boldsymbol{x}$. For example, it shouldn't matter if I list my probabilities for the suits of cards in the order $\clubsuit, \spadesuit, \heartsuit, \diamondsuit$ or $\clubsuit, \diamondsuit, \heartsuit, \spadesuit$, if I keep the order consistent across all of the distributions, I should get the same answer. For example: $I[a, b, c, d\|e, f, g, h] = I[a, d, c, b\|e, h, g, f]$.

4. *monotonic* for uniform distributions: While it's hard to say how large the updates in our beliefs are in general, there are some special cases for which we have a strong intuition. If our beliefs update from a uniform distribution on N elements to one that is uniform in N' elements, the information gain should be an increasing function of N and a decreasing function of N'. For instance changing from a uniform distribution on all four suits $[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$ (so $N = 4$) to only one suit, such as all clubs, $[1, 0, 0, 0]$ where $N' = 1$, is a larger update than if I only updated to the card being black, $[\frac{1}{2}, \frac{1}{2}, 0, 0]$ where $N' = 2$.

5. satisfy a natural *chain rule*: So far we've been describing our beliefs in what will happen on the next card draw as a single random variable representing the suit of the next card ($S \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$). We could equivalently describe the same physical process in two steps. First we consider the random variable representing the color of the card ($C \in \{\blacksquare, \square\}$), which could be either black ($\blacksquare = \{\clubsuit, \spadesuit\}$) or red ($\square = \{\heartsuit, \diamondsuit\}$). Then, if we draw a red card we describe our belief that it is $\heartsuit$ versus $\diamondsuit$. If it was instead black we would assign beliefs to it being $\clubsuit$ versus $\spadesuit$. We can convert any distribution over the four suits into this conditional factorization, for example:

$$p(S) = \left[\frac{3}{8}, \frac{2}{8}, \frac{2}{8}, \frac{1}{8}\right] \tag{5.1}$$

becomes

$$p(C) = \left[\frac{5}{8}, \frac{3}{8}\right] \quad p(\{\clubsuit, \spadesuit\}|C = \blacksquare) = \left[\frac{3}{5}, \frac{2}{5}\right] \quad p(\{\heartsuit, \diamondsuit\}|C = \square) = \left[\frac{2}{3}, \frac{1}{3}\right]. \tag{5.2}$$

In the same way we could decompose our uniform distribution q. Obviously, for our measure of information to be of use the magnitude of the update needs to be the same regardless of how we

choose to describe what is ultimately the same physical process. What we need is some way to relate what would be four different invocations of our information function:

$$I_S \equiv I\left[p(S) \| q(S)\right] \tag{5.3}$$
$$I_C \equiv I\left[p(C) \| q(C)\right] \tag{5.4}$$
$$I_{\blacksquare} \equiv I\left[p(\{\clubsuit, \spadesuit\}|C = \blacksquare) \| q(\{\clubsuit, \spadesuit\}|C = \blacksquare)\right] \tag{5.5}$$
$$I_{\square} \equiv I\left[p(\{\heartsuit, \diamondsuit\}|C = \square) \| q(\{\heartsuit, \diamondsuit\}|C = \square)\right]. \tag{5.6}$$

Clearly I_S should be some function of $\{I_C, I_{\blacksquare}, I_{\square}\}$. Our last desiderata is that the way we measure the magnitude of our updates will have I_S be a linear combination of $I_C, I_{\blacksquare}, I_{\square}$. In particular, we will require that they combine as a weighted linear combinations, with weights set by the probability that we would find ourselves in that branch according to the distribution p:

$$I_S = I_C + p(C = \blacksquare) I_{\blacksquare} + p(C = \square) I_{\square} = I_C + \frac{5}{8} I_{\blacksquare} + \frac{3}{5} I_{\square} \tag{5.7}$$

Stating this requirement more generally: If we partition $\boldsymbol{x}$ into two pieces $[\boldsymbol{x}_L, \boldsymbol{x}_R]$, so that we can write $p(\boldsymbol{x}) = p(\boldsymbol{x}_L)p(\boldsymbol{x}_R|\boldsymbol{x}_L)$ and similarly for q, the magnitude of the update should be

$$I[p(\boldsymbol{x}) \| q(\boldsymbol{x})] = I[p(\boldsymbol{x}_L) \| q(\boldsymbol{x}_L)] + \mathbb{E}_{p(\boldsymbol{x}_L)}\left[I[p(\boldsymbol{x}_R|\boldsymbol{x}_L) \| q(\boldsymbol{x}_R|\boldsymbol{x}_L)]\right]. \tag{5.8}$$

Notice that this requirement *breaks the symmetry between our two distributions*: The right hand side asks us to take the expected conditional information gain with respect to the marginal, but we need to decide which of two marginals to take the expectation with respect to.

5.1.2 The KL divergence uniquely satisfies the desiderata

We will now define a quantity that is the only measure (up to a multiplicative constant) that satisfies the above desiderata. The **Kullback-Leibler divergence** or **KL divergence**, also known as the **information gain** or **relative entropy**, is defined as follows:

$$D_{\mathbb{KL}}\left(p \parallel q\right) \triangleq \sum_{k=1}^{K} p_k \log \frac{p_k}{q_k}. \tag{5.9}$$

This naturally extends to continuous distributions:

$$D_{\mathbb{KL}}\left(p \parallel q\right) \triangleq \int dx\, p(x) \log \frac{p(x)}{q(x)}. \tag{5.10}$$

Next we will verify that this definition satisfies all of our desiderata. (The proof that it is the unique measure which captures these properties can be found in e.g., [Hob69; Rén61].)

5.1.2.1 Continuity of KL

One of our desiderata was that our measure of information gain should be continuous. The KL divergence is manifestly continuous in its arguments except potentially when p_k or q_k is zero. In the first case, notice that the limit as $p \to 0$ is well behaved:

$$\lim_{p \to 0} p \log \frac{p}{q} = 0. \tag{5.11}$$

Taking this as the definition of the value of the integrand when $p = 0$ will make it continuous there. Notice that we do have a problem however if $q = 0$ in some place that $p \neq 0$. Our information gain requires that our original distribution of beliefs q has some support everywhere the updated distribution does. Intuitively it would require an infinite amount of information for us to update our beliefs in some outcome to change from being exactly 0 to some positive value.

5.1.2.2 Non-negativity of KL divergence

In this section, we prove that the KL divergence as defined is always non-negative. We will make use of **Jensen's inequality**, which states that for any convex function f, we have that

$$f\left(\sum_{i=1}^{n} \lambda_i \boldsymbol{x}_i\right) \leq \sum_{i=1}^{n} \lambda_i f(\boldsymbol{x}_i) \tag{5.12}$$

where $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i = 1$. This can be proved by induction, where the base case with $n = 2$ follows by definition of convexity.

Theorem 5.1.1. *(Information inequality)* $D_{\mathbb{KL}}(p \parallel q) \geq 0$ *with equality iff* $p = q$.

Proof. We now prove the theorem, following [CT06, p28]. As we noted in the previous section, the KL divergence requires special consideration when $p(x)$ or $q(x) = 0$, the same is true here. Let $A = \{x : p(x) > 0\}$ be the support of $p(x)$. Using the convexity of the log function and Jensen's inequality, we have that

$$-D_{\mathbb{KL}}(p \parallel q) = -\sum_{x \in A} p(x) \log \frac{p(x)}{q(x)} = \sum_{x \in A} p(x) \log \frac{q(x)}{p(x)} \tag{5.13}$$

$$\leq \log \sum_{x \in A} p(x) \frac{q(x)}{p(x)} = \log \sum_{x \in A} q(x) \tag{5.14}$$

$$\leq \log \sum_{x \in \mathcal{X}} q(x) = \log 1 = 0 \tag{5.15}$$

Since $\log(x)$ is a strictly concave function ($-\log(x)$ is convex), we have equality in Equation (5.14) iff $p(x) = cq(x)$ for some c that tracks the fraction of the whole space $\mathcal{X}$ contained in A. We have equality in Equation (5.15) iff $\sum_{x \in A} q(x) = \sum_{x \in \mathcal{X}} q(x) = 1$, which implies $c = 1$. Hence $D_{\mathbb{KL}}(p \parallel q) = 0$ iff $p(x) = q(x)$ for all x. □

The non-negativity of KL divergence often feels as though it's one of the most useful results in Information Theory. It is a good result to keep in your back pocket. Anytime you can rearrange an expression in terms of KL divergence terms, since those are guaranteed to be non-negative, dropping them immediately generates a bound.

5.1.2.3 KL divergence is invariant to reparameterizations

We wanted our measure of information to be invariant to permutations of the labels. The discrete form is manifestly permutation invariant as summations are. The KL divergence actually satisfies a

much stronger property of reparameterization invariance. Namely, we can transform our random variable through an arbitrary invertible map and it won't change the value of the KL divergence.

If we transform our random variable from x to some $y = f(x)$ we know that $p(x)\,dx = p(y)\,dy$ and $q(x)\,dx = q(y)\,dy$. Hence the KL divergence remains the same for both random variables:

$$D_{\mathbb{KL}}\left(p(x) \parallel q(x)\right) = \int dx\, p(x) \log \frac{p(x)}{q(x)} = \int dy\, p(y) \log \left(\frac{p(y)\left|\frac{dy}{dx}\right|}{q(y)\left|\frac{dy}{dx}\right|} \right) = D_{\mathbb{KL}}\left(p(y) \parallel q(y)\right). \quad (5.16)$$

Because of this reparameterization invariance we can rest assured that when we measure the KL divergence between two distributions we are measuring something about the distributions and not the way we choose to represent the space in which they are defined. We are therefore free to transform our data into a convenient basis of our choosing, such as a Fourier bases for images, without affecting the result.

5.1.2.4 Montonicity for uniform distributions

Consider updating a probability distribution from a uniform distribution on N elements to a uniform distribution on N' elements. The KL divergence is:

$$D_{\mathbb{KL}}\left(p \parallel q\right) = \sum_k \frac{1}{N'} \log \frac{\frac{1}{N'}}{\frac{1}{N}} = \log \frac{N}{N'}, \quad (5.17)$$

or the log of the ratio of the elements before and after the update. This satisfies our monotonocity requirement.

We can interpret this result as follows: Consider finding an element of a sorted array by means of bisection. A well designed yes/no question can cut the search space in half. Measured in bits, the KL divergence tells us how many well designed yes/no questions are required on average to move from q to p.

5.1.2.5 Chain rule for KL divergence

Here we show that the KL divergence satisfies a natural chain rule:

$$D_{\mathbb{KL}}\left(p(x,y) \parallel q(x,y)\right) = \int dx\, dy\, p(x,y) \log \frac{p(x,y)}{q(x,y)} \quad (5.18)$$

$$= \int dx\, dy\, p(x,y) \left[\log \frac{p(x)}{q(x)} + \log \frac{p(y|x)}{q(y|x)} \right] \quad (5.19)$$

$$= D_{\mathbb{KL}}\left(p(x) \parallel q(x)\right) + \mathbb{E}_{p(x)}\left[D_{\mathbb{KL}}\left(p(y|x) \parallel q(y|x)\right) \right]. \quad (5.20)$$

We can rest assured that we can decompose our distributions into their conditionals and the KL divergences will just add.

As a notational convenience, the **conditional KL divergence** is defined to be the expected value of the KL divergence between two conditional distributions:

$$D_{\mathbb{KL}}\left(p(y|x) \parallel q(y|x)\right) \triangleq \int dx\, p(x) \int dy\, p(y|x) \log \frac{p(y|x)}{q(y|x)}. \quad (5.21)$$

This allows us to drop many expectation symbols.

5.1.3 Thinking about KL

In this section, we discuss some qualitative properties of the KL divergence.

5.1.3.1 Units of KL

Above we said that the desiderata we listed determined the KL divergence up to a multiplicative constant. Because the KL divergence is logarithmic, and logarithms in different bases are the same up to a multiplicative constant, our choice of the base of the logarithm when we compute the KL divergence is a choice akin to choosing which units to measure the information in.

If the KL divergence is measured with the base-2 logarithm, it is said to have units of **bits**, short for "binary digits". If measured using the natural logarithm as we normally do for mathematical convenience, it is said to be measured in **nats** for "natural units".

To convert between the systems, we use $\log_2 y = \frac{\log y}{\log 2}$. Hence

$$1 \text{ bit} = \log 2 \text{ nats} \sim 0.693 \text{ nats} \tag{5.22}$$

$$1 \text{ nat} = \frac{1}{\log 2} \text{ bits} \sim 1.44 \text{ bits}. \tag{5.23}$$

5.1.3.2 Asymmetry of the KL divergence

The KL divergence is *not* symmetric in its two arguments. While many find this asymmetry confusing at first, we can see that the asymmetry stems from our requirement that we have a natural chain rule. When we decompose the distribution into its conditional, we need to take an expectation with respect to the variables being conditioned on. In the KL divergence we take this expectation with respect to the first argument $p(x)$. This breaks the symmetry between the two distributions.

At a more intuitive level, we can see that the information required to move from q to p is in general different than the information required to move from p to q. For example, consider the KL divergence between two Bernoulli distributions, the first with the probability of success given by 0.443 and the second with 0.975:

$$\mathrm{D}_{\mathbb{KL}} = 0.975 \log \frac{0.975}{0.443} + 0.025 \log \frac{0.025}{0.557} = 0.692 \text{ nats} \sim 1.0 \text{ bits}. \tag{5.24}$$

So it takes 1 bit of information to update from a $[0.443, 0.557]$ distribution to a $[0.975, 0.025]$ Bernoulli distribution. What about the reverse?

$$\mathrm{D}_{\mathbb{KL}} = 0.443 \log \frac{0.443}{0.975} + 0.557 \log \frac{0.557}{0.025} = 1.38 \text{ nats} \sim 2.0 \text{ bits}, \tag{5.25}$$

so it takes two bits, or twice as much information to move the other way. Thus we see that starting with a distribution that is nearly even and moving to one that is nearly certain takes about 1 bit of information, or one well designed yes/no question. To instead move us from near certainty in an outcome to something that is akin to the flip of a coin requires more persuasion.

5.1.3.3 Minimizing forwards vs reverse KL

The asymmetry of KL means that finding a p that is close to q by minimizing $D_{\mathbb{KL}}(p \parallel q)$ (also called the **inclusive KL**) gives different behavior than minimizing $D_{\mathbb{KL}}(q \parallel p)$ (also called the **exclusive**

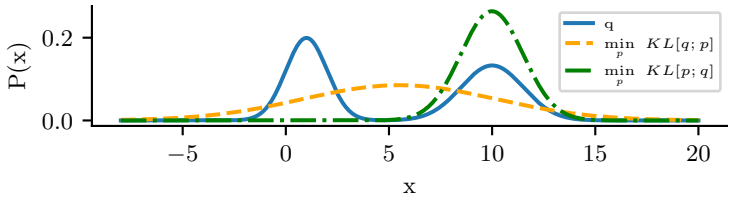
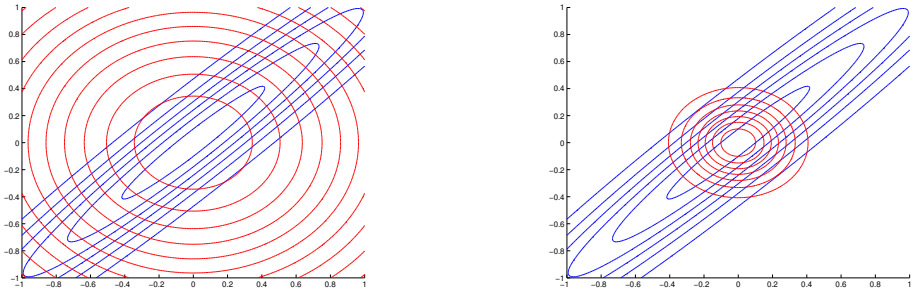


Figure 5.1: Demonstration of the mode-covering or mode-seeking behavior of KL divergence. The original distribution q is bimodal. When we minimize $D_{\mathbb{KL}}(q \parallel p)$, then p covers the modes of q (orange). When we minimize $D_{\mathbb{KL}}(p \parallel q)$, then p ignores some of the modes of q (green) Generated by minimize_kl_divergence.ipynb.



(a) *(b)*

Figure 5.2: Illustrating forwards vs reverse KL on a symmetric Gaussian. The blue curves are the contours of the true distribution p. The red curves are the contours of a factorized approximation q. (a) Minimizing $D_{\mathbb{KL}}(p \parallel q)$. (b) Minimizing $D_{\mathbb{KL}}(q \parallel p)$. Adapted from Figure 10.2 of [Bis06]. Generated by kl_pq_gauss.ipynb.

KL). For example, consider the bimodal distribution q shown in blue in Figure 5.1, which we approximate with a unimodal Gaussian. To prevent $D_{\mathbb{KL}}(q \parallel p)$ from becoming infinite, we must have $p > 0$ whenever $q > 0$ (i.e., p must have support everywhere q does), so p tends to *cover* both modes as it must be nonvanishing everywhere q is; this is called **mode-covering** or **zero-avoiding** behavior (orange curve).

By contrast, to prevent $D_{\mathbb{KL}}(p \parallel q)$ from becoming infinite, we must have $p = 0$ whenever $q = 0$, which creates **mode-seeking** or **zero-forcing** behavior (green curve).

5.1.3.4 Moment projection

Suppose we compute q by minimizing the forwards KL:

$$q = \operatorname*{argmin}_q D_{\mathbb{KL}}(p \parallel q) \tag{5.26}$$

This is called **M-projection**, or **moment projection** since the optimal q matches the moments of p; this is called **moment matching**.

To see why, let us assume that q is an exponential family distribution of the form

$$q(\boldsymbol{x}) = h(\boldsymbol{x}) \exp[\boldsymbol{\eta}^\mathsf{T} \mathcal{T}(\boldsymbol{x}) - \log Z(\boldsymbol{\eta})] \tag{5.27}$$

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

$$\partial_{\eta_i} D_{\mathbb{KL}}\left(p \parallel q\right) = -\partial_{\eta_i} \int_{\boldsymbol{x}} p(\boldsymbol{x}) \log q(\boldsymbol{x}) \tag{5.28}$$

$$= -\partial_{\eta_i} \int_{\boldsymbol{x}} p(\boldsymbol{x}) \log \left(h(\boldsymbol{x}) \exp[\boldsymbol{\eta}^{\mathsf{T}} \mathcal{T}(\boldsymbol{x}) - \log Z(\boldsymbol{\eta})]\right) \tag{5.29}$$

$$= -\partial_{\eta_i} \int_{\boldsymbol{x}} p(\boldsymbol{x}) \left(\boldsymbol{\eta}^{\mathsf{T}} \mathcal{T}(\boldsymbol{x}) - \log Z(\boldsymbol{\eta})\right) \tag{5.30}$$

$$= -\int_{\boldsymbol{x}} p(\boldsymbol{x}) \mathcal{T}_i(\boldsymbol{x}) + \mathbb{E}_{q(\boldsymbol{x})}\left[\mathcal{T}_i(\boldsymbol{x})\right] \tag{5.31}$$

$$= -\mathbb{E}_{p(\boldsymbol{x})}\left[\mathcal{T}_i(\boldsymbol{x})\right] + \mathbb{E}_{q(\boldsymbol{x})}\left[\mathcal{T}_i(\boldsymbol{x})\right] = 0 \tag{5.32}$$

where in the penultimate line we used the fact that the derivative of the log partition function yields the expected sufficient statistics, as shown in Equation (2.199). Hence the expected sufficient statistics (moments of the distribution) must match.

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

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{12}^{\mathsf{T}} & \Sigma_{22} \end{pmatrix} \quad \boldsymbol{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^{\mathsf{T}} & \Lambda_{22} \end{pmatrix} \tag{5.33}$$

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

$$q(\boldsymbol{x}|\boldsymbol{m}, \mathbf{V}) = \mathcal{N}(x_1|m_1, v_1)\mathcal{N}(x_2|m_2, v_2) \tag{5.34}$$

If we perform moment matching, the optimal q must therefore have the following form:

$$q(\boldsymbol{x}) = \mathcal{N}(x_1|\mu_1, \Sigma_{11})\mathcal{N}(x_2|\mu_2, \Sigma_{22}) \tag{5.35}$$

In Figure 5.2(a), we show the resulting distribution. We see that q covers (includes) p, but its support is too broad (under-confidence).

5.1.3.5 Information projection

Now suppose we compute q by minimizing the reverse KL:

$$q = \operatorname*{argmin}_q D_{\mathbb{KL}}\left(q \parallel p\right) \tag{5.36}$$

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

In the case of a 2d Gaussian, one can show [Mac03, p435] that the optimal solution has the form

$$q(\boldsymbol{x}) = \mathcal{N}(x_1|m_1, \Lambda_{11}^{-1})\mathcal{N}(x_2|m_2, \Lambda_{22}^{-1}) \tag{5.37}$$

$$m_1 = \mu_1 - \Lambda_{11}^{-1}\Lambda_{12}(m_2 - \mu_2) \tag{5.38}$$

$$m_2 = \mu_2 - \Lambda_{22}^{-1}\Lambda_{21}(m_1 - \mu_1) \tag{5.39}$$

To solve this fixed point equation, we can set $m_i = \mu_i$, so we have again captured the mean exactly. However, the posterior variance is too narrow, i..e, the approximate posterior is overconfident. This is shown in Figure 5.2(b). (Note, however, that minimizing the reverse KL does not always result in an overly compact approximation, as explained in [Tur+08].)

5.1.3.6 KL as expected weight of evidence

Imagine you have two different hypotheses you wish to select between, which we'll label P and Q. You collect some data D. Bayes' rule tells us how to update our beliefs in the hypotheses being correct:

$$\Pr(P|D) = \frac{\Pr(D|P)}{\Pr(D)} \Pr(P). \tag{5.40}$$

Normally this requires being able to evaluate the marginal likelihood $\Pr(D)$, which is difficult. If we instead consider the ratio of the probabilities for the two hypotheses:

$$\frac{\Pr(P|D)}{\Pr(Q|D)} = \frac{\Pr(D|P)}{\Pr(D|Q)} \frac{\Pr(P)}{\Pr(Q)}, \tag{5.41}$$

the marginal likelihood drops out. Taking the logarithm of both sides, and identifying the probability of the data under the model as the likelihood we find:

$$\log \frac{\Pr(P|D)}{\Pr(Q|D)} = \log \frac{p(D)}{q(D)} + \log \frac{\Pr(P)}{\Pr(Q)}. \tag{5.42}$$

The posterior log probability ratio for one hypothesis over the other is just our prior log probability ratio plus a term that I. J. Good called the **weight of evidence** [Goo85] D for hypothesis P over Q:

$$w[P/Q; D] \triangleq \log \frac{p(D)}{q(D)}. \tag{5.43}$$

With this interpretation, the KL divergence is the expected weight of evidence for P over Q given by each observation, provided P were correct. Thus we see that data will (on average) add rather than subtract evidence towards the correct hypothesis, since KL divergence is always non-negative in expectation (see Section 5.1.2.2).

5.1.4 Properties of KL

Below are some other useful properties of the KL divergence.

5.1.4.1 Compression Lemma

An important general purpose result for the KL divergence is the Compression Lemma:

Theorem 5.1.2. *For any distributions P and Q with a well-defined KL divergence, and for any scalar function ϕ defined on the domain of the distributions we have that:*

$$\mathbb{E}_P[\phi] \leq \log \mathbb{E}_Q\left[e^{\phi}\right] + D_{\mathbb{KL}}(P \parallel Q). \tag{5.44}$$

Proof. We know that the KL divergence between any two distributions is non-negative. Consider a distribution of the form:

$$g(x) = \frac{q(x)}{\mathcal{Z}} e^{\phi(x)}. \tag{5.45}$$

where the *partition function* is given by:

$$\mathcal{Z} = \int dx\, q(x) e^{\phi(x)}. \tag{5.46}$$

Taking the KL divergence between $p(x)$ and $g(x)$ and rearranging gives the bound:

$$D_{\mathbb{KL}}\left(P \parallel G\right) = D_{\mathbb{KL}}\left(P \parallel Q\right) - \mathbb{E}_P\left[\phi(x)\right] + \log(\mathcal{Z}) \geq 0. \tag{5.47}$$

□

One way to view the compression lemma is that it provides what is termed the Donsker-Varadhan variational representation of the KL divergence:

$$D_{\mathbb{KL}}\left(P \parallel Q\right) = \sup_{\phi} \mathbb{E}_P\left[\phi(x)\right] - \log \mathbb{E}_Q\left[e^{\phi(x)}\right]. \tag{5.48}$$

In the space of all possible functions ϕ defined on the same domain as the distributions, assuming all of the values above are finite, the KL divergence is the supremum achieved. For any fixed function $\phi(x)$, the right hand side provides a lower bound on the true KL divergence.

Another use of the compression lemma is that it provides a way to estimate the expectation of some function with respect to an unknown distribution P. In this spirit, the Compression Lemma can be used to power a set of what are known as PAC Bayes bounds of losses with respect to the true distribution in terms of measured losses with respect to a finite training set. See for example Section 17.4.5 or Banerjee [Ban06].

5.1.4.2 Data processing inequality for KL

We now show that any processing we do on samples from two different distributions makes their samples approach one another. This is called the **data processing inequality**, since it shows that we cannot increase the information gain from q to p by processing our data and then measuring it.

Theorem 5.1.3. *Consider two different distributions $p(x)$ and $q(x)$ combined with a probabilistic channel $t(y|x)$. If $p(y)$ is the distribution that results from sending samples from $p(x)$ through the channel $t(y|x)$ and similarly for $q(y)$ we have that:*

$$D_{\mathbb{KL}}\left(p(x) \parallel q(x)\right) \geq D_{\mathbb{KL}}\left(p(y) \parallel q(y)\right) \tag{5.49}$$

Proof. The proof uses Jensen's inequality from Section 5.1.2.2 again. Call $p(x,y) = p(x)t(y|x)$ and $q(x,y) = q(x)t(y|x)$.

$$D_{\mathbb{KL}}\left(p(x) \parallel q(x)\right) = \int dx\, p(x) \log \frac{p(x)}{q(x)} \tag{5.50}$$

$$= \int dx \int dy\, p(x)t(y|x) \log \frac{p(x)t(y|x)}{q(x)t(y|x)} \tag{5.51}$$

$$= \int dx \int dy\, p(x,y) \log \frac{p(x,y)}{q(x,y)} \tag{5.52}$$

$$= -\int dy\, p(y) \int dx\, p(x|y) \log \frac{q(x,y)}{p(x,y)} \tag{5.53}$$

$$\geq -\int dy\, p(y) \log \left(\int dx\, p(x|y) \frac{q(x,y)}{p(x,y)} \right) \tag{5.54}$$

$$= -\int dy\, p(y) \log \left(\frac{q(y)}{p(y)} \int dx\, q(x|y) \right) \tag{5.55}$$

$$= \int dy\, p(y) \log \frac{p(y)}{q(y)} = D_{\mathbb{KL}}\left(p(y) \parallel q(y)\right) \tag{5.56}$$

□

One way to interpret this result is that any processing done to random samples makes it harder to tell two distributions apart.

As a special form of processing, we can simply marginalize out a subset of random variables.

Corollary 5.1.1. *(Monotonicity of KL divergence)*

$$D_{\mathbb{KL}}\left(p(x,y) \parallel q(x,y)\right) \geq D_{\mathbb{KL}}\left(p(x) \parallel q(x)\right) \tag{5.57}$$

Proof. The proof is essentially the same as the one above.

$$D_{\mathbb{KL}}\left(p(x,y) \parallel q(x,y)\right) = \int dx \int dy\, p(x,y) \log \frac{p(x,y)}{q(x,y)} \tag{5.58}$$

$$= -\int dy\, p(y) \int dx\, p(x|y) \log \left(\frac{q(y)}{p(y)} \frac{q(x|y)}{p(x|y)} \right) \tag{5.59}$$

$$\geq -\int dy\, p(y) \log \left(\frac{q(y)}{p(y)} \int dx\, q(x|y) \right) \tag{5.60}$$

$$= \int dy\, p(y) \log \frac{p(y)}{q(y)} = D_{\mathbb{KL}}\left(p(y) \parallel q(y)\right) \tag{5.61}$$

$$\tag{5.62}$$

□

One intuitive interpretation of this result is that if you only partially observe random variables, it is harder to distinguish between two candidate distributions than if you observed all of them.

5.1.5 KL divergence and MLE

Suppose we want to find the distribution q that is as close as possible to p, as measured by KL divergence:

$$q^* = \arg\min_q D_{\mathbb{KL}}\left(p \parallel q\right) = \arg\min_q \int p(x) \log p(x) dx - \int p(x) \log q(x) dx \tag{5.63}$$

Now suppose p is the empirical distribution, which puts a probability atom on the observed training data and zero mass everywhere else:

$$p_{\mathcal{D}}(x) = \frac{1}{N_{\mathcal{D}}} \sum_{n=1}^{N_{\mathcal{D}}} \delta(x - x_n) \tag{5.64}$$

Using the sifting property of delta functions we get

$$D_{\mathbb{KL}}\left(p_{\mathcal{D}} \parallel q\right) = -\int p_{\mathcal{D}}(x) \log q(x) dx + C \tag{5.65}$$

$$= -\int \left[\frac{1}{N_{\mathcal{D}}} \sum_n \delta(x - x_n)\right] \log q(x) dx + C \tag{5.66}$$

$$= -\frac{1}{N_{\mathcal{D}}} \sum_n \log q(x_n) + C \tag{5.67}$$

where $C = \int p_{\mathcal{D}}(x) \log p_{\mathcal{D}}(x)$ is a constant independent of q.

We can rewrite the above as follows

$$D_{\mathbb{KL}}\left(p_{\mathcal{D}} \parallel q\right) = \mathbb{H}_{ce}(p_{\mathcal{D}}, q) - \mathbb{H}(p_{\mathcal{D}}) \tag{5.68}$$

where

$$\mathbb{H}_{ce}(p, q) \triangleq -\sum_k p_k \log q_k \tag{5.69}$$

is known as the **cross entropy**. The quantity $\mathbb{H}_{ce}(p_{\mathcal{D}}, q)$ is the average negative log likelihood of q evaluated on the training set. Thus we see that minimizing KL divergence to the empirical distribution is equivalent to maximizing likelihood.

This perspective points out the flaw with likelihood-based training, namely that it puts too much weight on the training set. In most applications, we do not really believe that the empirical distribution is a good representation of the true distribution, since it just puts "spikes" on a finite set of points, and zero density everywhere else. Even if the dataset is large (say 1M images), the universe from which the data is sampled is usually even larger (e.g., the set of "all natural images" is much larger than 1M). Thus we need to somehow smooth the empirical distribution by sharing probability mass between "similar" inputs.

5.1.6 KL divergence and Bayesian Inference

Bayesian inference itself can be motivated as the solution to a particular minimization problem of KL.

Consider a prior set of beliefs described by a joint distribution $q(\theta, D) = q(\theta)q(D|\theta)$, involving some *prior* $q(\theta)$ and some *likelihood* $q(D|\theta)$. If we happen to observe some particular dataset D_0, how should we update our beliefs? We could search for the joint distribution that is as close as possible to our prior beliefs but that respects the constraint that we now know the value of the data:

$$p(\theta, D) = \text{argmin}\, D_{\mathbb{KL}}\left(p(\theta, D) \parallel q(\theta, D)\right) \text{ such that } p(D) = \delta(D - D_0). \tag{5.70}$$

where $\delta(D - D_0)$ is a degenerate distribution that puts all its mass on the dataset D that is identically equal to D_0. Writing the KL out in its chain rule form:

$$D_{\mathbb{KL}}\left(p(\theta, D) \parallel q(\theta, D)\right) = D_{\mathbb{KL}}\left(p(D) \parallel q(D)\right) + D_{\mathbb{KL}}\left(p(\theta|D) \parallel q(\theta|D)\right), \tag{5.71}$$

makes clear that the solution is given by the joint distribution:

$$p(\theta, D) = p(D)p(\theta|D) = \delta(D - D_0)q(\theta|D). \tag{5.72}$$

Our updated beliefs have a marginal over the θ

$$p(\theta) = \int dD\, p(\theta, D) = \int dD\, \delta(D - D_0)q(\theta|D) = q(\theta|D = D_0), \tag{5.73}$$

which is just the usual Bayesian posterior from our prior beliefs evaluated at the data we observed.

By contrast, the usual statement of Bayes' rule is just a trivial observation about the chain rule of probabilities:

$$q(\theta, D) = q(D)q(\theta|D) = q(\theta)q(D|\theta) \implies q(\theta|D) = \frac{q(D|\theta)}{q(D)}q(\theta). \tag{5.74}$$

Notice that this relates the conditional distribution $q(\theta|D)$ in terms of $q(D|\theta)$, $q(\theta)$ and $q(D)$, but that these are all different ways to write the same distribution. Bayes rule does not tell us how we ought to *update* our beliefs in light of evidence, for that we need some other principle [Cat+11].

One of the nice things about this interpretation of Bayesian inference is that it naturally generalizes to other forms of constraints rather than assuming we have observed the data exactly.

If there was some additional measurement error that was well understood, we ought to instead of pegging out updated beliefs to be a delta function on the observed data, simply peg it to be the well understood distribution $p(D)$. For example, we might not know the precise value the data takes, but believe after measuring things that it is a Gaussian distribution with a certain mean and standard deviation.

Because of the chain rule of KL, this has no effect on our updated conditional distribution over parameters, which remains the Bayesian posterior: $p(\theta|D) = q(\theta|D)$. However, this does change our marginal beliefs about the parameters, which are now:

$$p(\theta) = \int dD\, p(D)q(\theta|D). \tag{5.75}$$

This generalization of Bayes' rule is sometimes called **Jeffrey's conditionalization rule** [Cat08].

5.1.7 KL divergence and Exponential Families

The KL divergence between two exponential family distributions from the same family has a nice closed form, as we explain below.

Consider $p(\boldsymbol{x})$ with natural parameter $\boldsymbol{\eta}$, base measure $h(\boldsymbol{x})$ and sufficient statistics $\mathcal{T}(\boldsymbol{x})$:

$$p(\boldsymbol{x}) = h(\boldsymbol{x}) \exp[\boldsymbol{\eta}^\mathsf{T} \mathcal{T}(\boldsymbol{x}) - A(\boldsymbol{\eta})] \tag{5.76}$$

where

$$A(\boldsymbol{\eta}) = \log \int h(\boldsymbol{x}) \exp(\boldsymbol{\eta}^\mathsf{T} \mathcal{T}(\boldsymbol{x})) d\boldsymbol{x} \tag{5.77}$$

is the *log partition function*, a convex function of $\boldsymbol{\eta}$.

The KL divergence between two exponential family distributions from the same family is as follows:

$$D_{\mathbb{KL}}\left(p(\boldsymbol{x}|\boldsymbol{\eta}_1) \parallel p(\boldsymbol{x}|\boldsymbol{\eta}_2)\right) = \mathbb{E}_{\boldsymbol{\eta}_1}\left[(\boldsymbol{\eta}_1 - \boldsymbol{\eta}_2)^\mathsf{T} \mathcal{T}(\boldsymbol{x}) - A(\boldsymbol{\eta}_1) + A(\boldsymbol{\eta}_2)\right] \tag{5.78}$$

$$= (\boldsymbol{\eta}_1 - \boldsymbol{\eta}_2)^\mathsf{T} \boldsymbol{\mu}_1 - A(\boldsymbol{\eta}_1) + A(\boldsymbol{\eta}_2) \tag{5.79}$$

where $\boldsymbol{\mu}_j \triangleq \mathbb{E}_{\boldsymbol{\eta}_j}\left[\mathcal{T}(\boldsymbol{x})\right]$.

5.1.7.1 Example: KL divergence between two Gaussians

An important example is the KL divergence between two multivariate Gaussian distributions, which is given by

$$D_{\mathbb{KL}}\left(\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \parallel \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)\right)$$

$$= \frac{1}{2}\left[\mathrm{tr}(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1) + (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^\mathsf{T} \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) - D + \log\left(\frac{\det(\boldsymbol{\Sigma}_2)}{\det(\boldsymbol{\Sigma}_1)}\right)\right] \tag{5.80}$$

In the scalar case, this becomes

$$D_{\mathbb{KL}}\left(\mathcal{N}(x|\mu_1, \sigma_1) \parallel \mathcal{N}(x|\mu_2, \sigma_2)\right) = \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \tag{5.81}$$

5.1.8 Bregman divergence

Let $f : \Omega \to \mathbb{R}$ be a continuously differentiable, strictly convex function defined on a closed convex set Ω. We define the **Bregman divergence** associated with f as follows [Bre67]:

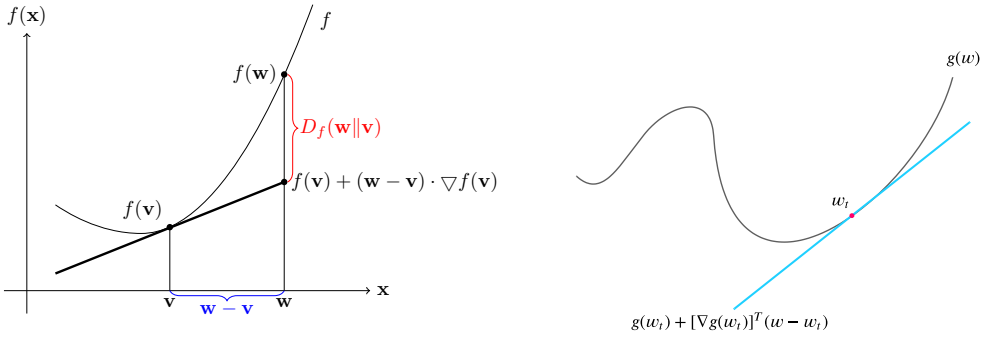
$$B_f(\boldsymbol{w}||\boldsymbol{v}) = f(\boldsymbol{w}) - f(\boldsymbol{v}) - (\boldsymbol{w} - \boldsymbol{v})^\mathsf{T} \nabla f(\boldsymbol{v}) \tag{5.82}$$

To understand this, let

$$\hat{f}_{\boldsymbol{v}}(\boldsymbol{w}) = f(\boldsymbol{v}) + (\boldsymbol{w} - \boldsymbol{v})^\mathsf{T} \nabla f(\boldsymbol{v}) \tag{5.83}$$

be a first order Taylor series approximation to f centered at $\boldsymbol{v}$. Then the Bregman divergence is the difference from this linear approximation:

$$B_f(\boldsymbol{w}||\boldsymbol{v}) = f(\boldsymbol{w}) - \hat{f}_{\boldsymbol{v}}(\boldsymbol{w}) \tag{5.84}$$



(a) (b)

Figure 5.3: (a) Illustration of Bregman divergence. (b) A locally linear approximation to a non-convex function.

See Figure 5.3a for an illustration. Since f is convex, we have $B_f(\boldsymbol{w}||\boldsymbol{v}) \geq 0$, since $\hat{f}_v$ is a linear lower bound on f.

Below we mention some important special cases of Bregman divergences.

- If $f(\boldsymbol{w}) = ||\boldsymbol{w}||^2$, then $B_f(\boldsymbol{w}||\boldsymbol{v}) = ||\boldsymbol{w} - \boldsymbol{v}||^2$ is the squared Euclidean distance.
- If $f(\boldsymbol{w}) = \boldsymbol{w}^\mathsf{T}\mathbf{Q}\boldsymbol{w}$, then $B_f(\boldsymbol{w}||\boldsymbol{v})$ is the squared Mahalanobis distance.
- If $\boldsymbol{w}$ are the natural parameters of an exponential family distribution, and $f(\boldsymbol{w}) = \log Z(\boldsymbol{w})$ is the log normalizer, then the Bregman divergence is the same as the Kullback Leibler divergence, as we show in Section 5.1.8.1.

5.1.8.1 KL is a Bregman divergence

Recall that the log partition function $A(\boldsymbol{\eta})$ is a convex function. We can therefore use it to define the Bregman divergence (Section 5.1.8) between the two distributions, p and q, as follows:

$$B_f(\boldsymbol{\eta}_q||\boldsymbol{\eta}_p) = A(\boldsymbol{\eta}_q) - A(\boldsymbol{\eta}_p) - (\boldsymbol{\eta}_q - \boldsymbol{\eta}_p)^\mathsf{T}\nabla_{\boldsymbol{\eta}_p} A(\boldsymbol{\eta}_p) \tag{5.85}$$

$$= A(\boldsymbol{\eta}_q) - A(\boldsymbol{\eta}_p) - (\boldsymbol{\eta}_q - \boldsymbol{\eta}_p)^\mathsf{T}\mathbb{E}_p\left[\mathcal{T}(\boldsymbol{x})\right] \tag{5.86}$$

$$= D_{\mathbb{KL}}\left(p \parallel q\right) \tag{5.87}$$

where we exploited the fact that the gradient of the log partition function computes the expected sufficient statistics as shown in Section 2.3.3.

In fact, the KL divergence is the only divergence that is both a Bregman divergence and an f-divergence (Section 2.7.1) [Ama09].

5.2 Entropy

In this section, we discuss the **entropy** of a distribution p, which is just a shifted and scaled version of the KL divergence between the probability distribution and the uniform distribution, as we will see.

5.2.1 Definition

The entropy of a discrete random variable X with distribution p over K states is defined by

$$\mathbb{H}(X) \triangleq -\sum_{k=1}^{K} p(X=k) \log p(X=k) = -\mathbb{E}_X[\log p(X)] \tag{5.88}$$

We can use logarithms to any base, but we commonly use log base 2, in which case the units are called bits, or log base e, in which case the units are called nats, as we explained in Section 5.1.3.1.

The entropy is equivalent to a constant minus the KL divergence from the uniform distribution:

$$\mathbb{H}(X) = \log K - D_{\mathbb{KL}}(p(X) \parallel u(X)) \tag{5.89}$$

$$D_{\mathbb{KL}}(p(X) \parallel u(X)) = \sum_{k=1}^{K} p(X=k) \log \frac{p(X=k)}{\frac{1}{K}} \tag{5.90}$$

$$= \log K + \sum_{k=1}^{K} p(X=k) \log p(X=k) \tag{5.91}$$

If p is uniform, the KL is zero, and we see that the entropy achieves its maximal value of $\log K$.

For the special case of binary random variables, $X \in \{0, 1\}$, we can write $p(X=1) = \theta$ and $p(X=0) = 1-\theta$. Hence the entropy becomes

$$\mathbb{H}(X) = -[p(X=1)\log p(X=1) + p(X=0)\log p(X=0)] \tag{5.92}$$

$$= -[\theta \log \theta + (1-\theta)\log(1-\theta)] \tag{5.93}$$

This is called the **binary entropy function**, and is also written $\mathbb{H}(\theta)$. We plot this in Figure 5.4. We see that the maximum value of 1 bit occurs when the distribution is uniform, $\theta = 0.5$. A fair coin requires a single yes/no question to determine its state.

5.2.2 Differential entropy for continuous random variables

If X is a continuous random variable with pdf $p(x)$, we define the **differential entropy** as

$$h(X) \triangleq -\int_{\mathcal{X}} dx\, p(x) \log p(x) \tag{5.94}$$

assuming this integral exists.

For example, one can show that the entropy of a d-dimensional Gaussian is

$$h(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})) = \frac{1}{2}\log|2\pi e\boldsymbol{\Sigma}| = \frac{1}{2}\log[(2\pi e)^d|\boldsymbol{\Sigma}|] = \frac{d}{2} + \frac{d}{2}\log(2\pi) + \frac{1}{2}\log|\boldsymbol{\Sigma}| \tag{5.95}$$

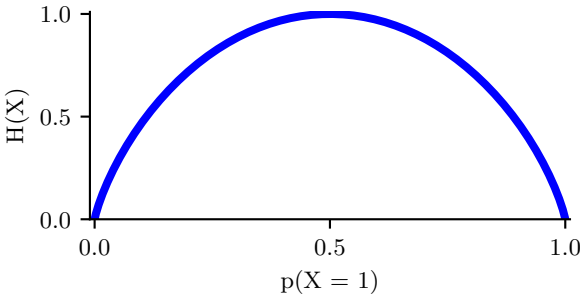


Figure 5.4: Entropy of a Bernoulli random variable as a function of θ. The maximum entropy is $\log_2 2 = 1$. Generated by bernoulli_entropy_fig.ipynb.

In the 1d case, this becomes

$$h(\mathcal{N}(\mu, \sigma^2)) = \frac{1}{2} \log \left[2\pi e \sigma^2\right] \tag{5.96}$$

Note that, unlike the discrete case, *differential entropy can be negative*. This is because pdf's can be bigger than 1. For example, suppose $X \sim U(0, a)$. Then

$$h(X) = -\int_0^a dx \, \frac{1}{a} \log \frac{1}{a} = \log a \tag{5.97}$$

If we set $a = 1/8$, we have $h(X) = \log_2(1/8) = -3$ bits.

One way to understand differential entropy is to realize that all real-valued quantities can only be represented to finite precision. It can be shown [CT91, p228] that the entropy of an n-bit quantization of a continuous random variable X is approximately $h(X) + n$. For example, suppose $X \sim U(0, \frac{1}{8})$. Then in a binary representation of X, the first 3 bits to the right of the binary point must be 0 (since the number is $\leq 1/8$). So to describe X to n bits of accuracy only requires $n - 3$ bits, which agrees with $h(X) = -3$ calculated above.

The continuous entropy also lacks the reparameterization independence of KL divergence Section 5.1.2.3. In particular, if we transform our random variable $y = f(x)$, the entropy transforms. To see this, note that the change of variables tells us that

$$p(y)\, dy = p(x)\, dx \implies p(y) = p(x) \left| \frac{dy}{dx} \right|^{-1}, \tag{5.98}$$

Thus the continuous entropy transforms as follows:

$$h(X) = -\int dx \, p(x) \log p(x) = h(Y) - \int dy \, p(y) \log \left| \frac{dy}{dx} \right|. \tag{5.99}$$

We pick up a factor in the continuous entropy of the log of the determinant of the Jacobian of the transformation. This changes the value for the continuous entropy even for simply rescaling the random variable such as when we change units. For example in Figure 5.5 we show the distribution

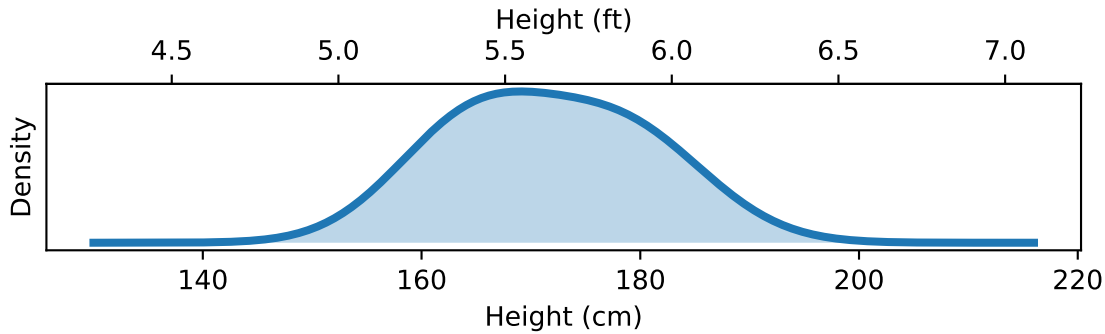


Figure 5.5: Distribution of adult heights. The continuous entropy of the distribution depends on its units of measurement. If heights are measured in feet, this distribution has a continuous entropy of 0.43 bits. If measured in centimeters it's 5.4 bits. If measured in meters it's -1.3 bits. Data taken from https://ourworldindata.org/human-height .

of adult human heights (it is bimodal because while both male and female heights are normally distributed, they differ noticeably). The continous entropy of this distribution depends on the units it is measured in. If measured in feet, the continuous entropy is 0.43 bits. Intuitively this is because human heights mostly span less than a foot. If measured in centimeters it is instead 5.4 bits. There are 30.48 centimeters in a foot, $\log_2 30.48 = 4.9$ explaining the difference. If we measured the continuous entropy of the same distribution measured in meters we would obtain -1.3 bits!

5.2.3 Typical sets

The **typical set** of a probability distribution is the set whose elements have an information content that is close to that of the expected information content from random samples from the distribution. More precisely, for a distribution $p(\boldsymbol{x})$ with support $\boldsymbol{x} \in \mathcal{X}$, the ϵ-typical set $\mathcal{A}_\epsilon^N \in \mathcal{X}^N$ for $p(\boldsymbol{x})$ is the set of all length N sequences such that

$$\mathbb{H}(p(\boldsymbol{x})) - \epsilon \leq -\frac{1}{N} \log p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) \leq \mathbb{H}(p(\boldsymbol{x})) + \epsilon \tag{5.100}$$

If we assume $p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) = \prod_{n=1}^N p(\boldsymbol{x}_n)$, then we can interpret the term in the middle as the N-sample empirical estimate of the entropy. The **asymptotic equipartition property** or **AEP** states that this will converge (in probability) to the true entropy as $N \to \infty$ [CT06]. Thus the typical set has probability close to 1, and is thus a compact summary of what we can expect to be generated by $p(\boldsymbol{x})$.

5.2.4 Cross entropy and perplexity

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), given by

$$D_{\mathbb{KL}}\left(p \parallel q\right) = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{H}_{ce}\left(p, q\right) - \mathbb{H}\left(p\right) \tag{5.101}$$

where $\mathbb{H}_{ce}\left(p, q\right)$ is the **cross entropy**

$$\mathbb{H}_{ce}\left(p, q\right) = -\sum_x p(x) \log q(x) \tag{5.102}$$

and $\mathbb{H}\left(p\right) = \mathbb{H}_{ce}\left(p, p\right)$ is the entropy, which is a constant independent of the model.

In language modeling, it is common to report an alternative performance measure known as the **perplexity**. This is defined as

$$\text{perplexity}(p, q) \triangleq 2^{\mathbb{H}_{ce}(p,q)} \tag{5.103}$$

We can compute an empirical approximation to the cross entropy as follows. Suppose we approximate the true distribution with an empirical distribution based on data sampled from p:

$$p_{\mathcal{D}}(x|\mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}\left(x = x_n\right) \tag{5.104}$$

In this case, the cross entropy is given by

$$H = -\frac{1}{N} \sum_{n=1}^{N} \log p(x_n) = -\frac{1}{N} \log \prod_{n=1}^{N} p(x_n) \tag{5.105}$$

The corresponding perplexity is given by

$$\text{perplexity}(p_{\mathcal{D}}, p) = 2^{-\frac{1}{N}\log(\prod_{n=1}^{N} p(x_n))} = 2^{\log(\prod_{n=1}^{N} p(x_n))^{-\frac{1}{N}}} \tag{5.106}$$

$$= (\prod_{n=1}^{N} p(x_n))^{-1/N} = \sqrt[N]{\prod_{n=1}^{N} \frac{1}{p(x_n)}} \tag{5.107}$$

In the case of language models, we usually condition on previous words when predicting the next word. For example, in a bigram model, we use a second order Markov model of the form $p(x_n|x_{n-1})$. We define the **branching factor** of a language model as the number of possible words that can follow any given word. For example, suppose the model predicts that each word is equally likely, regardless of context, so $p(x_n|x_{n-1}) = 1/K$, where K is the number of words in the vocabulary. Then the perplexity is $((1/K)^N)^{-1/N} = K$. If some symbols are more likely than others, and the model correctly reflects this, its perplexity will be lower than K. However, we have $\mathbb{H}\left(p^*\right) \leq \mathbb{H}_{ce}\left(p^*, p\right)$, so we can never reduce the perplexity below $2^{-\mathbb{H}(p^*)}$.

5.3 Mutual information

The KL divergence gave us a way to measure how similar two distributions were. How should we measure how dependant two random variables are? One thing we could do is turn the question of measuring the dependence of two random variables into a question about the similarity of their distributions. This gives rise to the notion of **mutual information** (MI) between two random variables, which we define below.

5.3.1 Definition

The mutual information between rv's X and Y is defined as follows:

$$\mathbb{I}(X;Y) \triangleq D_{\mathbb{KL}}(p(x,y) \parallel p(x)p(y)) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)} \tag{5.108}$$

(We write $\mathbb{I}(X;Y)$ instead of $\mathbb{I}(X,Y)$, in case X and/or Y represent sets of variables; for example, we can write $\mathbb{I}(X;Y,Z)$ to represent the MI between X and (Y,Z).) For continuous random variables, we just replace sums with integrals.

It is easy to see that MI is always non-negative, even for continuous random variables, since

$$\mathbb{I}(X;Y) = D_{\mathbb{KL}}(p(x,y) \parallel p(x)p(y)) \geq 0 \tag{5.109}$$

We achieve the bound of 0 iff $p(x,y) = p(x)p(y)$.

5.3.2 Interpretation

Knowing that the mutual information is a KL divergence between the joint and factored marginal distributions tells is that the MI measures the information gain if we update from a model that treats the two variables as independent $p(x)p(y)$ to one that models their true joint density $p(x,y)$.

To gain further insight into the meaning of MI, it helps to re-express it in terms of joint and conditional entropies, as follows:

$$\mathbb{I}(X;Y) = \mathbb{H}(X) - \mathbb{H}(X|Y) = \mathbb{H}(Y) - \mathbb{H}(Y|X) \tag{5.110}$$

Thus we can interpret the MI between X and Y as the reduction in uncertainty about X after observing Y, or, by symmetry, the reduction in uncertainty about Y after observing X. Incidentally, this result gives an alternative proof that conditioning, on average, reduces entropy. In particular, we have $0 \leq \mathbb{I}(X;Y) = \mathbb{H}(X) - \mathbb{H}(X|Y)$, and hence $\mathbb{H}(X|Y) \leq \mathbb{H}(X)$.

We can also obtain a different interpretation. One can show that

$$\mathbb{I}(X;Y) = \mathbb{H}(X,Y) - \mathbb{H}(X|Y) - \mathbb{H}(Y|X) \tag{5.111}$$

Finally, one can show that

$$\mathbb{I}(X;Y) = \mathbb{H}(X) + \mathbb{H}(Y) - \mathbb{H}(X,Y) \tag{5.112}$$

See Figure 5.6 for a summary of these equations in terms of an **information diagram**. (Formally, this is a signed measure mapping set expressions to their information-theoretic counterparts [Yeu91a].)

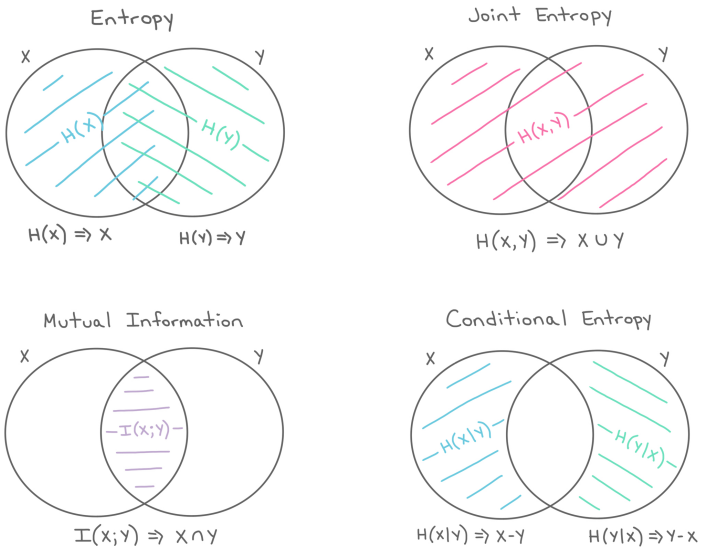


Figure 5.6: The marginal entropy, joint entropy, conditional entropy and mutual information represented as information diagrams. Used with kind permission of Katie Everett.

5.3.3 Data processing inequality

Suppose we have an unknown variable X, and we observe a noisy function of it, call it Y. If we process the noisy observations in some way to create a new variable Z, it should be intuitively obvious that we cannot increase the amount of information we have about the unknown quantity, X. This is known as the **data processing inequality**. We now state this more formally, and then prove it.

Theorem 5.3.1. *Suppose $X \to Y \to Z$ forms a Markov chain, so that $X \perp Z|Y$. Then $\mathbb{I}(X;Y) \geq \mathbb{I}(X;Z)$.*

Proof. By the chain rule for mutual information we can expand the mutual information in two different ways:

$$\mathbb{I}(X;Y,Z) = \mathbb{I}(X;Z) + \mathbb{I}(X;Y|Z) \tag{5.113}$$
$$= \mathbb{I}(X;Y) + \mathbb{I}(X;Z|Y) \tag{5.114}$$

Since $X \perp Z|Y$, we have $\mathbb{I}(X;Z|Y) = 0$, so

$$\mathbb{I}(X;Z) + \mathbb{I}(X;Y|Z) = \mathbb{I}(X;Y) \tag{5.115}$$

Since $\mathbb{I}(X;Y|Z) \geq 0$, we have $\mathbb{I}(X;Y) \geq \mathbb{I}(X;Z)$. Similarly one can prove that $\mathbb{I}(Y;Z) \geq \mathbb{I}(X;Z)$. □

5.3.4 Sufficient Statistics

An important consequence of the DPI is the following. Suppose we have the chain $\theta \to X \to s(X)$. Then

$$\mathbb{I}(\theta; s(X)) \leq \mathbb{I}(\theta; X) \tag{5.116}$$

If this holds with equality, then we say that $s(X)$ is a **sufficient statistic** of the data X for the purposes of inferring θ. In this case, we can equivalently write $\theta \to s(X) \to X$, since we can reconstruct the data from knowing $s(X)$ just as accurately as from knowing θ.

An example of a sufficient statistic is the data itself, $s(X) = X$, but this is not very useful, since it doesn't summarize the data at all. Hence we define a **minimal sufficient statistic** $s(X)$ as one which is sufficient, and which contains no extra information about θ; thus $s(X)$ maximally compresses the data X without losing information which is relevant to predicting θ. More formally, we say s is a minimal sufficient statistic for X if $s(X) = f(s'(X))$ for some function f and all sufficient statistics $s'(X)$. We can summarize the situation as follows:

$$\theta \to s(X) \to s'(X) \to X \tag{5.117}$$

Here $s'(X)$ takes $s(X)$ and adds redundant information to it, thus creating a one-to-many mapping.

For example, a minimal sufficient statistic for a set of N Bernoulli trials is simply N and $N_1 = \sum_n \mathbb{I}(X_n = 1)$, i.e., the number of successes. In other words, we don't need to keep track of the entire sequence of heads and tails and their ordering, we only need to keep track of the total number of heads and tails. Similarly, for inferring the mean of a Gaussian distribution with known variance we only need to know the empirical mean and number of samples.

Earlier in Section 5.1.7 we motivated the exponential family of distributions as being the ones that are minimal in the sense that they contain no other information than constraints on some statistics of the data. It makes sense then that the statistics used to generate exponential family distributions are sufficient. It also hints at the more remarkable fact of the **Pitman-Koopman-Darmois theorem**, which says that for any distribution whose domain is fixed, it is only the exponential family that admits sufficient statistics with bounded dimensionality as the number of samples increases Diaconis [Dia88].

5.3.5 Multivariate mutual information

There are several ways to generalize the idea of mutual information to a set of random variables as we discuss below.

5.3.5.1 Total correlation

The simplest way to define multivariate MI is to use the **total correlation** [Wat60] or **multi-information** [SV98], defined as

$$\mathbb{TC}(\{X_1, \ldots, X_D\}) \triangleq D_{\mathbb{KL}}\left(p(\boldsymbol{x}) \parallel \prod_d p(x_d)\right) \tag{5.118}$$

$$= \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{\prod_{d=1}^{D} p(x_d)} = \sum_d \mathbb{H}(x_d) - \mathbb{H}(\boldsymbol{x}) \tag{5.119}$$

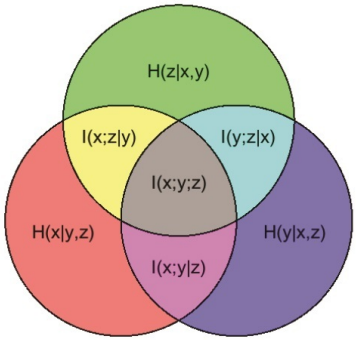


Figure 5.7: Illustration of multivariate mutual information between three random variables. From https://en.wikipedia.org/wiki/Mutual_information. Used with kind permission of Wikipedia author PAR.

For example, for 3 variables, this becomes

$$\mathbb{TC}(X,Y,Z) = \mathbb{H}(X) + \mathbb{H}(Y) + \mathbb{H}(Z) - \mathbb{H}(X,Y,Z) \tag{5.120}$$

where $\mathbb{H}(X,Y,Z)$ is the joint entropy

$$\mathbb{H}(X,Y,Z) = -\sum_x \sum_y \sum_z p(x,y,z) \log p(x,y,z) \tag{5.121}$$

One can show that the multi-information is always non-negative, and is zero iff $p(\boldsymbol{x}) = \prod_d p(x_d)$. However, this means the quantity is non-zero even if only a pair of variables interact. For example, if $p(X,Y,Z) = p(X,Y)p(Z)$, then the total correlation will be non-zero, even though there is no 3 way interaction. This motivates the alternative definition in Section 5.3.5.2.

5.3.5.2 Interaction information (co-information)

The conditional mutual information can be used to give an inductive definition of the **multivariate mutual information** (**MMI**) as follows:

$$\mathbb{I}(X_1;\cdots;X_D) = \mathbb{I}(X_1;\cdots;X_{D-1}) - \mathbb{I}(X_1;\cdots;X_{D-1}|X_D) \tag{5.122}$$

This is called the **multiple mutual information** [Yeu91b], or the **co-information** [Bel03]. This definition is equivalent, up to a sign change, to the **interaction information** [McG54; Han80; JB03; Bro09].

For 3 variables, the MMI is given by

$$\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Y) - \mathbb{I}(X;Y|Z) \tag{5.123}$$

$$= \mathbb{I}(X;Z) - \mathbb{I}(X;Z|Y) \tag{5.124}$$

$$= \mathbb{I}(Y;Z) - \mathbb{I}(Y;Z|X) \tag{5.125}$$

This can be interpreted as the change in mutual information between two pairs of variables when conditioning on the third. Note that this quantity is symmetric in its arguments.

By the definition of conditional mutual information, we have

$$\mathbb{I}(X;Z|Y) = \mathbb{I}(Z;X,Y) - \mathbb{I}(Y;Z) \tag{5.126}$$

Hence we can rewrite Equation (5.124) as follows:

$$\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Z) + \mathbb{I}(Y;Z) - \mathbb{I}(X,Y;Z) \tag{5.127}$$

This tells us that the MMI is the difference between how much we learn about Z given X and Y individually vs jointly (see also Section 5.3.5.3).

The 3-way MMI is illustrated in the information diagram in Figure 5.7. The way to interpret such diagrams when we have multiple variables is as follows: the area of a shaded area that includes circles $A, B, C, \ldots$ and excludes circles $F, G, H, \ldots$ represents $\mathbb{I}(A;B;C;\ldots|F,G,H,\ldots)$; if $B = C = \emptyset$, this is just $\mathbb{H}(A|F,G,H,\ldots)$; if $F = G = H = \emptyset$, this is just $\mathbb{I}(A;B;C,\ldots)$.

5.3.5.3 Synergy and redundancy

The MMI is $\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Z) + \mathbb{I}(Y;Z) - \mathbb{I}(X,Y;Z)$. We see that this can be positive, zero *or negative*. If some of the information about Z that is provided by X is also provided by Y, then there is some **redundancy** between X and Y (wrt Z). In this case, $\mathbb{I}(X;Z) + \mathbb{I}(Y;Z) > \mathbb{I}(X,Y;Z)$, so (from Equation (5.127)) we see that the MMI will be positive. If, by contrast, we learn more about Z when we see X and Y together, we say there is some **synergy** between them. In this case, $\mathbb{I}(X;Z) + \mathbb{I}(Y;Z) < \mathbb{I}(X,Y;Z)$, so the MMI will be negative.

5.3.5.4 MMI and causality

The sign of the MMI can be used to distinguish between different kinds of directed graphical models, which can sometimes be interpreted causally (see Chapter 36 for a general discussion of causality). For example, consider a model of the form $X \leftarrow Z \rightarrow Y$, where Z is a "cause" of X and Y. For example, suppose X represents the event it is raining, Y represents the event that the sky is dark, and Z represents the event that the sky is cloudy. Conditioning on the common cause Z renders the children X and Y independent, since if I know it is cloudy, noticing that the sky is dark does not change my beliefs about whether it will rain or not. Consequently $\mathbb{I}(X;Y|Z) \leq \mathbb{I}(X;Y)$, so $\mathbb{I}(X;Y;Z) \geq 0$.

Now consider the case where Z is a common effect, $X \rightarrow Z \leftarrow Y$. In this case, conditioning on Z makes X and Y dependent, due to the explaining away phenomenon (see Section 4.2.4.2). For example, if X and Y are independent random bits, and Z is the XOR of X and Y, then observing $Z = 1$ means that $p(X \neq Y|Z = 1) = 1$, so X and Y are now dependent (information-theoretically, not causally), even though they were a priori independent. Consequently $\mathbb{I}(X;Y|Z) \geq \mathbb{I}(X;Y)$, so $\mathbb{I}(X;Y;Z) \leq 0$.

Finally, consider a Markov chain, $X \rightarrow Y \rightarrow Z$. We have $\mathbb{I}(X;Z|Y) \leq \mathbb{I}(X;Z)$ and so the MMI must be positive.

5.3.5.5 MMI and entropy

We can also write the MMI in terms of entropies. Specifically, we know that

$$\mathbb{I}(X;Y) = \mathbb{H}(X) + \mathbb{H}(Y) - \mathbb{H}(X,Y) \tag{5.128}$$

and

$$\mathbb{I}(X;Y|Z) = \mathbb{H}(X,Z) + \mathbb{H}(Y,Z) - \mathbb{H}(Z) - \mathbb{H}(X,Y,Z) \tag{5.129}$$

Hence we can rewrite Equation (5.123) as follows:

$$\mathbb{I}(X;Y;Z) = [\mathbb{H}(X) + \mathbb{H}(Y) + \mathbb{H}(Z)] - [\mathbb{H}(X,Y) + \mathbb{H}(X,Z) + \mathbb{H}(Y,Z)] + \mathbb{H}(X,Y,Z) \tag{5.130}$$

Contrast this to Equation (5.120).

More generally, we have

$$\mathbb{I}(X_1,\ldots,X_D) = -\sum_{\mathcal{T}\subseteq\{1,\ldots,D\}} (-1)^{|\mathcal{T}|}\,\mathbb{H}(\mathcal{T}) \tag{5.131}$$

For sets of size 1, 2 and 3 this expands as follows:

$$I_1 = H_1 \tag{5.132}$$

$$I_{12} = H_1 + H_2 - H_{12} \tag{5.133}$$

$$I_{123} = H_1 + H_2 + H_3 - H_{12} - H_{13} - H_{23} + H_{123} \tag{5.134}$$

We can use the **Mobius inversion formula** to derive the following dual relationship:

$$\mathbb{H}(\mathcal{S}) = -\sum_{\mathcal{T}\subseteq\mathcal{S}} (-1)^{|\mathcal{T}|}\,\mathbb{I}(\mathcal{T}) \tag{5.135}$$

for sets of variables $\mathcal{S}$.

Using the chain rule for entropy, we can also derive the following expression for the 3-way MMI:

$$\mathbb{I}(X;Y;Z) = \mathbb{H}(Z) - \mathbb{H}(Z|X) - \mathbb{H}(Z|Y) + \mathbb{H}(Z|X,Y) \tag{5.136}$$

5.3.6 Variational bounds on mutual information

In this section, we discuss methods for computing upper and lower bounds on MI that use variational approximations to the intractable distributions. This can be useful for representation learning (Chapter 32). This approach was first suggested in [BA03]. For a more detailed overview of variational bounds on mutual information, see Poole et al. [Poo+19b].

5.3.6.1 Upper bound

Suppose that the joint $p(\boldsymbol{x},\boldsymbol{y})$ is intractable to evaluate, but that we can sample from $p(\boldsymbol{x})$ and evaluate the conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$. Furthermore, suppose we approximate $p(\boldsymbol{y})$ by $q(\boldsymbol{y})$.

Then we can compute an upper bound on the MI as follows:

$$\mathbb{I}(\boldsymbol{x};\boldsymbol{y}) = \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{p(\boldsymbol{y}|\boldsymbol{x})q(\boldsymbol{y})}{p(\boldsymbol{y})q(\boldsymbol{y})}\right] \tag{5.137}$$

$$= \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{p(\boldsymbol{y}|\boldsymbol{x})}{q(\boldsymbol{y})}\right] - D_{\mathbb{KL}}\left(p(\boldsymbol{y}) \parallel q(\boldsymbol{y})\right) \tag{5.138}$$

$$\leq \mathbb{E}_{p(\boldsymbol{x})}\left[\mathbb{E}_{p(\boldsymbol{y}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{y}|\boldsymbol{x})}{q(\boldsymbol{y})}\right]\right] \tag{5.139}$$

$$= \mathbb{E}_{p(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(p(\boldsymbol{y}|\boldsymbol{x}) \parallel q(\boldsymbol{y})\right)\right] \tag{5.140}$$

This bound is tight if $q(\boldsymbol{y}) = p(\boldsymbol{y})$.

What's happening here is that $\mathbb{I}(Y;X) = \mathbb{H}(Y) - \mathbb{H}(Y|X)$ and we've assumed we know $p(\boldsymbol{y}|\boldsymbol{x})$ and so can estimate $\mathbb{H}(Y|X)$ well. While we don't know $\mathbb{H}(Y)$, we can upper bound it using some model $q(\boldsymbol{y})$. Our model can never do better than $p(\boldsymbol{y})$ itself (the non-negativity of KL), so our entropy estimate errs too large, and hence our MI estimate will be an upper bound.

5.3.6.2 BA lower bound

Suppose that the joint $p(\boldsymbol{x},\boldsymbol{y})$ is intractable to evaluate, but that we can evaluate $p(\boldsymbol{x})$. Furthermore, suppose we approximate $p(\boldsymbol{x}|\boldsymbol{y})$ by $q(\boldsymbol{x}|\boldsymbol{y})$. Then we can derive the following variational lower bound on the mutual information:

$$\mathbb{I}(\boldsymbol{x};\boldsymbol{y}) = \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{p(\boldsymbol{x}|\boldsymbol{y})}{p(\boldsymbol{x})}\right] \tag{5.141}$$

$$= \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{q(\boldsymbol{x}|\boldsymbol{y})}{p(\boldsymbol{x})}\right] + \mathbb{E}_{p(\boldsymbol{y})}\left[D_{\mathbb{KL}}\left(p(\boldsymbol{x}|\boldsymbol{y}) \parallel q(\boldsymbol{x}|\boldsymbol{y})\right)\right] \tag{5.142}$$

$$\geq \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{q(\boldsymbol{x}|\boldsymbol{y})}{p(\boldsymbol{x})}\right] = \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log q(\boldsymbol{x}|\boldsymbol{y})\right] + h(\boldsymbol{x}) \tag{5.143}$$

where $h(\boldsymbol{x})$ is the differential entropy of $\boldsymbol{x}$. This is called the **BA lower bound**, after the authors Barber and Agakov [BA03].

5.3.6.3 NWJ lower bound

The BA lower bound requires a tractable normalized distribution $q(\boldsymbol{x}|\boldsymbol{y})$ that we can evaluate pointwise. If we reparameterize this distribution in a clever way, we can generate a lower bound that does not require a normalized distribution. Let's write:

$$q(\boldsymbol{x}|\boldsymbol{y}) = \frac{p(\boldsymbol{x})e^{f(\boldsymbol{x},\boldsymbol{y})}}{Z(\boldsymbol{y})} \tag{5.144}$$

with $Z(\boldsymbol{y}) = \mathbb{E}_{p(\boldsymbol{x})}\left[e^{f(\boldsymbol{x},\boldsymbol{y})}\right]$ the normalization constant or partition function. Plugging this into the BA lower bound above we obtain:

$$\mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[\log \frac{p(\boldsymbol{x})e^{f(\boldsymbol{x},\boldsymbol{y})}}{p(\boldsymbol{x})Z(\boldsymbol{y})}\right] = \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[f(\boldsymbol{x},\boldsymbol{y})\right] - \mathbb{E}_{p(\boldsymbol{y})}\left[Z(\boldsymbol{y})\right] \tag{5.145}$$

$$= \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}\left[f(\boldsymbol{x},\boldsymbol{y})\right] - \mathbb{E}_{p(\boldsymbol{y})}\left[\log \mathbb{E}_{p(\boldsymbol{x})}\left[e^{f(\boldsymbol{x},\boldsymbol{y})}\right]\right] \tag{5.146}$$

$$\triangleq I_{DV}(X;Y). \tag{5.147}$$

This is the **Donsker Varadhan lower bound** [DV75].

We can construct a more tractable version of this by using the fact that the log function can be upper bounded by a straight line using

$$\log x \leq \frac{x}{a} + \log a - 1 \tag{5.148}$$

If we set $a = e$, we get

$$\mathbb{I}(X;Y) \geq \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}[f(\boldsymbol{x},\boldsymbol{y})] - e^{-1}\mathbb{E}_{p(\boldsymbol{y})}Z(\boldsymbol{y}) \triangleq I_{NWJ}(X;Y) \tag{5.149}$$

This is called the **NWJ lower bound** (after the authors of Nguyen, Wainwright, and Jordan [NWJ10a]), or the f-GAN KL [NCT16a], or the MINE-f score [Bel+18].

5.3.6.4 InfoNCE lower bound

If we instead explore a multi-sample extension to the DV bound above, we can generate the following lower bound (see [Poo+19b] for the derivation):

$$\mathbb{I}_{\text{NCE}} = \mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\log \frac{e^{f(\boldsymbol{x}_i,\boldsymbol{y}_i)}}{\frac{1}{K}\sum_{j=1}^{K} e^{f(\boldsymbol{x}_i,\boldsymbol{y}_j)}}\right] \tag{5.150}$$

$$= \log K - \mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\log\left(1 + \sum_{j\neq i}^{K} e^{f(\boldsymbol{x}_i,\boldsymbol{y}_j) - f(\boldsymbol{x}_i,\boldsymbol{y}_i)}\right)\right] \tag{5.151}$$

where the expectation is over paired samples from the joint $p(X,Y)$. The quantity in Equation (5.151) is called the **InfoNCE** estimate, and was proposed in [OLV18a; Hen+19a]. (NCE stands for "noise contrastive estimation", and is discussed in Section 24.4.)

The intuition here is that mutual information is a divergence between the joint $p(\boldsymbol{x},\boldsymbol{y})$ and the product of the marginals, $p(\boldsymbol{x})p(\boldsymbol{y})$. In other words, mutual information is a measurement of how distinct sampling pairs jointly is from sampling $\boldsymbol{x}$s and $\boldsymbol{y}$s independently. The InfoNCE bound provides a lower bound on the true mutual information by attempting to train a model to distinguish between these two situations.

Although this is a valid lower bound, we may need to use a large batch size K to estimate the MI if the MI is large, since $\mathbb{I}_{\text{NCE}} \leq \log K$. (Recently [SE20a] proposed to use a multi-label classifier, rather than a multi-class classifier, to overcome this limitation.)

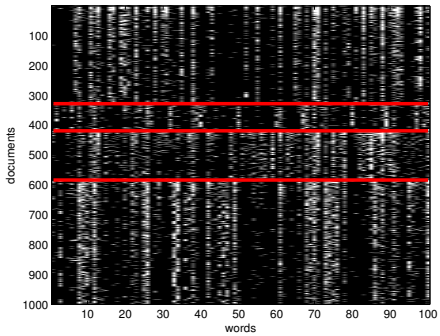


Figure 5.8: Subset of size 16242 x 100 of the 20-newsgroups data. We only show 1000 rows, for clarity. Each row is a document (represented as a bag-of-words bit vector), each column is a word. The red lines separate the 4 classes, which are (in descending order) comp, rec, sci, talk (these are the titles of USENET groups). We can see that there are subsets of words whose presence or absence is indicative of the class. The data is available from `http://cs.nyu.edu/~roweis/data.html`. *Generated by newsgroups_visualize.ipynb.*

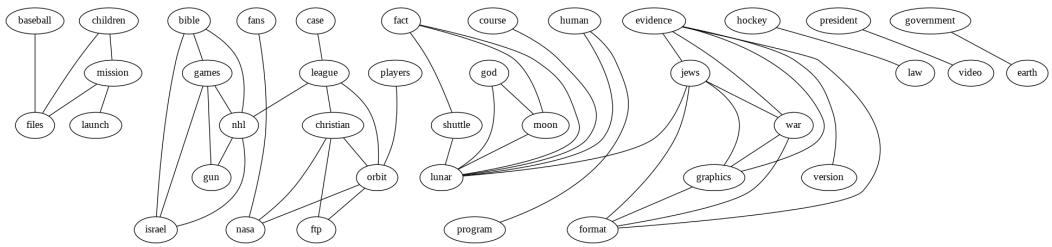


Figure 5.9: Part of a relevance network constructed from the 20 newsgroup data. data shown in Figure 5.8. We show edges whose mutual information is greater than or equal to 20% of the maximum pairwise MI. For clarity, the graph has been cropped, so we only show a subset of the nodes and edges. Generated by relevance_network_newsgroup_demo.ipynb.

5.3.7 Relevance networks

If we have a set of related variables, we can compute a **relevance network**, in which we add an $i-j$ edge if the pairwise mutual information $\mathbb{I}(X_i; X_j)$ is above some threshold. In the Gaussian case, $\mathbb{I}(X_i; X_j) = -\frac{1}{2}\log(1-\rho_{ij}^2)$, where ρ_{ij} is the correlation coefficient, and the resulting graph is called a **covariance graph** (Section 4.5.5.1). However, we can also apply it to discrete random variables.

Relevance networks are quite popular in systems biology [Mar+06], where they are used to visualize the interaction between genes. But they can also be applied to other kinds of datasets. For example, Figure 5.9 visualizes the MI between words in the 20 newsgroup dataset shown in Figure 5.8. The results seem intuitively reasonable.

However, relevance networks suffer from a major problem: the graphs are usually very dense, since most variables are dependent on most other variables, even after thresholding the MIs. For example, suppose X_1 directly influences X_2 which directly influences X_3 (e.g., these form components of a signalling cascade, $X_1 - X_2 - X_3$). Then X_1 has non-zero MI with X_3 (and vice versa), so there

will be a $1-3$ edge as well as the $1-2$ and $2-3$ edges; thus the graph may be fully connected, depending on the threshold.

A solution to this is to learn a probablistic graphical model, which represents conditional *independence*, rather than *dependence*. In the chain example, there will not be a $1-3$ edge, since $X_1 \perp X_3|X_2$. Consequently graphical models are usually much sparser than relevance networks. See Chapter 30 for details.

5.4 Data compression (source coding)

Data compression, also known as **source coding**, is at the heart of information theory. It is also related to probabilistic machine learning. The reason for this is as follows: if we can model the probability of different kinds of data samples, then we can assign short **code words** to the most frequently occuring ones, reserving longer encodings for the less frequent ones. This is similar to the situation in natural language, where common words (such as "a", "the", "and") are generally much shorter than rare words. Thus the ability to compress data requires an ability to discover the underlying patterns, and their relative frequencies, in the data. This has led Marcus Hutter to propose that compression be used as an objective way to measure performance towards general purpose AI. More precisely, he is offering 50,000 Euros to anyone who can compress the first 100MB of (English) Wikipedia better than some baseline. This is known as the **Hutter prize**.[1]

In this section, we give a brief summary of some of the key ideas in data compression. For details, see e.g., [Mac03; CT06; YMT22].

5.4.1 Lossless compression

Discrete data, such as natural language, can always be compressed in such a way that we can uniquely recover the original data. This is called **lossless compression**.

Claude Shannon proved that the expected number of bits needed to losslessly encode some data coming from distribution p is at least $\mathbb{H}(p)$. This is known as the **source coding theorem**. Achieving this lower bound requires coming up with good probability models, as well as good ways to design codes based on those models. Because of the non-negativity of the KL divergence, $\mathbb{H}_{ce}(p,q) \geq \mathbb{H}(p)$, so if we use any model q other than the true model p to compress the data, it will take some excess bits. The number of excess bits is exactly $D_{\mathbb{KL}}(p \parallel q)$.

Common techniques for realizing lossless codes include Huffman coding, arithmetic coding and asymmetric numeral systems [Dud13]. The input to these algorithms is a probability distribution over strings (which is where ML comes in). This distribution is often represented using a latent variable model (see e.g., [TBB19; KAH19]).

5.4.2 Lossy compression and the rate-distortion tradeoff

To encode real-valued signals, such as images and sound, as a digital signal, we first have to quantize the signal into a sequence of symbols. A simple way to do this is to use vector quantization. We can then compress this discrete sequence of symbols using lossless coding methods. However, when we uncompress, we lose some information. Hence this approach is called **lossy compression**.

1. For details, see http://prize.hutter1.net.

In this section, we quantify this tradeoff between the size of the representation (number of symbols we use), and the resulting error. We will use the terminology of the variational information bottleneck discussed in Section 5.6.2 (except here we are in the unsupervised setting). In particular, we assume we have a stochastic encoder $p(\boldsymbol{z}|\boldsymbol{x})$, a stochastic decoder $d(\boldsymbol{x}|\boldsymbol{z})$ and a prior marginal $m(\boldsymbol{z})$.

We define the **distortion** of an encoder-decoder pair (as in Section 5.6.2) as follows:

$$D = -\int d\boldsymbol{x}\; p(\boldsymbol{x}) \int d\boldsymbol{z}\; e(\boldsymbol{z}|\boldsymbol{x}) \log d(\boldsymbol{x}|\boldsymbol{z}) \tag{5.152}$$

If the decoder is a deterministic model plus Gaussian noise, $d(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}|f_d(\boldsymbol{z}), \sigma^2)$, and the encoder is deterministic, $e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - f_e(\boldsymbol{x}))$, then this becomes

$$D = \frac{1}{\sigma^2}\mathbb{E}_{p(\boldsymbol{x})}\left[||f_d(f_e(\boldsymbol{x})) - \boldsymbol{x}||^2\right] \tag{5.153}$$

This is just the expected **reconstruction error** that occurs if we (deterministically) encode and then decode the data using f_e and f_d.

We define the **rate** of our model as follows:

$$R = \int d\boldsymbol{x}\; p(\boldsymbol{x}) \int d\boldsymbol{z}\; e(\boldsymbol{z}|\boldsymbol{x}) \log \frac{e(\boldsymbol{z}|\boldsymbol{x})}{m(\boldsymbol{z})} \tag{5.154}$$

$$= \mathbb{E}_{p(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(e(\boldsymbol{z}|\boldsymbol{x}) \parallel m(\boldsymbol{z})\right)\right] \tag{5.155}$$

$$= \int d\boldsymbol{x} \int d\boldsymbol{z}\; p(\boldsymbol{x}, \boldsymbol{z}) \log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{p(\boldsymbol{x})m(\boldsymbol{z})} \geq \mathbb{I}(\boldsymbol{x}, \boldsymbol{z}) \tag{5.156}$$

This is just the average KL between our encoding distribution and the marginal. If we use $m(\boldsymbol{z})$ to design an optimal code, then the rate is the *excess* number of bits we need to pay to encode our data using $m(\boldsymbol{z})$ rather than the true **aggregate posterior** $p(\boldsymbol{z}) = \int d\boldsymbol{x}\; p(\boldsymbol{x})e(\boldsymbol{z}|\boldsymbol{x})$.

There is a fundamental tradeoff between the rate and distortion. To see why, note that a trivial encoding scheme would set $e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - \boldsymbol{x})$, which simply uses $\boldsymbol{x}$ as its own best representation. This would incur 0 distortion (and hence maximize the likelihood), but it would incur a high rate, since each $e(\boldsymbol{z}|\boldsymbol{x})$ distribution would be unique, and far from $m(\boldsymbol{z})$. In other words, there would be no compression. Conversely, if $e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - \boldsymbol{0})$, the encoder would ignore the input. In this case, the rate would be 0, but the distortion would be high.

We can characterize the tradeoff more precisely using the variational lower and upper bounds on the mutual information from Section 5.3.6. From that section, we know that

$$H - D \leq \mathbb{I}(\boldsymbol{x}; \boldsymbol{z}) \leq R \tag{5.157}$$

where H is the (differential) entropy

$$H = -\int d\boldsymbol{x}\; p(\boldsymbol{x}) \log p(\boldsymbol{x}) \tag{5.158}$$

For discrete data, all probabilities are bounded above by 1, and hence $H \geq 0$ and $D \geq 0$. In addition, the rate is always non-negative, $R \geq 0$, since it is the average of a KL divergence. (This is true for either discrete or continuous encodings $\boldsymbol{z}$.) Consequently, we can plot the set of achievable values of R and D as shown in Figure 5.10. This is known as a **rate distortion curve**.

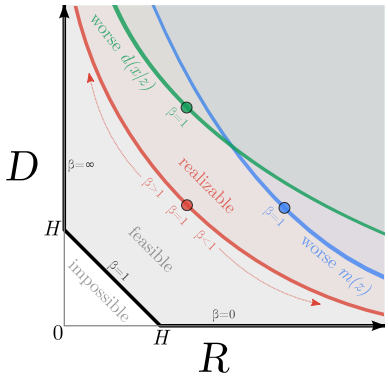


Figure 5.10: Illustration of the rate-distortion tradeoff. See text for details. From Figure 1 of [Ale+18]. Used with kind permission of Alex Alemi.

The bottom horizontal line corresponds to the zero distortion setting, $D = 0$, in which we can perfectly encode and decode our data. This can be achieved by using the trivial encoder where $e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - \boldsymbol{x})$. Shannon's source coding theorem tells us that the minimum number of bits we need to use to encode data in this setting is the entropy of the data, so $R \geq H$ when $D = 0$. If we use a suboptimal marginal distribution $m(\boldsymbol{z})$ for coding, we will increase the rate without affecting the distortion.

The left vertical line corresponds to the zero rate setting, $R = 0$, in which the latent code is independent of $\boldsymbol{z}$. In this case, the decoder $d(\boldsymbol{x}|\boldsymbol{z})$ is independent of $\boldsymbol{z}$. However, we can still learn a joint probability model $p(\boldsymbol{x})$ which does not use latent variables, e.g., this could be an autoregressive model. The minimal distortion such a model could achieve is again the entropy of the data, $D \geq H$.

The black diagonal line illustrates solutions that satisfy $D = H - R$, where the upper and lower bounds are tight. In practice, we cannot achieve points on the diagonal, since that requires the bounds to be tight, and therefore assumes our models $e(\boldsymbol{z}|\boldsymbol{x})$ and $d(\boldsymbol{x}|\boldsymbol{z})$ are perfect. This is called the "non-parametric limit". In the finite data setting, we will always incur additional error, so the RD plot will trace a curve which is shifted up, as shown in Figure 5.10.

We can generate different solutions along this curve by minimizing the following objective:

$$J = D + \beta R = \int d\boldsymbol{x}\, p(\boldsymbol{x}) \int d\boldsymbol{z}\, e(\boldsymbol{z}|\boldsymbol{x}) \left[-\log d(\boldsymbol{x}|\boldsymbol{z}) + \beta \log \frac{e(\boldsymbol{z}|\boldsymbol{x})}{m(\boldsymbol{z})} \right] \tag{5.159}$$

If we set $\beta = 1$, and define $q(\boldsymbol{z}|\boldsymbol{x}) = e(\boldsymbol{z}|\boldsymbol{x})$, $p(\boldsymbol{x}|\boldsymbol{z}) = d(\boldsymbol{x}|\boldsymbol{z})$, and $p(\boldsymbol{z}) = m(\boldsymbol{z})$, this exactly matches the VAE objective in Section 21.2. To see this, note that the ELBO from Section 10.1.2 can be written as

$$Ł = -(D + R) = \mathbb{E}_{p(\boldsymbol{x})} \left[\mathbb{E}_{e(\boldsymbol{z}|\boldsymbol{x})} \left[\log d(\boldsymbol{x}|\boldsymbol{z}) \right] - \mathbb{E}_{e(\boldsymbol{z}|\boldsymbol{x})} \left[\log \frac{e(\boldsymbol{z}|\boldsymbol{x})}{m(\boldsymbol{z})} \right] \right] \tag{5.160}$$

which we recognize as the expected reconstruction error minus the KL term $D_{\mathbb{KL}}\left(e(\boldsymbol{z}|\boldsymbol{x}) \parallel m(\boldsymbol{z})\right)$.

If we allow $\beta \neq 1$, we recover the β-VAE objective discussed in Section 21.3.1. Note, however, that the β-VAE model cannot distinguish between different solutions on the diagonal line, all of which have

$\beta = 1$. This is because all such models have the same marginal likelihood (and hence same ELBO), although they differ radically in terms of whether they learn an interesting latent representation or not. Thus likelihood is not a sufficient metric for comparing the quality of unsupervised representation learning methods, as discussed in Section 21.3.1.

For further discussion on the inherent conflict between rate, distortion and *perception*, see Blau and Michaeli [BM19]. For techniques for evaluating rate distortion curves for models see Huang, Cao, and Grosse [HCG20].

5.4.3 Bits back coding

In the previous section we penalized the rate of our code using the average KL divergence, $\mathbb{E}_{p(\boldsymbol{x})}[R(\boldsymbol{x})]$, where

$$R(\boldsymbol{x}) \triangleq \int d\boldsymbol{z}\, p(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p(\boldsymbol{z}|\boldsymbol{x})}{m(\boldsymbol{z})} = \mathbb{H}_{ce}(p(\boldsymbol{z}|\boldsymbol{x}), m(\boldsymbol{z})) - \mathbb{H}(p(\boldsymbol{z}|\boldsymbol{x})). \tag{5.161}$$

The first term is the cross entropy, which is the expected number of bits we need to encode $\boldsymbol{x}$; the second term is the entropy, which is the minimum number of bits. Thus we are penalizing the *excess* number of bits required to communicate the code to a receiver. How come we don't have to "pay for" the actual (total) number of bits we use, which is the cross entropy?

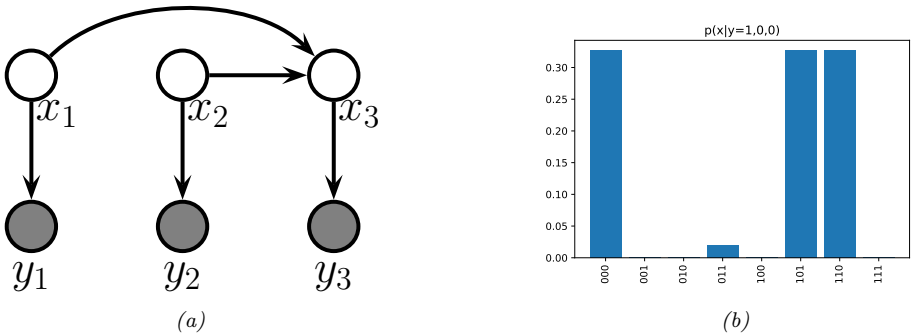
The reason is that we could in principle get the bits needed by the optimal code given back to us; this is called **bits back coding** [HC93; FH97]. The argument goes as follows. Imagine Alice is trying to (losslessly) communicate some data, such as an image $\boldsymbol{x}$, to Bob. Before they went their separate ways, both Alice and Bob decided to share their encoder $p(\boldsymbol{z}|\boldsymbol{x})$, marginal $m(\boldsymbol{z})$ and decoder distributions $d(\boldsymbol{x}|\boldsymbol{z})$. To communicate an image, Alice will use a **two part code**. First, she will sample a code $\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{x})$ from her encoder, and communicate that to Bob over a channel designed to efficiently encode samples from the marginal $m(\boldsymbol{z})$; this costs $-\log_2 m(\boldsymbol{z})$ bits. Next Alice will use her decoder $d(\boldsymbol{x}|\boldsymbol{z})$ to compute the residual error, and losslessly send that to Bob at the cost of $-\log_2 d(\boldsymbol{x}|\boldsymbol{z})$ bits. The expected total number of bits required here is what we naively expected:

$$\mathbb{E}_{p(\boldsymbol{z}|\boldsymbol{x})}[-\log_2 d(\boldsymbol{x}|\boldsymbol{z}) - \log_2 m(\boldsymbol{z})] = D + \mathbb{H}_{ce}(p(\boldsymbol{z}|\boldsymbol{x}), m(\boldsymbol{z})). \tag{5.162}$$

We see that this is the distortion plus cross entropy, not distortion plus rate. So how do we get the bits back, to convert the cross entropy to a rate term?

The trick is that Bob actually receives more information than we suspected. Bob can use the code $\boldsymbol{z}$ and the residual error to perfectly reconstruct $\boldsymbol{x}$. However, Bob also knows what specific code Alice sent, $\boldsymbol{z}$, as well as what encoder she used, $p(\boldsymbol{z}|\boldsymbol{x})$. When Alice drew the sample code $\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{x})$, she had to use some kind of entropy source in order to generate the random sample. Suppose she did it by picking words sequentially from a compressed copy of Moby Dick, in order to generate a stream of random bits. On Bob's end, he can reverse engineer all of the sampling bits, and thus recover the compressed copy of Moby Dick! Thus Alice can use the extra randomness in the choice of $\boldsymbol{z}$ to share more information.

While in the original formulation the bits-back argument was largely theoretical, offering a thought experiment for why we should penalize our models with the KL instead of the cross entropy, recently several practical real world algorithms have been developed that actually achieve the bits-back goal. These include [HHLMF18; AT20; TBB19; YBM20; HLA19].



(a) *(b)*

Figure 5.11: (a) A simple error-correcting code DPGM. x_i are the sent bits, y_i are the received bits. x_3 is an even parity check bit computed from x_1 and x_2. (b) Posterior over codewords given that $\boldsymbol{y} = (1, 0, 0)$; the probability of a bit flip is 0.2. Generated by error_correcting_code_demo.ipynb.

5.5 Error-correcting codes (channel coding)

The idea behind **error correcting codes** is to add redundancy to a signal $\boldsymbol{x}$ (which is the result of encoding the original data), such that when it is sent over to the receiver via a noisy transmission line (such as a cell phone connection), the receiver can recover from any corruptions that might occur to the signal. This is called **channel coding**.

In more detail, let $\boldsymbol{x} \in \{0, 1\}^m$ be the source message, where m is called the **block length**. Let $\boldsymbol{y}$ be the result of sending $\boldsymbol{x}$ over a **noisy channel**. This is a corrupted version of the message. For example, each message bit may get flipped independently with probability α, in which case $p(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=1}^m p(y_i|x_i)$, where $p(y_i|x_i = 0) = [1 - \alpha, \alpha]$ and $p(y_i|x_i = 1) = [\alpha, 1 - \alpha]$. Alternatively, we may add Gaussian noise, so $p(y_i|x_i = b) = \mathcal{N}(y_i|\mu_b, \sigma^2)$. The receiver's goal is to infer the true message from the noisy observations, i.e., to compute $\text{argmax}_{\boldsymbol{x}} p(\boldsymbol{x}|\boldsymbol{y})$.

A common way to increase the chance of being able to recover the original signal is to add **parity check bits** to it before sending it. These are deterministic functions of the original signal, which specify if the sum of the input bits is odd or even. This provides a form of **redundancy**, so that if one bit is corrupted, we can still infer its value, assuming the other bits are not flipped. (This is reasonable since we assume the bits are corrupted independently at random, so it is less likely that multiple bits are flipped than just one bit.)

For example, suppose we have two original message bits, and we add one parity bit. This can be modeled using a directed graphical model as shown in Figure 5.11(a). This graph encodes the following joint probability distribution:

$$p(\boldsymbol{x}, \boldsymbol{y}) = p(x_1)p(x_2)p(x_3|x_1, x_2)\prod_{i=1}^{3} p(y_i|x_i) \tag{5.163}$$

The priors $p(x_1)$ and $p(x_2)$ are uniform. The conditional term $p(x_3|x_1, x_2)$ is deterministic, and computes the parity of (x_1, x_2). In particular, we have $p(x_3 = 1|x_1, x_2) = 1$ if the total number of 1s in the block $x_{1:2}$ is odd. The likelihood terms $p(y_i|x_i)$ represent a bit flipping noisy channel model, with noise level $\alpha = 0.2$.

Suppose we observe $\boldsymbol{y} = (1, 0, 0)$. We know that this cannot be what the sender sent, since this violates the parity constraint (if $x_1 = 1$ then we know $x_3 = 1$). Instead, the 3 posterior modes for $\boldsymbol{x}$ are 000 (first bit was flipped), 110 (second bit was flipped), and 101 (third bit was flipped). The only other configuration with non-zero support in the posterior is 011, which corresponds to the much less likely hypothesis that three bits were flipped (see Figure 5.11(b)). All other hypotheses (001, 010 and 100) are inconsistent with the deterministic method used to create codewords. (See Section 9.2.3.2 for further discussion of this point.)

In practice, we use more complex coding schemes that are more efficient, in the sense that they add less redundant bits to the message, but still guarantee that errors can be corrected. For details, see Section 9.3.8.

5.6 The information bottleneck

In this section, we discuss discriminative models $p(\boldsymbol{y}|\boldsymbol{x})$ that use a *stochastic bottleneck* between the input $\boldsymbol{x}$ and the output $\boldsymbol{y}$ to prevent overfitting, and improve robustness and calbration.

5.6.1 Vanilla IB

We say that $\boldsymbol{z}$ is a **representation** of $\boldsymbol{x}$ if $\boldsymbol{z}$ is a (possibly stochastic) function of $\boldsymbol{x}$, and hence can be described by the conditional $p(\boldsymbol{z}|\boldsymbol{x})$. We say that a representation $\boldsymbol{z}$ of $\boldsymbol{x}$ is **sufficient** for task $\boldsymbol{y}$ if $\boldsymbol{y} \perp \boldsymbol{x}|\boldsymbol{z}$, or equivalently, if $\mathbb{I}(\boldsymbol{z}; \boldsymbol{y}) = \mathbb{I}(\boldsymbol{x}; \boldsymbol{y})$, i.e., $\mathbb{H}(\boldsymbol{y}|\boldsymbol{z}) = \mathbb{H}(\boldsymbol{y}|\boldsymbol{x})$. We say that a representation is a **minimal sufficient statistic** if $\boldsymbol{z}$ is sufficient and there is no other $\boldsymbol{z}$ with smaller $\mathbb{I}(\boldsymbol{z}; \boldsymbol{x})$ value. Thus we would like to find a representation $\boldsymbol{z}$ that maximizes $\mathbb{I}(\boldsymbol{z}; \boldsymbol{y})$ while minimizing $\mathbb{I}(\boldsymbol{z}; \boldsymbol{x})$. That is, we would like to optimize the following objective:

$$\min \beta \mathbb{I}(\boldsymbol{z}; \boldsymbol{x}) - \mathbb{I}(\boldsymbol{z}; \boldsymbol{y}) \tag{5.164}$$

where $\beta \geq 0$, and we optimize wrt the distributions $p(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{y}|\boldsymbol{z})$. This is called the **information bottleneck principle** [TPB99]. This generalizes the concept of minimal sufficient statistic to take into account that there is a tradeoff between sufficiency and minimality, which is captured by the Lagrange multiplier $\beta > 0$.

This principle is illustrated in Figure 5.12. We assume Z is a function of X, but is independent of Y, i.e., we assume the graphical model $Z \leftarrow X \leftrightarrow Y$. This corresponds to the following joint distribution:

$$p(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) = p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x}) \tag{5.165}$$

Thus Z can capture any amount of information about X that it wants, but cannot contain information that is unique to Y, as illustrated in Figure 5.12a. The optimal representation only captures information about X that is useful for Y; to prevent us "wasting capacity" and fitting irrelevant details of the input, Z should also minimize information about X, as shown in Figure 5.12b.

If all the random variables are discrete, and $\boldsymbol{z} = e(\boldsymbol{x})$ is a deterministic function of $\boldsymbol{x}$, then the algorithm of [TPB99] can be used to minimize the IB objective in Section 5.6. The objective can also be solved analytically if all variables are jointly Gaussian [Che+05] (the resulting method can be viewed as a form of supervised PCA). But in general, it is intractable to solve this problem exactly. We discuss a tractable approximation in Section 5.6.2. (More details can be found in e.g., [SZ22].)

(a) (b)

Figure 5.12: Information diagrams for information bottleneck. (a) Z can contain any amount of information about X (whether it useful for predicting Y or not), but it cannot contain information about Y that is not shared with X. (b) The optimal representation for Z maximizes $\mathbb{I}(Z, Y)$ and minimizes $\mathbb{I}(Z, X)$. Used with kind permission of Katie Everett.

5.6.2 Variational IB

In this section, we derive a variational upper bound on Equation (5.164), leveraging ideas from Section 5.3.6. This is called the **variational IB** or **VIB** method [Ale+16]. The key trick will be to use the non-negativity of the KL divergence to write

$$\int d\boldsymbol{x}\; p(\boldsymbol{x}) \log p(\boldsymbol{x}) \geq \int d\boldsymbol{x}\; p(\boldsymbol{x}) \log q(\boldsymbol{x}) \tag{5.166}$$

for any distribution q. (Note that both p and q may be conditioned on other variables.)

To explain the method in more detail, let us define the following notation. Let $e(\boldsymbol{z}|\boldsymbol{x}) = p(\boldsymbol{z}|\boldsymbol{x})$ represent the encoder, $b(\boldsymbol{z}|\boldsymbol{y}) \approx p(\boldsymbol{z}|\boldsymbol{y})$ represent the backwards encoder, $d(\boldsymbol{z}|\boldsymbol{y}) \approx p(\boldsymbol{z}|\boldsymbol{y})$ represent the classifier (decoder), and $m(\boldsymbol{z}) \approx p(\boldsymbol{z})$ represent the marginal. (Note that we get to choose $p(\boldsymbol{z}|\boldsymbol{x})$, but the other distributions are derived by approximations of the corresponding marginals and conditionals of the exact joint $p(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$.) Also, let $\langle \cdot \rangle$ represent expectations wrt the relevant terms from the $p(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ joint.

With this notation, we can derive a lower bound on $\mathbb{I}(\boldsymbol{z}; \boldsymbol{y})$ as follows:

$$\mathbb{I}(\boldsymbol{z}; \boldsymbol{y}) = \int d\boldsymbol{y} d\boldsymbol{z}\; p(\boldsymbol{y}, \boldsymbol{z}) \log \frac{p(\boldsymbol{y}, \boldsymbol{z})}{p(\boldsymbol{y})p(\boldsymbol{z})} \tag{5.167}$$

$$= \int d\boldsymbol{y} d\boldsymbol{z}\; p(\boldsymbol{y}, \boldsymbol{z}) \log p(\boldsymbol{y}|\boldsymbol{z}) - \int d\boldsymbol{y} d\boldsymbol{z}\; p(\boldsymbol{y}, \boldsymbol{z}) \log p(\boldsymbol{y}) \tag{5.168}$$

$$= \int d\boldsymbol{y} d\boldsymbol{z}\; p(\boldsymbol{z}) p(\boldsymbol{y}|\boldsymbol{z}) \log p(\boldsymbol{y}|\boldsymbol{z}) - \text{const} \tag{5.169}$$

$$\geq \int d\boldsymbol{y} d\boldsymbol{z}\; p(\boldsymbol{y}, \boldsymbol{z}) \log d(\boldsymbol{y}|\boldsymbol{z}) \tag{5.170}$$

$$= \langle \log d(\boldsymbol{y}|\boldsymbol{z}) \rangle \tag{5.171}$$

where we exploited the fact that $\mathbb{H}(p(\boldsymbol{y}))$ is a constant that is independent of our representation.

Note that we can approximate the expections by sampling from

$$p(\boldsymbol{y}, \boldsymbol{z}) = \int d\boldsymbol{x}\; p(\boldsymbol{x}) p(\boldsymbol{y}|\boldsymbol{x}) p(\boldsymbol{z}|\boldsymbol{x}) = \int d\boldsymbol{x}\; p(\boldsymbol{x}, \boldsymbol{y}) e(\boldsymbol{z}|\boldsymbol{x}) \tag{5.172}$$

This is just the empirical distribution "pushed through" the encoder.

Similarly, we can derive an upper bound on $\mathbb{I}(\boldsymbol{z}; \boldsymbol{x})$ as follows:

$$\mathbb{I}(\boldsymbol{z}; \boldsymbol{x}) = \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x}, \boldsymbol{z}) \log \frac{p(\boldsymbol{z}, \boldsymbol{x})}{p(\boldsymbol{x}) p(\boldsymbol{z})} \tag{5.173}$$

$$= \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x}, \boldsymbol{z}) \log p(\boldsymbol{z}|\boldsymbol{x}) - \int d\boldsymbol{z}\; p(\boldsymbol{z}) \log p(\boldsymbol{z}) \tag{5.174}$$

$$\leq \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x}, \boldsymbol{z}) \log p(\boldsymbol{z}|\boldsymbol{x}) - \int d\boldsymbol{z}\; p(\boldsymbol{z}) \log m(\boldsymbol{z}) \tag{5.175}$$

$$= \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x}, \boldsymbol{z}) \log \frac{e(\boldsymbol{z}|\boldsymbol{x})}{m(\boldsymbol{z})} \tag{5.176}$$

$$= \langle \log e(\boldsymbol{z}|\boldsymbol{x}) \rangle - \langle m(\boldsymbol{z}) \rangle \tag{5.177}$$

Note that we can approximate the expectations by sampling from $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{x}) p(\boldsymbol{z}|\boldsymbol{x})$.

Putting it altogether, we get the following upper bound on the IB objective:

$$\beta\, \mathbb{I}(\boldsymbol{x}; \boldsymbol{z}) - \mathbb{I}(\boldsymbol{z}; \boldsymbol{y}) \leq \beta \left(\langle \log e(\boldsymbol{z}|\boldsymbol{x}) \rangle - \langle \log m(\boldsymbol{z}) \rangle \right) - \langle \log d(\boldsymbol{y}|\boldsymbol{z}) \rangle \tag{5.178}$$

Thus the VIB objective is

$$\mathcal{L}_{\text{VIB}} = \beta \left(\mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x}) e(\boldsymbol{z}|\boldsymbol{x})} \left[\log e(\boldsymbol{z}|\boldsymbol{x}) - \log m(\boldsymbol{z}) \right] \right) - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x}) e(\boldsymbol{z}|\boldsymbol{x}) d(\boldsymbol{y}|\boldsymbol{z})} \left[\log d(\boldsymbol{y}|\boldsymbol{z}) \right] \tag{5.179}$$

$$= -\mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x}) e(\boldsymbol{z}|\boldsymbol{x}) d(\boldsymbol{y}|\boldsymbol{z})} \left[\log d(\boldsymbol{y}|\boldsymbol{z}) \right] + \beta \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(e(\boldsymbol{z}|\boldsymbol{x}) \parallel m(\boldsymbol{z}) \right) \right] \tag{5.180}$$

We can now take stochastic gradients of this objective and minimize it (wrt the parameters of the encoder, decoder and marginal) using SGD. (We assume the distributions are reparameterizable, as discussed in Section 6.5.4.) For the encoder $e(\boldsymbol{z}|\boldsymbol{x})$, we often use a conditional Gaussian, and for the decoder $d(\boldsymbol{y}|\boldsymbol{z})$, we often use a softmax classifier. For the marignal, $m(\boldsymbol{z})$, we should use a flexible model, such as a mixture of Gaussians, since it needs to approximate the **aggregated posterior** $p(\boldsymbol{z}) = \int d\boldsymbol{z} p(\boldsymbol{x}) e(\boldsymbol{z}|\boldsymbol{x})$, which is a mixture of N Gaussians (assuming $p(\boldsymbol{x})$ is an empirical distribution with N samples, and $e(\boldsymbol{z}|\boldsymbol{x})$ is a Gaussian).

We illustrate this in Figure 5.13, where we fit the an MLP model to MNIST. We use a 2d bottleneck layer before passing to the softmax. In panel a, we show the embedding learned by a determinisic encoder. We see that each image gets mapped to a point, and there is little overlap between classes, or between instances. In panels b-c, we show the embedding learned by a stochastic encoder. Each image gets mapped to a Gaussian distribution, we show the mean and the covariance separately. The classes are still well separated, but individual instances of a class are no longer distinguishable, since such information is not relevant for prediction purposes.

5.6.3 Conditional entropy bottleneck

The IB tries to maximize $\mathbb{I}(Z; Y)$ while minimizing $\mathbb{I}(Z; X)$. We can write this objective as

$$\min \mathbb{I}(\boldsymbol{x}; \boldsymbol{z}) - \lambda\, \mathbb{I}(\boldsymbol{y}; \boldsymbol{z}) \tag{5.181}$$

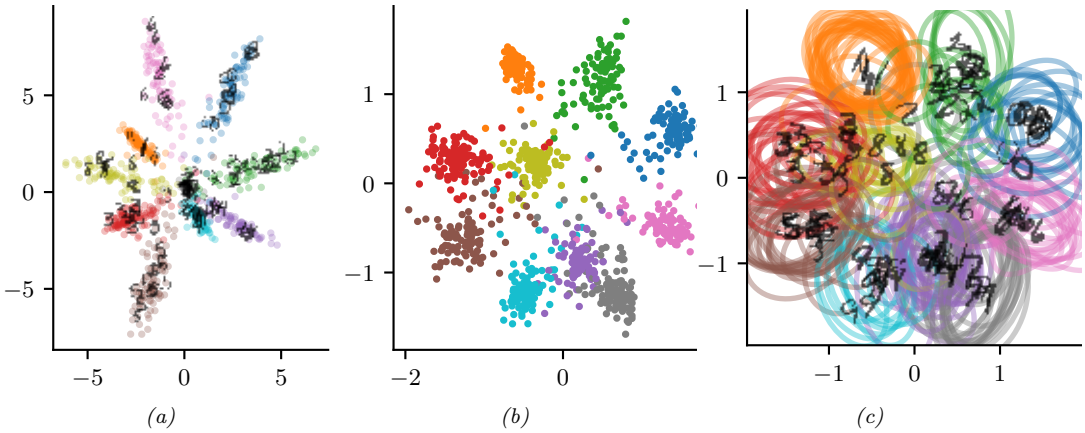


Figure 5.13: 2d embeddings of MNIST digits created by an MLP classifier. (a) Deterministic model. (b-c) VIB model, means and covariances. Generated by vib_demo.ipynb. Used with kind permission of Alex Alemi.

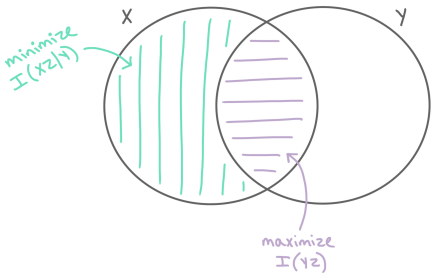


Figure 5.14: Conditional entropy bottleneck (CEB) chooses a representation Z that maximizes $\mathbb{I}(Z,Y)$ and minimizes $\mathbb{I}(X,Z|Y)$. Used with kind permission of Katie Everett.

for $\lambda \geq 0$. However, we see from the information diagram in Figure 5.12b that $\mathbb{I}(Z;X)$ contains some information that is relevant to Y. A sensible alternative objective is to minimizes the residual mutual information, $\mathbb{I}(X;Z|Y)$. This gives rise to the following objective:

$$\min \mathbb{I}(\boldsymbol{x};\boldsymbol{z}|\boldsymbol{y}) - \lambda' \mathbb{I}(\boldsymbol{y};\boldsymbol{z}) \tag{5.182}$$

for $\lambda' \geq 0$. This is known as the **conditional entropy bottleck** or **CEB** [Fis20]. See Figure 5.14 for an illustration.

Since $\mathbb{I}(\boldsymbol{x};\boldsymbol{z}|\boldsymbol{y}) = \mathbb{I}(\boldsymbol{x};\boldsymbol{z}) - \mathbb{I}(\boldsymbol{y};\boldsymbol{z})$, we see that the CEB is equivalent to standard IB with $\lambda' = \lambda + 1$. However, it is easier to upper bound $\mathbb{I}(\boldsymbol{x};\boldsymbol{z}|\boldsymbol{y})$ than $\mathbb{I}(\boldsymbol{x};\boldsymbol{z})$, since we are conditioning on $\boldsymbol{y}$, which

provides information about $\boldsymbol{z}$. In particular, we have

$$\begin{aligned}
\mathbb{I}(\boldsymbol{x};\boldsymbol{z}|\boldsymbol{y}) &= \mathbb{I}(\boldsymbol{x};\boldsymbol{z}) - \mathbb{I}(\boldsymbol{y};\boldsymbol{z}) && (5.183)\\
&= \mathbb{H}(\boldsymbol{z}) - \mathbb{H}(\boldsymbol{z}|\boldsymbol{x}) - [\mathbb{H}(\boldsymbol{z}) - \mathbb{H}(\boldsymbol{z}|\boldsymbol{y})] && (5.184)\\
&= -\mathbb{H}(\boldsymbol{z}|\boldsymbol{x}) - \mathbb{H}(\boldsymbol{z}|\boldsymbol{y}) && (5.185)\\
&= \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x},\boldsymbol{z}) \log p(\boldsymbol{z}|\boldsymbol{x}) - \int d\boldsymbol{z} d\boldsymbol{y}\; p(\boldsymbol{z},\boldsymbol{y}) \log p(\boldsymbol{z}|\boldsymbol{y}) && (5.186)\\
&\leq \int d\boldsymbol{z} d\boldsymbol{x}\; p(\boldsymbol{x},\boldsymbol{z}) \log e(\boldsymbol{z}|\boldsymbol{x}) - \int d\boldsymbol{z} d\boldsymbol{y}\; p(\boldsymbol{z},\boldsymbol{y}) \log b(\boldsymbol{z}|\boldsymbol{y}) && (5.187)\\
&= \langle \log e(\boldsymbol{z}|\boldsymbol{x}) \rangle - \langle \log b(\boldsymbol{z}|\boldsymbol{y}) \rangle && (5.188)
\end{aligned}$$

Putting it altogether, we get the final CEB objective:

$$\min \beta \left(\langle \log e(\boldsymbol{z}|\boldsymbol{x}) \rangle - \langle \log b(\boldsymbol{z}|\boldsymbol{y}) \rangle \right) - \langle \log d(\boldsymbol{y}|\boldsymbol{z}) \rangle \tag{5.189}$$

Note that it is generally easier to learn the conditional backwards encoder $b(\boldsymbol{z}|\boldsymbol{y})$ than the unconditional marginal $m(\boldsymbol{z})$. Also, we know that the tightest upper bound occurs when $\mathbb{I}(\boldsymbol{x};\boldsymbol{z}|\boldsymbol{y}) = \mathbb{I}(\boldsymbol{x};\boldsymbol{z}) - \mathbb{I}(\boldsymbol{y};\boldsymbol{z}) = 0$. The corresponding value of β corresponds to an optimal representation. By contrast, it is not clear how to measure distance from optimality when using IB.

6 Optimization

6.1 Introduction

In this chapter, we consider solving **optimization problems** of various forms. Abstractly these can all be written as

$$\boldsymbol{\theta}^* \in \operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}) \tag{6.1}$$

where $\mathcal{L} : \Theta \to \mathbb{R}$ is the objective or loss function, and Θ is the parameter space we are optimizing over. However, this abstraction hides many details, such as whether the problem is constrained or unconstrained, discrete or continuous, convex or non-convex, etc. In the prequel to this book, [Mur22], we discussed some simple optimization algorithms for some common problems that arise in machine learning. In this chapter, we discuss some more advanced methods. For more details on optimization, please consult some of the many excellent textbooks, such as [KW19b; BV04; NW06; Ber15; Ber16] as well as various review articles, such as [BCN18; Sun+19b; PPS18; Pey20].

6.2 Automatic differentiation

This section was written by Roy Frostig.

This section is concerned with computing (partial) derivatives of complicated functions in an automatic manner. By "complicated" we mean those expressed as a composition of an arbitrary number of more basic operations, such as in deep neural networks. This task is known as **automatic differentiation** (**AD**), or **autodiff**. AD is an essential component in optimization and deep learning, and is also used in several other fields across science and engineering. See e.g. Baydin et al. [Bay+15] for a review focused on machine learning and Griewank and Walther [GW08] for a classical textbook.

6.2.1 Differentiation in functional form

Before covering automatic differentiation, it is useful to review the mathematics of differentiation. We will use a particular **functional** notation for partial derivatives, rather than the typical one used throughout much of this book. We will refer to the latter as the **named variable** notation for the moment. Named variable notation relies on associating function arguments with names. For instance, given a function $f : \mathbb{R}^2 \to \mathbb{R}$, the partial derivative of f with respect to its first scalar argument, at a

point $\boldsymbol{a} = (a_1, a_2)$, might be written:

$$\left.\frac{\partial f}{\partial x_1}\right|_{\boldsymbol{x}=\boldsymbol{a}} \tag{6.2}$$

This notation is not entirely self-contained. It refers to a name $\boldsymbol{x} = (x_1, x_2)$, implicit or inferred from context, suggesting the argument of f. An alternative expression is:

$$\frac{\partial}{\partial a_1} f(a_1, a_2) \tag{6.3}$$

where now a_1 serves both as an argument name (or a symbol in an expression) and as a particular evaluation point. Tracking names can become an increasingly complicated endeavor as we compose many functions together, each possibly taking several arguments.

A functional notation instead defines derivatives as operators on functions. If a function has multiple arguments, they are identified by position rather than by name, alleviating the need for auxiliary variable definitions. Some of the following definitions draw on those in Spivak's *Calculus on Manifolds* [Spi71], in Sussman and Wisdom's *Functional Differential Geometry* [SW13], and generally appear more regularly in accounts of differential calculus and geometry. These texts are recommended for a more formal treatment, and a more mathematically general view, of the material briefly covered in this section.

Beside notation, we will rely on some basic multivariable calculus concepts. This includes the notion of (partial) derivatives, the differential or Jacobian of a function at a point, its role as a linear approximation local to the point, and various properties of linear maps, matrices, and transposition. We will focus on a finite-dimensional setting and write $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$ for the standard basis in $\mathbb{R}^n$.

Linear and multilinear functions. We use $F : \mathbb{R}^n \multimap \mathbb{R}^m$ to denote a function $F : \mathbb{R}^n \to \mathbb{R}^m$ that is linear, and by $F[\boldsymbol{x}]$ its application to $\boldsymbol{x} \in \mathbb{R}^n$. Recall that such a linear map corresponds to a matrix in $\mathbb{R}^{m \times n}$ whose columns are $F[\boldsymbol{e}_1], \ldots, F[\boldsymbol{e}_n]$; both interpretations will prove useful. Conveniently, function composition and matrix multiplication expressions look similar: to compose two linear maps F and G we can write $F \circ G$ or, barely abusing notation, consider the matrix FG. Every linear map $F : \mathbb{R}^n \multimap \mathbb{R}^m$ has a transpose $F : \mathbb{R}^m \multimap \mathbb{R}^n$, which is another linear map identified with transposing the corresponding matrix.

Repeatedly using the linear arrow symbol, we can denote by:

$$T : \underbrace{\mathbb{R}^n \multimap \cdots \multimap \mathbb{R}^n}_{k \text{ times}} \multimap \mathbb{R}^m \tag{6.4}$$

a multilinear, or more specifically k-linear, map:

$$T : \underbrace{\mathbb{R}^n \times \cdots \times \mathbb{R}^n}_{k \text{ times}} \to \mathbb{R}^m \tag{6.5}$$

which corresponds to an array (or tensor) in $\mathbb{R}^{m \times n \times \cdots \times n}$. We denote by $T[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k] \in \mathbb{R}^m$ the application of such a k-linear map to vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathbb{R}^n$.

The derivative operator. For an open set $U \subset \mathbb{R}^n$ and a differentiable function $f : U \to \mathbb{R}^m$, denote its **derivative function**:

$$\partial f : U \to (\mathbb{R}^n \multimap \mathbb{R}^m) \tag{6.6}$$

or equivalently $\partial f : U \to \mathbb{R}^{m \times n}$. This function maps a point $\boldsymbol{x} \in U$ to the Jacobian of all partial derivatives evaluated at $\boldsymbol{x}$. The symbol ∂ itself denotes the **derivative operator**, a function mapping functions to their derivative functions. When $m = 1$, the map $\partial f(\boldsymbol{x})$ recovers the standard gradient $\nabla f(\boldsymbol{x})$ at any $\boldsymbol{x} \in U$, by considering the matrix view of the former. Indeed, the nabla symbol ∇ is sometimes described as an operator as well, such that ∇f is a function. When $n = m = 1$, the Jacobian is scalar-valued, and ∂f is the familiar derivative f'.

In the expression $\partial f(\boldsymbol{x})[\boldsymbol{v}]$, we will sometimes refer to the argument $\boldsymbol{x}$ as the **linearization point** for the Jacobian, and to $\boldsymbol{v}$ as the **perturbation**. We call the map:

$$(\boldsymbol{x}, \boldsymbol{v}) \mapsto \partial f(\boldsymbol{x})[\boldsymbol{v}] \tag{6.7}$$

over linearization points $\boldsymbol{x} \in U$ and *input* perturbations $\boldsymbol{v} \in \mathbb{R}^n$ the **Jacobian-vector product (JVP)**. We similarly call its transpose:

$$(\boldsymbol{x}, \boldsymbol{u}) \mapsto \partial f(\boldsymbol{x})^\mathsf{T}[\boldsymbol{u}] \tag{6.8}$$

over linearization points $\boldsymbol{x} \in U$ and *output* perturbations $\boldsymbol{u} \in \mathbb{R}^m$ the **vector-Jacobian product (VJP)**.

Thinking about maps instead of matrices can help us define higher-order derivatives recursively, as we proceed to do below. It separately suggests how the action of a Jacobian is commonly written in code. When we consider writing $\partial f(\boldsymbol{x})$ in a program for a fixed $\boldsymbol{x}$, we often implement it as a function that carries out multiplication by the Jacobian matrix, i.e. $\boldsymbol{v} \mapsto \partial f(\boldsymbol{x})[\boldsymbol{v}]$, instead of explicitly representing it as a matrix of numbers in memory. Going a step further, for that matter, we often implement ∂f as an entire JVP at once, i.e. over any linearization point $\boldsymbol{x}$ and perturbation $\boldsymbol{v}$. As a toy example with scalars, consider the cosine:

$$(x, v) \mapsto \partial \cos(x) v = -v \sin(x) \tag{6.9}$$

If we express this at once in code, we can, say, avoid computing $\sin(x)$ whenever $v = 0$.[1]

Higher-order derivatives. Suppose the function f above remains arbitrarily differentiable over its domain $U \subset \mathbb{R}^n$. To take another derivative, we write:

$$\partial^2 f : U \to (\mathbb{R}^n \multimap \mathbb{R}^n \multimap \mathbb{R}^m) \tag{6.10}$$

where $\partial^2 f(\boldsymbol{x})$ is a bilinear map representing all second-order partial derivatives. In named variable notation, one might write $\frac{\partial f(\boldsymbol{x})}{\partial x_i \partial x_j}$ to refer to $\partial^2 f(\boldsymbol{x})[\boldsymbol{e}_i, \boldsymbol{e}_j]$, for example.

1. This example ignores that such an optimization might be done (best) by a compiler. Then again, for more complex examples, implementing $(\boldsymbol{x}, \boldsymbol{v}) \mapsto \partial f(\boldsymbol{x})[\boldsymbol{v}]$ as a single subroutine can help guide compiler optimizations all the same.

The second derivative function $\partial^2 f$ can be treated coherently as the outcome of applying the derivative operator twice. That is, it makes sense to say that $\partial^2 = \partial \circ \partial$. This observation extends recursively to cover arbitrary higher-order derivatives. For $k \geq 1$:

$$\partial^k f : U \to (\underbrace{\mathbb{R}^n \multimap \ldots \multimap \mathbb{R}^n}_{k \text{ times}} \multimap \mathbb{R}^m) \tag{6.11}$$

is such that $\partial^k f(\boldsymbol{x})$ is a k-linear map.

With $m = 1$, the map $\partial^2 f(\boldsymbol{x})$ corresponds to the Hessian matrix at any $\boldsymbol{x} \in U$. Although Jacobians and Hessians suffice to make sense of many machine learning techniques, arbitrary higher-order derivatives are not hard to come by either (e.g. [Kel+20]). As an example, they appear when writing down something as basic as a function's Taylor series approximation, which we can express with our derivative operator as:

$$f(\boldsymbol{x} + \boldsymbol{v}) \approx f(\boldsymbol{x}) + \partial f(\boldsymbol{x})[\boldsymbol{v}] + \frac{1}{2!}\partial^2 f(\boldsymbol{x})[\boldsymbol{v}, \boldsymbol{v}] + \cdots + \frac{1}{k!}\partial^k f(\boldsymbol{x})[\boldsymbol{v}, \ldots, \boldsymbol{v}] \tag{6.12}$$

Multiple inputs. Now consider a function of two arguments:

$$g : U \times V \to \mathbb{R}^m \,. \tag{6.13}$$

where $U \subset \mathbb{R}^{n_1}$ and $V \subset \mathbb{R}^{n_2}$. For our purposes, a product domain like $U \times V$ mainly serves to suggest a convenient partitioning of a function's input components. It is isomorphic to a subset of $\mathbb{R}^{n_1+n_2}$, corresponding to a single-input function. The latter tells us how the derivative functions of g ought to look, based on previous definitions, and we will swap between the two views with little warning. Multiple inputs tend to arise in the context of computational circuits and programs: many functions in code are written to accept multiple arguments, and many basic operations (such as $+$) do the same.

With multiple inputs, we can denote by $\partial_i g$ the derivative function with respect to the i'th argument:

$$\partial_1 g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \to (\mathbb{R}^{n_1} \multimap \mathbb{R}^m)\,, \text{ and} \tag{6.14}$$

$$\partial_2 g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \to (\mathbb{R}^{n_2} \multimap \mathbb{R}^m)\,. \tag{6.15}$$

Under the matrix view, the function $\partial_1 g$ maps a pair of points $\boldsymbol{x} \in \mathbb{R}^{n_1}$ and $\boldsymbol{y} \in \mathbb{R}^{n_2}$ to the matrix of all partial derivatives of g with respect to its first argument, evaluated at $(\boldsymbol{x}, \boldsymbol{y})$. We take ∂g with no subscript to simply mean the concatenation of $\partial_1 g$ and $\partial_2 g$:

$$\partial g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \to (\mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \multimap \mathbb{R}^m) \tag{6.16}$$

where, for every linearization point $(\boldsymbol{x}, \boldsymbol{y}) \in U \times V$ and perturbations $\dot{\boldsymbol{x}} \in \mathbb{R}^{n_1}$, $\dot{\boldsymbol{y}} \in \mathbb{R}^{n_2}$:

$$\partial g(\boldsymbol{x}, \boldsymbol{y})[\dot{\boldsymbol{x}}, \dot{\boldsymbol{y}}] = \partial_1 g(\boldsymbol{x}, \boldsymbol{y})[\dot{\boldsymbol{x}}] + \partial_2 g(\boldsymbol{x}, \boldsymbol{y})[\dot{\boldsymbol{y}}]\,. \tag{6.17}$$

Alternatively, taking the matrix view:

$$\partial g(\boldsymbol{x}, \boldsymbol{y}) = \begin{pmatrix} \partial_1 g(\boldsymbol{x}, \boldsymbol{y}) & \partial_2 g(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix}\,. \tag{6.18}$$

This convention will simplify our chain rule statement below. When $n_1 = n_2 = m = 1$, both sub-matrices are scalar, and $\partial g_1(x, y)$ recovers the partial derivative that might otherwise be written in named variable notation as:

$$\frac{\partial}{\partial x} g(x, y)\,. \tag{6.19}$$

However, the expression ∂g_1 bears a meaning on its own (as a function) whereas the expression $\frac{\partial g}{\partial x}$ may be ambiguous without further context. Again composing operators lets us write higher-order derivatives. For instance, $\partial_2 \partial_1 g(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^{m \times n_1 \times n_2}$, and if $m = 1$, the Hessian of g at $(\boldsymbol{x}, \boldsymbol{y})$ is:

$$\begin{pmatrix} \partial_1 \partial_1 g(\boldsymbol{x}, \boldsymbol{y}) & \partial_1 \partial_2 g(\boldsymbol{x}, \boldsymbol{y}) \\ \partial_2 \partial_1 g(\boldsymbol{x}, \boldsymbol{y}) & \partial_2 \partial_2 g(\boldsymbol{x}, \boldsymbol{y}) \end{pmatrix}\,. \tag{6.20}$$

Composition and fan-out. If $f = g \circ h$ for some $h : \mathbb{R}^n \to \mathbb{R}^p$ and $g : \mathbb{R}^p \to \mathbb{R}^m$, then the **chain rule** of calculus observes that:

$$\partial f(\boldsymbol{x}) = \partial g(h(\boldsymbol{x})) \circ \partial h(\boldsymbol{x}) \;\text{ for all } \boldsymbol{x} \in \mathbb{R}^n \tag{6.21}$$

How does this interact with our notation for multi-argument functions? For one, it can lead us to consider expressions with **fan-out**, where several sub-expressions are functions of the same input. For instance, assume two functions $a : \mathbb{R}^n \to \mathbb{R}^{m_1}$ and $b : \mathbb{R}^n \to \mathbb{R}^{m_2}$, and that:

$$f(\boldsymbol{x}) = g(a(\boldsymbol{x}), b(\boldsymbol{x})) \tag{6.22}$$

for some function g. Abbreviating $h(\boldsymbol{x}) = (a(\boldsymbol{x}), b(\boldsymbol{x}))$ so that $f(\boldsymbol{x}) = g(h(\boldsymbol{x}))$, Equations (6.16) and (6.21) tell us that:

$$\partial f(\boldsymbol{x}) = \partial g(h(\boldsymbol{x})) \circ \partial h(\boldsymbol{x}) \tag{6.23}$$

$$= \partial_1 g(a(\boldsymbol{x}), b(\boldsymbol{x})) \circ \partial a(\boldsymbol{x}) + \partial_2 g(a(\boldsymbol{x}), b(\boldsymbol{x})) \circ \partial b(\boldsymbol{x}) \tag{6.24}$$

Note that $+$ is meant pointwise here. It also follows from the above that if instead:

$$f(\boldsymbol{x}, \boldsymbol{y}) = g(a(\boldsymbol{x}), b(\boldsymbol{y})) \tag{6.25}$$

in other words, if we write multiple arguments but exhibit no fan-out, then:

$$\partial_1 f(\boldsymbol{x}, \boldsymbol{y}) = \partial_1 g(a(\boldsymbol{x}), b(\boldsymbol{y})) \circ \partial a(\boldsymbol{x})\,, \text{ and} \tag{6.26}$$

$$\partial_2 f(\boldsymbol{x}, \boldsymbol{y}) = \partial_2 g(a(\boldsymbol{x}), b(\boldsymbol{y})) \circ \partial b(\boldsymbol{y}) \tag{6.27}$$

Composition and fan-out rules for derivatives are what let us break down a complex derivative calculation into simpler ones. This is what automatic differentiation techniques rely on when processing the sort of elaborate numerical computations that turn up in modern machine learning and numerical programming.

6.2.2 Differentiating chains, circuits, and programs

The purpose of automatic differentiation is to compute derivatives of arbitrary functions provided as input. Given a function $f : U \subset \mathbb{R}^n \to \mathbb{R}^m$ and a linearization point $\boldsymbol{x} \in U$, AD computes either:

- the JVP $\partial f(\boldsymbol{x})[v]$ for an input perturbation $\boldsymbol{v} \in \mathbb{R}^n$, or
- the VJP $\partial f(\boldsymbol{x})^\mathsf{T}[u]$ for an output perturbation $\boldsymbol{u} \in \mathbb{R}^m$.

In other words, JVPs and VJPs capture the two essential tasks of AD.[2]

Deciding what functions f to handle as input, and how to represent them, is perhaps the most load-bearding aspect of this setup. Over what *language* of functions should we operate? By a language, we mean some formal way of describing functions by composing a set of basic primitive operations. For primitives, we can think of various differentiable array operations (elementwise arithmetic, reductions, contractions, indexing and slicing, concatenation, etc.), but we will largely consider primitives and their derivatives as a given, and focus on how elaborately we can compose them. AD becomes increasingly challenging with increasingly expressive languages. Considering this, we introduce it in stages.

6.2.2.1 Chain compositions and the chain rule

To start, take only functions that are **chain compositions** of basic operations. Chains are a convenient class of function representations because derivatives *decompose* along the same structure according to the aptly-named chain rule.

As a toy example, consider $f : \mathbb{R}^n \to \mathbb{R}^m$ composed of three operations in sequence:

$$f = c \circ b \circ a \tag{6.28}$$

By the chain rule, its derivatives are given by

$$\partial f(\boldsymbol{x}) = \partial c(b(a(\boldsymbol{x}))) \circ \partial b(a(\boldsymbol{x})) \circ \partial a(\boldsymbol{x}) \tag{6.29}$$

Now consider the JVP against an input perturbation $\boldsymbol{v} \in \mathbb{R}^n$:

$$\partial f(\boldsymbol{x})[\boldsymbol{v}] = \partial c(b(a(\boldsymbol{x}))) \left[\partial b(a(\boldsymbol{x})) \left[\partial a(\boldsymbol{x})[\boldsymbol{v}]\right]\right] \tag{6.30}$$

This expression's bracketing highlights a right-to-left evaluation order that corresponds to **forward-mode automatic differentiation**. Namely, to carry out this JVP, it makes sense to compute prefixes of the original chain:

$$\boldsymbol{x},\ a(\boldsymbol{x}),\ b(a(\boldsymbol{x})) \tag{6.31}$$

alongside the partial JVPs, because each is then immediately used as a subsequent linearization point, respectively:

$$\partial a(\underline{\boldsymbol{x}}),\ \partial b(\underline{a(\boldsymbol{x})}),\ \partial c(\underline{b(a(\boldsymbol{x}))}) \tag{6.32}$$

Extending this idea to arbitrary chain compositions gives Algorithm 1.

2. Materalizing the Jacobian as a numerical array, as is commonly required in an optimization context, is a special case of computing a JVP or VJP against the standard basis vectors in $\mathbb{R}^n$ or $\mathbb{R}^m$ respectively.

Algorithm 1: Forward-mode automatic differentiation (JVP) on chains

input: $f : \mathbb{R}^n \to \mathbb{R}^m$ as a chain composition $f = f_T \circ \cdots \circ f_1$
input: linearization point $\boldsymbol{x} \in \mathbb{R}^n$ and input perturbation $\boldsymbol{v} \in \mathbb{R}^n$
$\boldsymbol{x}_0, \boldsymbol{v}_0 := \boldsymbol{x}, \boldsymbol{v}$
for $t := 1, \ldots, T$ **do**
  $\boldsymbol{x}_t := f_t(\boldsymbol{x}_{t-1})$
  $\boldsymbol{v}_t := \partial f_t(\boldsymbol{x}_{t-1})[\boldsymbol{v}_{t-1}]$
output: $\boldsymbol{x}_T$, equal to $f(\boldsymbol{x})$
output: $\boldsymbol{v}_T$, equal to $\partial f(\boldsymbol{x})[\boldsymbol{v}]$

By contrast, we can transpose Equation (6.29) to consider a VJP against an output perturbation $\boldsymbol{u} \in \mathbb{R}^m$:

$$\partial f(\boldsymbol{x})^\mathsf{T}[\boldsymbol{u}] = \partial a(\boldsymbol{x})^\mathsf{T}\left[\partial b(a(\boldsymbol{x}))^\mathsf{T}\left[\partial c(b(a(\boldsymbol{x})))^\mathsf{T}[\boldsymbol{u}]\right]\right] \tag{6.33}$$

Transposition reverses the Jacobian maps relative to their order in Equation (6.29), and now the bracketed evaluation corresponds to **reverse-mode automatic differentiation**. To carry out this VJP, we can compute the original chain prefixes $\boldsymbol{x}$, $a(\boldsymbol{x})$, and $b(a(\boldsymbol{x}))$ first, and then read them *in reverse* as successive linearization points:

$$\partial c(\underline{b(a(\boldsymbol{x}))})^\mathsf{T},\ \partial b(\underline{a(\boldsymbol{x})})^\mathsf{T},\ \partial a(\underline{\boldsymbol{x}})^\mathsf{T} \tag{6.34}$$

Extending this idea to arbitrary chain compositions gives Algorithm 2.

Algorithm 2: Reverse-mode automatic differentiation (VJP) on chains

input: $f : \mathbb{R}^n \to \mathbb{R}^m$ as a chain composition $f = f_T \circ \cdots \circ f_1$
input: linearization point $\boldsymbol{x} \in \mathbb{R}^n$ and output perturbation $\boldsymbol{u} \in \mathbb{R}^n$
$\boldsymbol{x}_0 := \boldsymbol{x}$
for $t := 1, \ldots, T$ **do**
  $\boldsymbol{x}_t := f_t(\boldsymbol{x}_{t-1})$
$\boldsymbol{u}_{T+1} := \boldsymbol{u}$
for $t := T, \ldots, 1$ **do**
  $\boldsymbol{u}_t := \partial f_t(\boldsymbol{x}_{t-1})^\mathsf{T}[\boldsymbol{u}_{t+1}]$
output: $\boldsymbol{x}_T$, equal to $f(\boldsymbol{x})$
output: $\boldsymbol{u}_1$, equal to $\partial f(\boldsymbol{x})^\mathsf{T}[\boldsymbol{u}]$

Although chain compositions impose a very specific structure, they already capture some deep neural network models, such as multi-layer perceptrons (provided matrix multiplication is a primitive operation), as covered in this book's prequel [Mur22, Ch.13].

Reverse-mode AD is faster than forward-mode when the output is scalar valued (as often arises in deep learning, where the output is a loss function). However, reverse-mode AD stores all chain

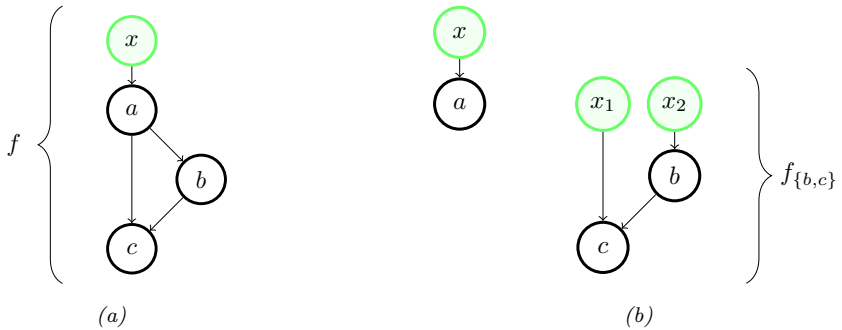


Figure 6.1: A circuit for a function f over three primitives, and its decomposition into two circuits without fan-out. Input nodes are drawn in green.

prefixes before its backward traversal, so it consumes more memory than forward-mode. There are ways to combat this memory requirement in special-case scenarios, such as when the chained operations are each reversible [MDA15; Gom+17; KKL20]. One can also trade off memory for computation by discarding some prefixes and re-computing them as needed.

6.2.2.2 From chains to circuits

When primitives can accept multiple inputs, we can naturally extend chains to **circuits**—directed acyclic graphs over primitive operations, sometimes also called computation graphs. To set up for this section, we will distinguish between (i) **input nodes** of a circuit, which symbolize a function's arguments, and (ii) **primitive nodes**, each of which is labeled by a primitive operation. We assume that input nodes have no incoming edges and (without loss of generality) exactly one outgoing edge each, and that the graph has exactly one sink node. The overall function of the circuit is composition of operations from the input nodes to the sink, where the output of each operation is input to others according to its outgoing edges.

What made AD work in Section 6.2.2.1 is the fact that derivatives decompose along chains thanks to the aptly-named chain rule. When moving from chains to directed acyclic graphs, do we need some sort of "graph rule" in order to decompose our calculation along the circuit's structure? Circuits introduce two new features: **fan-in** and **fan-out**. In graphical terms, fan-in simply refers to multiple edges incoming to a node, and fan-out refers to multiple edges outgoing.

What do these mean in functional terms? Fan-in happens when a primitive operation accepts multiple arguments. We observed in Section 6.2.1 that multiple arguments can be treated as one, and how the chain rule then applies. Fan-out requires slightly more care, specifically for reverse-mode differentiation.

The gist of an answer can be illustrated with a small example. Consider the circuit in Figure 6.1a. The operation a precedes b and c topologically, with an outgoing edge to each of both. We can cut a away from $\{b, c\}$ to produce two new circuits, shown in Figure 6.1b. The first corresponds to a and the second corresponds to the remaining computation, given by:

$$f_{\{b,c\}}(\boldsymbol{x}_1, \boldsymbol{x}_2) = c(\boldsymbol{x}_1, b(\boldsymbol{x}_2))\,. \tag{6.35}$$

We can recover the complete function f from a and $f_{\{b,c\}}$ with the help of a function dup given by:

$$\text{dup}(\boldsymbol{x}) = (\boldsymbol{x}, \boldsymbol{x}) \equiv \begin{pmatrix} I \\ I \end{pmatrix} \boldsymbol{x} \tag{6.36}$$

so that f can be written as a chain composition:

$$f = f_{\{b,c\}} \circ \text{dup} \circ a\,. \tag{6.37}$$

The circuit for $f_{\{b,c\}}$ contains no fan-out, and composition rules such as Equation (6.25) tell us its derivatives in terms of b, c, and their derivatives, all via the chain rule. Meanwhile, the chain rule applied to Equation (6.37) says that:

$$\partial f(\boldsymbol{x}) = \partial f_{\{b,c\}}(\text{dup}(a(\boldsymbol{x}))) \circ \partial\,\text{dup}(a(\boldsymbol{x})) \circ \partial a(\boldsymbol{x}) \tag{6.38}$$

$$= \partial f_{\{b,c\}}(a(\boldsymbol{x}), a(\boldsymbol{x})) \circ \begin{pmatrix} I \\ I \end{pmatrix} \circ \partial a(\boldsymbol{x})\,. \tag{6.39}$$

The above expression suggests calculating a JVP of f by right-to-left evaluation. It is similar to the JVP calculation suggested by Equation (6.30), but with a *duplication* operation $\begin{pmatrix} I & I \end{pmatrix}^\mathsf{T}$ in the middle that arises from the Jacobian of dup.

Transposing the derivative of f at $\boldsymbol{x}$:

$$\partial f(\boldsymbol{x})^\mathsf{T} = \partial a(\boldsymbol{x})^\mathsf{T} \circ \begin{pmatrix} I & I \end{pmatrix} \circ \partial f_{\{b,c\}}(a(\boldsymbol{x}), a(\boldsymbol{x}))^\mathsf{T}\,. \tag{6.40}$$

Considering right-to-left evaluation, this too is similar to the VJP calculation suggested by Equation (6.33), but with a *summation* operation $\begin{pmatrix} I & I \end{pmatrix}$ in the middle that arises from the *transposed* Jacobian of dup. The lesson of using dup in this small example is that, more generally, in order to handle fan-out in reverse mode AD, we can process operations in topological order—first forward and then in reverse—and then *sum* partial VJPs along multiple outgoing edges.

Algorithm 3: Foward-mode circuit differentiation (JVP)

input: $f : \mathbb{R}^n \to \mathbb{R}^m$ composing $f_1, \ldots, f_T$ in topological order, where f_1 is identity
input: linearization point $\boldsymbol{x} \in \mathbb{R}^n$ and perturbation $\boldsymbol{v} \in \mathbb{R}^n$
$\boldsymbol{x}_1, \boldsymbol{v}_1 := \boldsymbol{x}, \boldsymbol{v}$
for $t := 2, \ldots, T$ **do**
  let $[q_1, \ldots, q_r] = \text{Pa}(t)$
  $\boldsymbol{x}_t := f_t(\boldsymbol{x}_{q_1}, \ldots, \boldsymbol{x}_{q_r})$
  $\boldsymbol{v}_t := \sum_{i=1}^r \partial_i f_t(\boldsymbol{x}_{q_1}, \ldots, \boldsymbol{x}_{q_r})[\boldsymbol{v}_{q_i}]$
output: $\boldsymbol{x}_T$, equal to $f(\boldsymbol{x})$
output: $\boldsymbol{v}_T$, equal to $\partial f(\boldsymbol{x})[\boldsymbol{v}]$

Algorithms 3 and 4 give a complete description of forward- and reverse-mode differentiation on circuits. For brevity they assume a single argument to the entire circuit function. Nodes are indexed $1, \ldots, T$. The first is the input node associated and the remaining $T-1$ are labeled by their operation $f_2, \ldots, f_T$. We take f_1 to be the identity. For each t, if f_t takes k arguments, let $\text{Pa}(t)$ be the ordered

Algorithm 4: Reverse-mode circuit differentiation (VJP)

input: $f : \mathbb{R}^n \to \mathbb{R}^m$ composing $f_1, \ldots, f_T$ in topological order, where f_1, f_T are identity
input: linearization point $\boldsymbol{x} \in \mathbb{R}^n$ and perturbation $\boldsymbol{u} \in \mathbb{R}^m$
$\boldsymbol{x}_1 := \boldsymbol{x}$
for $t := 2, \ldots, T$ **do**
  let $[q_1, \ldots, q_r] = \mathrm{Pa}(t)$
  $\boldsymbol{x}_t := f_t(\boldsymbol{x}_{q_1}, \ldots, \boldsymbol{x}_{q_r})$
$\boldsymbol{u}_{(T-1)\to T} := \boldsymbol{u}$
for $t := T-1, \ldots, 2$ **do**
  let $[q_1, \ldots, q_r] = \mathrm{Pa}(t)$
  $\boldsymbol{u}'_t := \sum_{c \in \mathrm{Ch}(t)} \boldsymbol{u}_{t\to c}$
  $\boldsymbol{u}_{q_i \to t} := \partial_i f_t(\boldsymbol{x}_{q_1}, \ldots, \boldsymbol{x}_{q_r})^\mathsf{T} \boldsymbol{u}'_t \quad \text{for } i = 1, \ldots, r$
output: $\boldsymbol{x}_T$, equal to $f(\boldsymbol{x})$
output: $\boldsymbol{u}_{1\to 2}$, equal to $\partial f(\boldsymbol{x})^\mathsf{T} \boldsymbol{u}$

list of k indices of its parent nodes (possibly containing duplicates, due to fan-out), and let $\mathrm{Ch}(t)$ be the indices of its children (again possibly duplicate). Algorithm 4 takes a few more conventions: that f_T is the identity, that node T has $T-1$ as its only parent, and that the child of node 1 is node 2.

Fan-out is a feature of *graphs*, but arguably not an essential feature of *functions*. One can always remove all fan-out from a circuit representation by duplicating nodes. Our interest in fan-out is precisely to avoid this, allowing for an efficient representation and, in turn, efficient memory use in Algorithms 3 and 4.

Reverse-mode AD on circuits has appeared under various names and formulations over the years. The algorithm is precisely the **backpropagation** algorithm in neural networks, a term introduced in the 1980s [RHW86b; RHW86a], and has separately come up in the context of control theory and sensitivity, as summarized in historical notes by Goodfellow, Bengio, and Courville [GBC16, Section 6.6].

6.2.2.3 From circuits to programs

Graphs are useful for introducing AD algorithms, and they might align well enough with neural network applications. But computer scientists have spent decades formalizing and studying various "languages for expressing functions compositionally." Simply put, this is what programming languages are for! Can we automatically differentiate numerical functions expressed in, say, Python, Haskell, or some variant of the lambda calculus? These offer a far more widespread—and intuitively more expressive—way to describe an input function.[3]

In the previous sections, our approach to AD became more complex as we allowed for more complex graph structure. Something similar happens when we introduce grammatical constructs in a programming language. How do we adapt AD to handle a language with loops, conditionals, and

3. In Python, what the language calls a "function" does not always describe a pure function of the arguments listed in its syntactic definition; its behavior may rely on side effects or global state, as allowed by the language. Here, we specifically mean a Python function that is pure and functional. JAX's documentation details this restriction [Bra+18].

recursive calls? What about parallel programming constructs? We have partial answers to questions like these today, although they invite a deeper dive into language details such as type systems and implementation concerns [Yu+18; Inn20; Pas+21b].

One example language construct that we already know how to handle, due to Section 6.2.2.2, is a standard `let` expression. In languages with a means of name or variable binding, multiple appearances of the same variable are analogous to fan-out in a circuit. Figure 6.1a corresponds to a function f that we could write in a functional language as:

```
f(x) =
  let ax = a(x)
   in c(ax, b(ax))
```

in which `ax` indeed appears twice after it is bound.

Understanding the interaction between language capacity and automatic differentiability is an ongoing topic of computer science research [PS08a; AP19; Vyt+19; BMP19; MP21]. In the meantime, functional languages have proven quite effective in recent AD systems, both widely-used and experimental. Systems such as JAX, Dex, and others are designed around pure functional programming models, and internally rely on functional program representations for differentiation [Mac+15; BPS16; Sha+19; FJL18; Bra+18; Mac+19; Dex; Fro+21; Pas+21a].

6.3 Stochastic gradient descent

In this section, we consider optimizers for unconstrained differentiable objectives. We consider gradient-based solvers which perform iterative updates of the following form

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{C}_t \boldsymbol{g}_t \tag{6.41}$$

where $\boldsymbol{g}_t = \nabla \mathcal{L}(\boldsymbol{\theta}_t)$ is the gradient of the loss, and $\mathbf{C}_t$ is an optional **conditioning matrix**.

If we set $\mathbf{C}_t = \mathbf{I}$, the method is known as **steepest descent** or **gradient descent**. If we set $\mathbf{C}_t = \mathbf{H}_t^{-1}$, where $\mathbf{H}_t = \nabla^2 \mathcal{L}(\boldsymbol{\theta}_t)$ is the Hessian, we get **Newton's method**. There are many variants of Newton's method that are either more numerically stable, or more computationally efficient, or both.

In many problems, we cannot compute the exact gradient, either because the loss is stochastic (e.g., due to random factors in the environment), or because we approximate the loss by randomly subsampling the data. In such cases, we can modify the update in Equation (6.41) to use an unbiased approximation of the gradient. For example, suppose the loss is a finite-sum objective from a supervised learning problem:

$$\mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N} \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t)) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{\theta}_t) \tag{6.42}$$

We can approximate the gradient using a minibatch $\mathcal{B}_t$ of size $B = |\mathcal{B}_t|$ as follows:

$$\hat{\boldsymbol{g}}_t = \hat{\nabla} \mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{B} \sum_{n \in \mathcal{B}_t} \nabla \mathcal{L}_n(\boldsymbol{\theta}_t) \tag{6.43}$$

Since the minibatches are randomly sampled, this is a stochastic, but unbiased, estimate of the gradient. If we insert $\hat{\boldsymbol{g}}_t$ into Equation (6.41), the method is called **stochastic gradient descent** or **SGD**.

(a) *(b)*

Figure 6.2: Changing the mean of a Gaussian by a fixed amount (from solid to dotted curve) can have more impact when the (shared) variance is small (as in a) compared to when the variance is large (as in b). Hence the impact (in terms of prediction accuracy) of a change to μ depends on where the optimizer is in (μ, σ) space. From Figure 3 of [Hon+10], reproduced from [Val00]. Used with kind permission of Antti Honkela.

6.4 Natural gradient descent

In this section, we discuss **natural gradient descent** (**NGD**) [Ama98], which is a second order method for optimizing the parameters of (conditional) probability distributions $p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$. The key idea is to compute parameter updates by measuring distances between the induced distributions, rather than comparing parameter values directly.

For example, consider comparing two Gaussians, $p_{\boldsymbol{\theta}} = p(y|\mu, \sigma)$ and $p_{\boldsymbol{\theta}'} = p(y|\mu', \sigma')$. The (squared) Euclidean distance between the parameter vectors decomposes as $||\boldsymbol{\theta} - \boldsymbol{\theta}'||^2 = (\mu - \mu')^2 + (\sigma - \sigma')^2$. However, the predictive distribution has the form $\exp(-\frac{1}{2\sigma^2}(y - \mu)^2)$, so changes in μ need to be measured relative to σ. This is illustrated in Figure 6.2(a-b), which shows two univariate Gaussian distributions (dotted and solid lines) whose means differ by δ. In Figure 6.2(a), they share the same small variance σ^2, whereas in Figure 6.2(b), they share the same large variance. It is clear that the value of δ matters much more (in terms of the effect on the distribution) when the variance is small. Thus we see that the two parameters interact with each other, which the Euclidean distance cannot capture. This problem gets much worse when we consider more complex models, such as deep neural networks. By modeling such correlations, NGD can converge much faster than other gradient methods.

6.4.1 Defining the natural gradient

The key to NGD is to measure the notion of distance between two probability distributions in terms of the KL divergence. As we show in Section 2.4.4, this can be appproximated in terms of the Fisher information matrix (FIM). In particular, for any given input $\boldsymbol{x}$, we have

$$D_{\mathbb{KL}}\left(p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}+\boldsymbol{\delta}}(\boldsymbol{y}|\boldsymbol{x})\right) \approx \frac{1}{2}\boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_{\boldsymbol{x}}\boldsymbol{\delta} \tag{6.44}$$

where $\mathbf{F}_{\boldsymbol{x}}$ is the FIM

$$\mathbf{F}_{\boldsymbol{x}}(\boldsymbol{\theta}) = -\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[\nabla^2 \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})\right] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))^{\mathsf{T}}\right] \tag{6.45}$$

We can compute the average KL between the current and updated distributions using $\frac{1}{2}\boldsymbol{\delta}^\mathsf{T}\mathbf{F}\boldsymbol{\delta}$, where $\mathbf{F}$ is the averaged FIM:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{p_\mathcal{D}(\boldsymbol{x})}\left[\mathbf{F}_{\boldsymbol{x}}(\boldsymbol{\theta})\right] \tag{6.46}$$

NGD uses the inverse FIM as a preconditioning matrix, i.e., we perform updates of the following form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{F}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{g}_t \tag{6.47}$$

The term

$$\mathbf{F}^{-1}\boldsymbol{g}_t = \mathbf{F}^{-1}\nabla\mathcal{L}(\boldsymbol{\theta}_t) \triangleq \tilde{\nabla}\mathcal{L}(\boldsymbol{\theta}_t) \tag{6.48}$$

is called the **natural gradient**.

6.4.2 Interpretations of NGD

6.4.2.1 NGD as a trust region method

In Supplementary Section 6.1.3.1 we show that we can interpret standard gradient descent as optimizing a linear approximation to the objective subject to a penalty on the ℓ_2 norm of the change in parameters, i.e., if $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\delta}$, then we optimize

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}_t^\mathsf{T}\boldsymbol{\delta} + \eta||\boldsymbol{\delta}||_2^2 \tag{6.49}$$

Now let us replace the squared distance with the squared FIM-based distance, $||\boldsymbol{\delta}||_F^2 = \boldsymbol{\delta}^\mathsf{T}\mathbf{F}\boldsymbol{\delta}$. This is equivalent to squared Euclidean distance in the **whitened coordinate system** $\boldsymbol{\phi} = \mathbf{F}^{\frac{1}{2}}\boldsymbol{\theta}$, since

$$||\boldsymbol{\phi}_{t+1} - \boldsymbol{\phi}_t||_2^2 = ||\mathbf{F}^{\frac{1}{2}}(\boldsymbol{\theta}_t + \boldsymbol{\delta}) - \mathbf{F}^{\frac{1}{2}}\boldsymbol{\theta}_t||_2^2 = ||\mathbf{F}^{\frac{1}{2}}\boldsymbol{\delta}||_2^2 = ||\boldsymbol{\delta}||_\mathbf{F}^2 \tag{6.50}$$

The new objective becomes

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}_t^\mathsf{T}\boldsymbol{\delta} + \eta\boldsymbol{\delta}^\mathsf{T}\mathbf{F}\boldsymbol{\delta} \tag{6.51}$$

Solving $\nabla_{\boldsymbol{\delta}} M_t(\boldsymbol{\delta}) = \mathbf{0}$ gives the update

$$\boldsymbol{\delta}_t = -\eta\mathbf{F}^{-1}\boldsymbol{g}_t \tag{6.52}$$

This is the same as the natural gradient direction. Thus we can view NGD as a trust region method, where we use a first-order approximation to the objective, and use FIM-distance in the constraint.

In the above derivation, we assumed $\mathbf{F}$ was a constant matrix. Im most problems, it will change at each point in space, since we are optimizing in a curved space known as a **Riemannian manifold**. For certain models, we can compute the FIM efficiently, allowing us to capture curvature information, even though we use a first-order approximation to the objective.

6.4.2.2 NGD as a Gauss-Newton method

If $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$ is an exponential family distribution with natural parameters computed by $\boldsymbol{\eta} = f(\boldsymbol{x}, \boldsymbol{\theta})$, then one can show [Hes00; PB14] that NGD is identical to the generalized Gauss-Newton (GGN) method (Section 17.3.2). Furthermore, in the online setting, these methods are equivalent to performing sequential Bayesian inference using the extended Kalman filter, as shown in [Oll18].

6.4.3 Benefits of NGD

The use of the FIM as a preconditioning matrix, rather than the Hessian, has two advantages. First, $\mathbf{F}$ is always positive definite, whereas $\mathbf{H}$ can have negative eigenvalues at saddle points, which are prevalent in high dimensional spaces. Second, it is easy to approximate $\mathbf{F}$ online from minibatches, since it is an expectation (wrt the empirical distribution) of outer products of gradient vectors. This is in contrast to Hessian-based methods [Byr+16; Liu+18a], which are much more sensitive to noise introduced by the minibatch approximation.

In addition, the connection with trust region optimization makes it clear that NGD updates parameters in a way that matter most for prediction, which allows the method to take larger steps in uninformative regions of parameter space, which can help avoid getting stuck on plateaus. This can also help with issues that arise when the parameters are highly correlated.

For example, consider a 2d Gaussian with an unusual, highly coupled parameterization, proposed in [SD12]:

$$p(\boldsymbol{x};\boldsymbol{\theta}) = \frac{1}{2\pi}\exp\left[-\frac{1}{2}\left(x_1 - \left[3\theta_1 + \frac{1}{3}\theta_2\right]\right)^2 - \frac{1}{2}\left(x_2 - \left[\frac{1}{3}\theta_1\right]\right)^2\right] \tag{6.53}$$

The objective is the cross entropy loss:

$$\mathcal{L}(\boldsymbol{\theta}) = -\mathbb{E}_{p^*(\boldsymbol{x})}\left[\log p(\boldsymbol{x};\boldsymbol{\theta})\right] \tag{6.54}$$

The gradient of this objective is given by

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \begin{pmatrix} = \mathbb{E}_{p^*(\boldsymbol{x})}\left[3(x_1 - [3\theta_1 + \frac{1}{3}\theta_2]) + \frac{1}{3}(x_2 - [\frac{1}{3}\theta_1])\right] \\ \mathbb{E}_{p^*(\boldsymbol{x})}\left[\frac{1}{3}(x_1 - [3\theta_1 + \frac{1}{3}\theta_2])\right] \end{pmatrix} \tag{6.55}$$

Suppose that $p^*(\boldsymbol{x}) = p(\boldsymbol{x}; [0, 0])$. Then the Fisher matrix is a constant matrix, given by

$$\mathbf{F} = \begin{pmatrix} 3^2 + \frac{1}{3^2} & 1 \\ 1 & \frac{1}{3^2} \end{pmatrix} \tag{6.56}$$

Figure 6.3 compares steepest descent in $\boldsymbol{\theta}$ space with the natural gradient method, which is equivalent to steepest descent in $\boldsymbol{\phi}$ space. Both methods start at $\boldsymbol{\theta} = (1, -1)$. The global optimum is at $\boldsymbol{\theta} = (0, 0)$. We see that the NG method (blue dots) converges much faster to this optimum and takes the shortest path, whereas steepest descent takes a very circuitous route. We also see that the gradient field in the whitened parameter space is more "spherical", which makes descent much simpler and faster.

Finally, note that since NGD is invariant to how we parameterize the distribution, we will get the same results even for a standard parameterization of the Gaussian. This is particularly useful if our probability model is more complex, such as a DNN (see e.g., [SSE18]).

6.4.4 Approximating the natural gradient

The main drawback of NGD is the computational cost of computing (the inverse of) the Fisher Information Matrix (FIM). To speed this up, several methods make assumptions about the form of $\mathbf{F}$, so it can be inverted efficiently. For example, [LeC+98] uses a diagonal approximation for

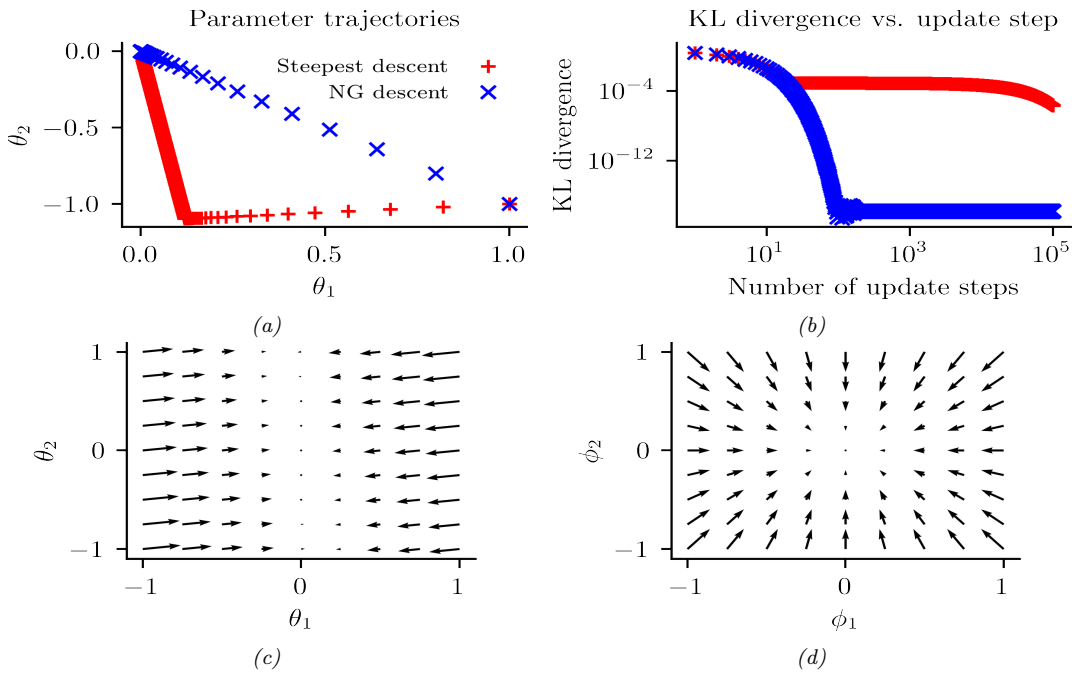


Figure 6.3: Illustration of the benefits of natural gradient vs steepest descent on a 2d problem. (a) Trajectories of the two methods in parameter space (red = steepest descent, blue = NG). They both start in the bottom right, at $(1, -1)$. (b) Objective vs number of iterations. (c) Gradient field in the $\boldsymbol{\theta}$ parameter space. (d) Gradient field in the whitened $\boldsymbol{\phi} = \mathbf{F}^{\frac{1}{2}}\boldsymbol{\theta}$ parameter space used by NG. Generated by nat_grad_demo.ipynb.

neural net training; [RMB08] uses a low-rank plus block diagonal approximation; and [GS15] assumes the covariance of the gradients can be modeled by a directed Gaussian graphical model with low treewidth (i.e., the Cholesky factorization of $\mathbf{F}$ is sparse).

[MG15] propose the **KFAC** method, which stands for "Kronecker-Factored approximate curvature"; this approximates the FIM of a DNN as a block diagonal matrix, where each block is a Kronecker product of two small matrices. This method has shown good results on supervised learning of neural nets [GM16; BGM17; Geo+18; Osa+19b] as well as reinforcement learning of neural policy networks [Wu+17]. The KFAC approximation can be justified using the mean field analysis of [AKO18]. In addition, [ZMG19] prove that KFAC will converge to the global optimum of a DNN if it is overparameterized (i.e., acts like an interpolator).

A simpler approach is to approximate the FIM by replacing the model's distribution with the empirical distribution. In particular, define $p_{\mathcal{D}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N}\sum_{n=1}^{N} \delta_{\boldsymbol{x}_n}(\boldsymbol{x})\delta_{\boldsymbol{y}_n}(\boldsymbol{y})$, $p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N}\sum_{n=1}^{N} \delta_{\boldsymbol{x}_n}(\boldsymbol{x})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$. Then we can compute the **empirical Fisher** Martens [Mar16] as

follows:

$$\mathbf{F} = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}}\right] \tag{6.57}$$

$$\approx \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}}\right] \tag{6.58}$$

$$= \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}} \tag{6.59}$$

This approximation is widely used, since it is simple to compute. In particular, we can compute a diagonal approximation using the squared gradient vector. (This is similar to AdaGrad, but only uses the current gradient instead of a moving average of gradients; the latter is a better approach when performing stochastic optimization.)

Unfortunately, the empirical Fisher does not work as well as the true Fisher [KBH19; Tho+19]. To see why, note that when we reach a flat part of parameter space where the gradient vector goes to zero, the empirical Fisher will become singular, and hence the algorithm will get stuck on this plateau. However, the true Fisher takes expectations over the outputs, i.e., it marginalizes out $\boldsymbol{y}$. This will allow it to detect small changes in the output if we change the parameters. This is why the natural gradient method can "escape" plateaus better than standard gradient methods.

An alternative strategy is to use exact computation of $\mathbf{F}$, but solve for $\mathbf{F}^{-1}\boldsymbol{g}$ approximately using truncated conjugate gradient (CG) methods, where each CG step uses efficient methods for Hessian-vector products [Pea94]. This is called **Hessian free optimization** [Mar10a]. However, this approach can be slow, since it may take many CG iterations to compute a single parameter update.

6.4.5 Natural gradients for the exponential family

In this section, we asssume $\mathcal{L}$ is an expected loss of the following form:

$$\mathcal{L}(\boldsymbol{\mu}) = \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] \tag{6.60}$$

where $q_{\boldsymbol{\mu}}(\boldsymbol{\theta})$ is an exponential family distribution with moment parameters $\boldsymbol{\mu}$. This is the basis of variational optimization (discussed in Section 6.7.3) and natural evolutionary strategies (discussed in Section 6.9.6).

It turns out the gradient wrt the moment parameters is the same as the natural gradient wrt the natural parameters $\boldsymbol{\lambda}$. This follows from the chain rule:

$$\frac{d}{d\boldsymbol{\lambda}}\mathcal{L}(\boldsymbol{\lambda}) = \frac{d\boldsymbol{\mu}}{d\boldsymbol{\lambda}}\frac{d}{d\boldsymbol{\mu}}\mathcal{L}(\boldsymbol{\mu}) = \mathbf{F}(\boldsymbol{\lambda})\nabla_{\boldsymbol{\mu}}\mathcal{L}(\boldsymbol{\mu}) \tag{6.61}$$

where $\mathcal{L}(\boldsymbol{\mu}) = \mathcal{L}(\boldsymbol{\lambda}(\boldsymbol{\mu}))$, and where we used Equation (2.215) to write

$$\mathbf{F}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}}\boldsymbol{\mu}(\boldsymbol{\lambda}) = \nabla^2_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda}) \tag{6.62}$$

Hence

$$\tilde{\nabla}_{\boldsymbol{\lambda}}\mathcal{L}(\boldsymbol{\lambda}) = \mathbf{F}(\boldsymbol{\lambda})^{-1}\nabla_{\boldsymbol{\lambda}}\mathcal{L}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\mu}}\mathcal{L}(\boldsymbol{\mu}) \tag{6.63}$$

It remains to compute the (regular) gradient wrt the moment parameters. The details on how to do this will depend on the form of the q and the form of $\mathcal{L}(\boldsymbol{\lambda})$. We discuss some approaches to this problem below.

6.4.5.1 Analytic computation for the Gaussian case

In this section, we assume that $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{m}, \mathbf{V})$. We now show how to compute the relevant gradients analytically.

Following Section 2.3.2.5, the natural parameters of q are

$$\boldsymbol{\lambda}^{(1)} = \mathbf{V}^{-1}\boldsymbol{m}, \; \boldsymbol{\lambda}^{(2)} = -\frac{1}{2}\mathbf{V}^{-1} \tag{6.64}$$

and the moment parameters are

$$\boldsymbol{\mu}^{(1)} = \boldsymbol{m}, \; \boldsymbol{\mu}^{(2)} = \mathbf{V} + \boldsymbol{m}\boldsymbol{m}^{\mathsf{T}} \tag{6.65}$$

For simplicity, we derive the result for the scalar case. Let $m = \mu^{(1)}$ and $v = \mu^{(2)} - (\mu^{(1)})^2$. By using the chain rule, the gradient wrt the moment parameters are

$$\frac{\partial \mathcal{L}}{\partial \mu^{(1)}} = \frac{\partial \mathcal{L}}{\partial m}\frac{\partial m}{\partial \mu^{(1)}} + \frac{\partial \mathcal{L}}{\partial v}\frac{\partial v}{\partial \mu^{(1)}} = \frac{\partial \mathcal{L}}{\partial m} - 2\frac{\partial \mathcal{L}}{\partial v}m \tag{6.66}$$

$$\frac{\partial \mathcal{L}}{\partial \mu^{(2)}} = \frac{\partial \mathcal{L}}{\partial m}\frac{\partial m}{\partial \mu^{(2)}} + \frac{\partial \mathcal{L}}{\partial v}\frac{\partial v}{\partial \mu^{(2)}} = \frac{\partial \mathcal{L}}{\partial v} \tag{6.67}$$

It remains to compute the derivatives wrt m and v. If $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{m}, \mathbf{V})$, then from **Bonnet's theorem** [Bon64] we have

$$\frac{\partial}{\partial m_i}\mathbb{E}\left[\ell(\boldsymbol{\theta})\right] = \mathbb{E}\left[\frac{\partial}{\partial \theta_i}\ell(\boldsymbol{\theta})\right] \tag{6.68}$$

And from **Price's theorem** [Pri58] we have

$$\frac{\partial}{\partial V_{ij}}\mathbb{E}\left[\ell(\boldsymbol{\theta})\right] = c_{ij}\mathbb{E}\left[\frac{\partial^2}{\partial \theta_i \theta_j}\ell(\boldsymbol{\theta})\right] \tag{6.69}$$

where $c_{ij} = \frac{1}{2}$ is $i = j$ and $c_{ij} = 1$ otherwise. (See gradient_expected_value_gaussian.ipynb for a "proof by example" of these claims.)

In the multivariate case, the result is as follows [OA09; KR21a]:

$$\nabla_{\boldsymbol{\mu}^{(1)}}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] = \nabla_{\boldsymbol{m}}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] - 2\nabla_{\mathbf{V}}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right]\boldsymbol{m} \tag{6.70}$$

$$= \mathbb{E}_{q(\boldsymbol{\theta})}\left[\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta})\right] - \mathbb{E}_{q(\boldsymbol{\theta})}\left[\nabla^2_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta})\right]\boldsymbol{m} \tag{6.71}$$

$$\nabla_{\boldsymbol{\mu}^{(2)}}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] = \nabla_{\mathbf{V}}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] \tag{6.72}$$

$$= \frac{1}{2}\mathbb{E}_{q(\boldsymbol{\theta})}\left[\nabla^2_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta})\right] \tag{6.73}$$

Thus we see that the natural gradients rely on both the gradient and Hessian of the loss function $\ell(\boldsymbol{\theta})$. We will see applications of this result in Section 6.7.2.2.

6.4.5.2 Stochastic approximation for the general case

In general, it can be hard to analytically compute the natural gradient. However, we can compute a Monte Carlo approximation. To see this, let us assume $\mathcal{L}$ is an expected loss of the following form:

$$\mathcal{L}(\boldsymbol{\mu}) = \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})}\left[\ell(\boldsymbol{\theta})\right] \tag{6.74}$$

From Equation (6.63) the natural gradient is given by

$$\nabla_{\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}) = \mathbf{F}(\boldsymbol{\lambda})^{-1} \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) \tag{6.75}$$

For exponential family distributions, both of these terms on the RHS can be written as expectations, and hence can be approximated by Monte Carlo, as noted by [KL17a]. To see this, note that

$$\mathbf{F}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \boldsymbol{\mu}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\boldsymbol{\theta})} \left[\mathcal{T}(\boldsymbol{\theta}) \right] \tag{6.76}$$

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\boldsymbol{\theta})} \left[\ell(\boldsymbol{\theta}) \right] \tag{6.77}$$

If q is reparameterizable, we can apply the reparameterization trick (Section 6.5.4) to push the gradient inside the expectation operator. This lets us sample $\boldsymbol{\theta}$ from q, compute the gradients, and average; we can then pass the resulting stochastic gradients to SGD.

6.4.5.3 Natural gradient of the entropy function

In this section, we discuss how to compute the natural gradient of the entropy of an exponential family distribution, which is useful when performing variational inference (Chapter 10). The natural gradient is given by

$$\tilde{\nabla}_{\boldsymbol{\lambda}} \mathbb{H}(\boldsymbol{\lambda}) = -\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\log q(\boldsymbol{\theta}) \right] \tag{6.78}$$

where, from Equation (2.143), we have

$$\log q(\boldsymbol{\theta}) = \log h(\boldsymbol{\theta}) + \mathcal{T}(\boldsymbol{\theta})^{\mathsf{T}} \boldsymbol{\lambda} - A(\boldsymbol{\lambda}) \tag{6.79}$$

Since $\mathbb{E}\left[\mathcal{T}(\boldsymbol{\theta})\right] = \boldsymbol{\mu}$, we have

$$\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\log q(\boldsymbol{\theta}) \right] = \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q(\boldsymbol{\theta})} \left[\log h(\boldsymbol{\theta}) \right] + \nabla_{\boldsymbol{\mu}} \boldsymbol{\mu}^{\mathsf{T}} \boldsymbol{\lambda}(\boldsymbol{\mu}) - \nabla_{\boldsymbol{\mu}} A(\boldsymbol{\lambda}) \tag{6.80}$$

where $h(\boldsymbol{\theta})$ is the base measure. Since $\boldsymbol{\lambda}$ is a function of $\boldsymbol{\mu}$, we have

$$\nabla_{\boldsymbol{\mu}} \boldsymbol{\mu}^{\mathsf{T}} \boldsymbol{\lambda} = \boldsymbol{\lambda} + (\nabla_{\boldsymbol{\mu}} \boldsymbol{\lambda})^{\mathsf{T}} \boldsymbol{\mu} = \boldsymbol{\lambda} + (\mathbf{F}_{\lambda}^{-1} \nabla_{\boldsymbol{\lambda}} \boldsymbol{\lambda})^{\mathsf{T}} \boldsymbol{\mu} = \boldsymbol{\lambda} + \mathbf{F}_{\lambda}^{-1} \boldsymbol{\mu} \tag{6.81}$$

and since $\boldsymbol{\mu} = \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda})$ we have

$$\nabla_{\boldsymbol{\mu}} A(\boldsymbol{\lambda}) = \mathbf{F}_{\boldsymbol{\lambda}}^{-1} \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda}) = \mathbf{F}_{\boldsymbol{\lambda}}^{-1} \boldsymbol{\mu} \tag{6.82}$$

Hence

$$-\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\log q(\boldsymbol{\theta}) \right] = -\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q(\boldsymbol{\theta})} \left[\log h(\boldsymbol{\theta}) \right] - \boldsymbol{\lambda} \tag{6.83}$$

If we assume that $h(\boldsymbol{\theta}) = \text{const}$, as is often the case, we get

$$\tilde{\nabla}_{\boldsymbol{\lambda}} \mathbb{H}(q) = -\boldsymbol{\lambda} \tag{6.84}$$

6.5 Gradients of stochastic functions

In this section, we discuss how to compute the gradient of stochastic functions of the form

$$\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\boldsymbol{z})} \left[\ell(\boldsymbol{\psi}, \boldsymbol{z}) \right] \tag{6.85}$$

6.5.1 Minibatch approximation to finite-sum objectives

In the simplest case, $q_{\boldsymbol{\psi}}(\boldsymbol{z})$ does not depend on $\boldsymbol{\psi}$. In this case, we can push gradients inside the expectation operator, $\nabla \mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}\left[\nabla \ell(\boldsymbol{\psi}, \boldsymbol{z})\right]$ and then use Monte Carlo sampling for $\boldsymbol{z}$ to approximate the gradient.

For example, consider the empirical risk minimization (ERM) problem of minimizing

$$\mathcal{L}(\boldsymbol{\psi}) = \frac{1}{N_{\mathcal{D}}} \sum_{n=1}^{N_{\mathcal{D}}} \ell(\boldsymbol{\psi}, \boldsymbol{z}_n) \tag{6.86}$$

where $\boldsymbol{z}_n = (\boldsymbol{x}_n, \boldsymbol{y}_n)$ and

$$\ell(\boldsymbol{\psi}, \boldsymbol{z}_n) = \ell(h(\boldsymbol{x}_n; \boldsymbol{\psi}), \boldsymbol{y}_n) \tag{6.87}$$

is the per-example loss, where h is a prediction function. This kind of objective is called a **finite sum objective**.

Now consider trying to minimize this objective. If, at each iteration, we evaluate the objective (and its gradient) using all $N_{\mathcal{D}}$ datapoints, the method is called **batch optimization**. However, this can be very slow if the dataset is large. Fortunately, we can reformulate it as a stochastic optimization problem, which will be faster to solve. To do this, note that Equation (6.86) can be written as an expectation wrt the empirical distribution:

$$\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{\boldsymbol{z} \sim p_{\mathcal{D}}}\left[\ell(\boldsymbol{\psi}, \boldsymbol{z})\right] \tag{6.88}$$

Since the distribution is independent of the parameters, we can easily use Monte Carlo sampling to approximate the objective and its gradient. In particular, we will sample a **minibatch** of $B = |\mathcal{B}|$ datapoints from the full set $\mathcal{D}$ at each iteration. More precisely, we have

$$\mathcal{L}(\boldsymbol{\psi}) \approx \frac{1}{B} \sum_{n \in \mathcal{B}} \ell(h(\boldsymbol{x}_n; \boldsymbol{\psi}), \boldsymbol{y}_n) \tag{6.89}$$

$$\nabla \mathcal{L}(\boldsymbol{\psi}) \approx \frac{1}{B} \sum_{n \in \mathcal{B}} \nabla \ell(h(\boldsymbol{x}_n; \boldsymbol{\psi}), \boldsymbol{y}_n) \tag{6.90}$$

These noisy gradients can then be passed to SGD, which is robust to noisy gradients (see Section 6.3).

6.5.2 Optimizing parameters of a distribution

Now suppose the stochasticity depends on the parameters we are optimizing. For example, $\boldsymbol{z}$ could be an action sampled from a stochastic policy $q_{\boldsymbol{\psi}}$, as in RL (Section 35.3.2). In this case, the gradient is given by

$$\nabla_{\boldsymbol{\psi}} \mathbb{E}_{q_{\boldsymbol{\psi}}(\boldsymbol{z})}\left[\ell(\boldsymbol{\psi}, \boldsymbol{z}\right] = \nabla_{\boldsymbol{\psi}} \int \ell(\boldsymbol{\psi}, \boldsymbol{z}) q_{\boldsymbol{\psi}}(\boldsymbol{z}) d\boldsymbol{z} \tag{6.91}$$

$$= \int \left[\nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z})\right] q_{\boldsymbol{\psi}}(\boldsymbol{z}) d\boldsymbol{z} + \int \ell(\boldsymbol{\psi}, \boldsymbol{z}) \left[\nabla_{\boldsymbol{\psi}} q_{\boldsymbol{\psi}}(\boldsymbol{z})\right] d\boldsymbol{z} \tag{6.92}$$

The first term can be approximated by Monte Carlo sampling:

$$\int \left[\nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z})\right] q_{\boldsymbol{\psi}}(\boldsymbol{z}) d\boldsymbol{z} \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z}_s) \tag{6.93}$$

where $\boldsymbol{z}_s \sim q_{\boldsymbol{\psi}}$. Note that if $\ell()$ is independent of $\boldsymbol{\psi}$, this term vanishes.

Now consider the second term, that takes the gradients of the distribution itself:

$$I \triangleq \int \ell(\boldsymbol{\psi}, \boldsymbol{z}) \left[\nabla_{\boldsymbol{\psi}} q_{\boldsymbol{\psi}}(\boldsymbol{z})\right] d\boldsymbol{z} \tag{6.94}$$

We can no longer use vanilla Monte Carlo sampling to approximate this integral. However, there are various other ways to approximate this (see [Moh+19] for an extensive review). We briefly describe the two main methods in Section 6.5.3 and Section 6.5.4.

6.5.3 Score function estimator (likelihood ratio trick)

The simplest way to approximate Equation (6.94) is to exploit the **log derivative trick**, which is the following identity:

$$\nabla_{\boldsymbol{\psi}} q_{\boldsymbol{\psi}}(\boldsymbol{z}) = q_{\boldsymbol{\psi}}(\boldsymbol{z}) \nabla_{\boldsymbol{\psi}} \log q_{\boldsymbol{\psi}}(\boldsymbol{z}) \tag{6.95}$$

With this, we can rewrite Equation (6.94) as follows:

$$I = \int \ell(\boldsymbol{\psi}, \boldsymbol{z}) [q_{\boldsymbol{\psi}}(\boldsymbol{z}) \nabla_{\boldsymbol{\psi}} \log q_{\boldsymbol{\psi}}(\boldsymbol{z})] d\boldsymbol{z} = \mathbb{E}_{q_{\boldsymbol{\psi}}(\boldsymbol{z})} \left[\ell(\boldsymbol{\psi}, \boldsymbol{z}) \nabla_{\boldsymbol{\psi}} \log q_{\boldsymbol{\psi}}(\boldsymbol{z})\right] \tag{6.96}$$

This is called the **score function estimator** or **SFE** [Fu15]. (The term "score function" refers to the gradient of a log probability distribution, as explained in Section 2.4.1.) It is also called the **likelihood ratio gradient estimator**. We can now easily approximate this with Monte Carlo:

$$I \approx \frac{1}{S} \sum_{s=1}^{S} \ell(\boldsymbol{\psi}, \boldsymbol{z}_s) \nabla_{\boldsymbol{\psi}} \log q_{\boldsymbol{\psi}}(\boldsymbol{z}_s) \tag{6.97}$$

We only require that the sampling distribution is differentiable, not the objective $\ell(\boldsymbol{\psi}, \boldsymbol{z})$ itself. This allows the method to be used for blackbox stochastic optimization problems, such as variational optimization (Section 6.7.3), black-box variational inference (Section 10.3.2), reinforcement learning (Section 35.3.2), etc.

6.5.3.1 Control variates

The score function estimate can have high variance. One way to reduce this is to use **control variates**, in which we replace $\ell(\boldsymbol{\psi}, \boldsymbol{z})$ with

$$\hat{\ell}(\boldsymbol{\psi}, \boldsymbol{z}) = \ell(\boldsymbol{\psi}, \boldsymbol{z}) - c \left(b(\boldsymbol{\psi}, \boldsymbol{z}) - \mathbb{E}\left[b(\boldsymbol{\psi}, \boldsymbol{z})\right]\right) \tag{6.98}$$

where $b(\boldsymbol{\psi}, \boldsymbol{z})$ is a **baseline function** that is correlated with $\ell(\boldsymbol{\psi}, \boldsymbol{z})$, and $c > 0$ is a coefficient. Since $\mathbb{E}\left[\hat{\ell}(\boldsymbol{\psi}, \boldsymbol{z})\right] = \mathbb{E}\left[\ell(\boldsymbol{\psi}, \boldsymbol{z})\right]$, we can use $\hat{\ell}$ to compute unbiased gradient estimates of ℓ. The advantage is that this new estimate can result in lower variance, as we show in Section 11.6.3.

6.5.3.2 Rao-Blackwellisation

Suppose $q_{\boldsymbol{\psi}}(\boldsymbol{z})$ is a discrete distribution. In this case, our objective becomes $\mathcal{L}(\boldsymbol{\psi}) = \sum_{\boldsymbol{z}} \ell(\boldsymbol{\psi}, \boldsymbol{z}) q_{\boldsymbol{\psi}}(\boldsymbol{z})$. For simplicity, let us assume $\ell(\boldsymbol{\psi}, \boldsymbol{z}) = \ell(\boldsymbol{z})$.

We can now easily compute gradients using $\nabla_{\boldsymbol{\psi}} \mathcal{L}(\boldsymbol{\psi}) = \sum_{\boldsymbol{z}} \ell(\boldsymbol{z}) \nabla_{\boldsymbol{\psi}} q_{\boldsymbol{\psi}}(\boldsymbol{z})$. Of course, if $\boldsymbol{z}$ can take on exponentially many values (e.g., we are optimizing over the space of strings), this expression is intractable. However, suppose we can partition this sum into two sets, a small set S_1 of high probability values and a large set S_2 of all other values. Then we can enumerate over S_1 and use the score function estimator for S_2:

$$\nabla_{\boldsymbol{\psi}} \mathcal{L}(\boldsymbol{\psi}) = \sum_{\boldsymbol{z} \in S_1} \ell(\boldsymbol{z}) \nabla_{\boldsymbol{\psi}} q_{\boldsymbol{\psi}}(\boldsymbol{z}) + \mathbb{E}_{q_{\boldsymbol{\psi}}(\boldsymbol{z} | \boldsymbol{z} \in S_2)} \left[\ell(\boldsymbol{z}) \nabla_{\boldsymbol{\psi}} \log q_{\boldsymbol{\psi}}(\boldsymbol{z}) \right] \tag{6.99}$$

To compute the second expectation, we can use rejection sampling applied to samples from $q_{\boldsymbol{\psi}}(\boldsymbol{z})$. This procedure is a form of Rao-Blackwellisation as shown in [Liu+19b], and reduces the variance compared to standard SFE (see Section 11.6.2 for details on Rao-Blackwellisation).

6.5.4 Reparameterization trick

The score function estimator can have high variance, even when using a control variate. In this section, we derive a lower variance estimator, which can be applied if $\ell(\boldsymbol{\psi}, \boldsymbol{z})$ is differentiable wrt $\boldsymbol{z}$. We additionally require that we can compute a sample from $q_{\boldsymbol{\psi}}(\boldsymbol{z})$ by first sampling $\boldsymbol{\epsilon}$ from some noise distribution q_0 which is independent of $\boldsymbol{\psi}$, and then transforming to $\boldsymbol{z}$ using a deterministic and differentiable function $\boldsymbol{z} = r(\boldsymbol{\psi}, \boldsymbol{\epsilon})$. For example, instead of sampling $\boldsymbol{z} \sim \mathcal{N}(\mu, \sigma^2)$, we can sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$ and compute

$$\boldsymbol{z} = r(\boldsymbol{\psi}, \boldsymbol{\epsilon}) = \mu + \sigma \boldsymbol{\epsilon} \tag{6.100}$$

where $\boldsymbol{\psi} = (\mu, \sigma)$. This allows us to rewrite our stochastic objective as follows:

$$\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\boldsymbol{z})} \left[\ell(\boldsymbol{\psi}, \boldsymbol{z}) \right] = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} \left[\ell(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \boldsymbol{\epsilon})) \right] \tag{6.101}$$

Since $q_0(\boldsymbol{\epsilon})$ is independent of $\boldsymbol{\psi}$, we can push the gradient operator inside the expectation, which we can approximate with Monte Carlo:

$$\nabla_{\boldsymbol{\psi}} \mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} \left[\nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \boldsymbol{\epsilon})) \right] \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \boldsymbol{\epsilon}_s)) \tag{6.102}$$

where $\boldsymbol{\epsilon}_s \sim q_0$. This is called the **reparameterization gradient** or the **pathwise derivative** [Gla03; Fu15; KW14; RMW14a; TLG14; JO18; FMM18], and is widely used in variational inference (Section 10.3.3), and when fitting VAE models (Section 21.2).

6.5.4.1 Example

As a simple example, suppose we define some arbitrary function, such as $\ell(z) = z^2 - 3z$, and then define its expected value as $\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{\mathcal{N}(z|\mu,v)} \left[\ell(z) \right]$, where $\boldsymbol{\psi} = (\mu, v)$ and $v = \sigma^2$. Suppose we want to compute

$$\nabla_{\boldsymbol{\psi}} \mathcal{L}(\boldsymbol{\psi}) = \left[\frac{\partial}{\partial \mu} \mathbb{E} \left[\ell(z) \right], \frac{\partial}{\partial v} \mathbb{E} \left[\ell(z) \right] \right] \tag{6.103}$$

Since the Gaussian distribution is reparameterizable, we can sample $z \sim \mathcal{N}(z|\mu, v)$, and then use automatic differentiation to compute each of these gradient terms, and then average.

However, in the special case of Gaussian distributions, we can also compute the gradient vector directly. In particular, in Section 6.4.5.1 we present Bonnet's theorem, which states that

$$\frac{\partial}{\partial \mu}\mathbb{E}\left[\ell(z)\right] = \mathbb{E}\left[\frac{\partial}{\partial z}\ell(z)\right] \tag{6.104}$$

Similarly, Price's theorem states that

$$\frac{\partial}{\partial v}\mathbb{E}\left[\ell(z)\right] = 0.5\mathbb{E}\left[\frac{\partial^2}{\partial z^2}\ell(z)\right] \tag{6.105}$$

In gradient_expected_value_gaussian.ipynb we show that these two methods are numerically equivalent, as theory suggests.

6.5.4.2 Total derivative

To compute the gradient term inside the expectation in Equation (6.102) we need to use the **total derivative**, since the function ℓ depends on $\boldsymbol{\psi}$ directly and via the noise sample. Recall that, for a function of the form $f(\psi_1, \ldots, \psi_{d_\psi}, z_1(\boldsymbol{\psi}), \ldots, z_{d_z}(\boldsymbol{\psi}))$, the total derivative wrt ψ_i is given by the chain rule as follows:

$$\frac{\partial \ell}{\partial \psi_i}^{\text{TD}} = \frac{\partial \ell}{\partial \psi_i} + \sum_j \frac{\partial \ell}{\partial z_j}\frac{\partial z_j}{\partial \psi_i} \tag{6.106}$$

and hence

$$\nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z})^{\text{TD}} = \nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z}) + \mathbf{J}^\mathsf{T} \nabla_{\boldsymbol{z}} \ell(\boldsymbol{\psi}, \boldsymbol{z}) \tag{6.107}$$

where $\mathbf{J} = \frac{\partial \boldsymbol{z}^\mathsf{T}}{\partial \boldsymbol{\psi}}$ is the $d_z \times d_\psi$ Jacobian matrix of the noise transformation:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial \boldsymbol{z}_1}{\partial \boldsymbol{\psi}_1} & \cdots & \frac{\partial \boldsymbol{z}_1}{\partial \boldsymbol{\psi}_{d_\psi}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \boldsymbol{z}_{d_z}}{\partial \boldsymbol{\psi}_{d_\psi}} & \cdots & \frac{\partial \boldsymbol{z}_{d_z}}{\partial \boldsymbol{\psi}_{d_\psi}} \end{pmatrix} \tag{6.108}$$

Hence we can compute the gradient using

$$\nabla_{\boldsymbol{\psi}} \mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_0(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\psi}} \ell(\boldsymbol{\psi}, \boldsymbol{z}) + \mathbf{J}(\boldsymbol{\psi}, \boldsymbol{\epsilon})^\mathsf{T} \nabla_{\boldsymbol{z}} \ell(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \boldsymbol{\epsilon}))\right] \tag{6.109}$$

We leverage this decomposition in Section 10.3.3.1, where we derive a lower variance gradient estimator in the special case of variational inference.

6.5.5 The delta method

The **delta method** [Hoe12] approximates the expectation of a function of a random variable by the expectation of the function's Taylor expansion. For example, suppose $\boldsymbol{\theta} \sim q$ with mean $\mathbb{E}_{q(\boldsymbol{\theta})}[\boldsymbol{\theta}] = \boldsymbol{m}$, and we use a first order expansion. Then we have

$$\mathbb{E}_{q(\boldsymbol{\theta})}[f(\boldsymbol{\theta})] \approx \mathbb{E}_{q(\boldsymbol{\theta})}\left[f(\boldsymbol{m}) + (\boldsymbol{\theta} - \boldsymbol{m})^{\mathsf{T}} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}}\right] \approx f(\boldsymbol{m}) \tag{6.110}$$

Now let $f(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. Then we have

$$\mathbb{E}_{q(\boldsymbol{\theta})}[\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})] \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}} \tag{6.111}$$

This is called the **first-order delta method**.

Now let $f(\boldsymbol{\theta}) = \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. Then we have

$$\mathbb{E}_{q(\boldsymbol{\theta})}\left[\nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})\right] \approx \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}} \tag{6.112}$$

This is called the **second-order delta method**.

6.5.6 Gumbel softmax trick

When working with discrete variables, we cannot use the reparameterization trick. However, we can often relax the discrete variables to continuous ones in a way which allows the trick to be used, as we explain below.

Consider a one-hot vector $\boldsymbol{d}$ with K bits, so $d_k \in \{0, 1\}$ and $\sum_{k=1}^K d_k = 1$. This can be used to represent a K-ary categorical variable d. Let $P(d) = \text{Cat}(d|\boldsymbol{\pi})$, where $\pi_k = P(d_k = 1)$, so $0 \leq \pi_k \leq 1$. Alternatively we can parameterize the distribution in terms of $(\alpha_1, \ldots, \alpha_k)$, where $\pi_k = \alpha_k / (\sum_{k'=1}^K \alpha_{k'})$. We will denote this by $d \sim \text{Cat}(d|\boldsymbol{\alpha})$.

We can sample a one-hot vector $\boldsymbol{d}$ from this distribution by computing

$$\boldsymbol{d} = \text{onehot}(\underset{k}{\text{argmax}}[\epsilon_k + \log \alpha_k]) \tag{6.113}$$

where $\epsilon_k \sim \text{Gumbel}(0, 1)$ is sampled from the **Gumbel distribution** [Gum54]. We can draw such samples by first sampling $u_k \sim \text{Unif}(0, 1)$ and then computing $\epsilon_k = -\log(-\log(u_k))$. This is called the **Gumbel-Max trick** [MTM14], and gives us a reparameterizable representation for the categorical distribution.

Unfortunately, the derivative of the argmax is 0 everywhere except at the boundary of transitions from one label to another, where the derivative is undefined. However, suppose we replace the argmax with a softmax, and replace the discrete one-hot vector $\boldsymbol{d}$ with a continuous relaxation $\boldsymbol{x} \in \Delta^{K-1}$, where $\Delta^{K-1} = \{\boldsymbol{x} \in \mathbb{R}^K : x_k \in [0, 1], \sum_{k=1}^K x_k = 1\}$ is the K-dimensional simplex. Then we can write

$$x_k = \frac{\exp((\log \alpha_k + \epsilon_k)/\tau)}{\sum_{k'=1}^K \exp((\log \alpha_{k'} + \epsilon_{k'})/\tau)} \tag{6.114}$$

where $\tau > 0$ is a temperature parameter. This is called the **Gumbel-Softmax distribution** [JGP17] or the **concrete distribution** [MMT17]. This smoothly approaches the discrete distribution as $\tau \to 0$, as illustrated in Figure 6.4.

We can now replace $f(\boldsymbol{d})$ with $f(\boldsymbol{x})$, which allows us to take reparameterized gradients wrt $\boldsymbol{x}$.

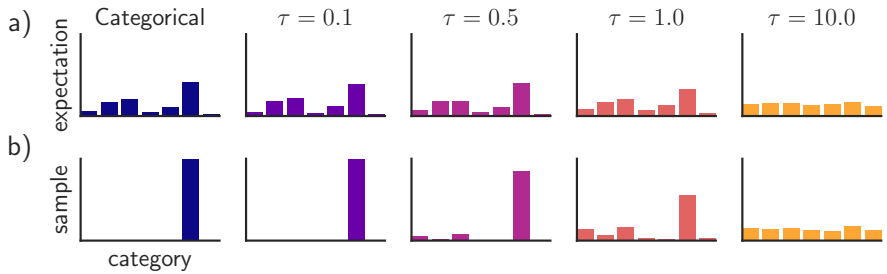


Figure 6.4: Illustration of the Gumbel-Softmax (concrete) distribution with $K = 7$ states at different temperatures τ. The top row shows $\mathbb{E}[\boldsymbol{z}]$, and the bottom row shows samples $\boldsymbol{z} \sim \text{GumbelSoftmax}(\boldsymbol{\alpha}, \tau)$. The left column shows a discrete (categorical) distribution, which always produces one-hot samples. From Figure 1 of [JGP17]. Used with kind permission of Ben Poole.

6.5.7 Stochastic computation graphs

We can represent an arbitrary function containing both deterministic and stochastic components as a **stochastic computation graph**. We can then generalize the AD algorithm (Section 6.2) to leverage score function estimation (Section 6.5.3) and reparameterization (Section 6.5.4) to compute Monte Carlo gradients for complex nested functions. For details, see [Sch+15a; Gaj+19].

6.5.8 Straight-through estimator

In this section, we discuss how to approximate the gradient of a quantized version of a signal. For example, suppose we have the following thresholding function, that binarizes its output:

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{6.115}$$

This does not have a well-defined gradient. However, we can use the **straight-through estimator** proposed in [Ben13] as an approximation. The basic idea is to replace $g(x) = f'(x)$, where $f'(x)$ is the derivative of f wrt input, with $g(x) = x$ when computing the backwards pass. See Figure 6.5 for a visualization, and [Yin+19b] for an analysis of why this is a valid approximation.

In practice, we sometimes replace $g(x) = x$ with the **hard tanh** function, defined by

$$\text{HardTanh}(x) = \begin{cases} x & \text{if } -1 \leq x \leq 1 \\ 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \end{cases} \tag{6.116}$$

This ensures the gradients that are backpropagated don't get too large. See Section 21.6 for an application of this approach to discrete autoencoders.

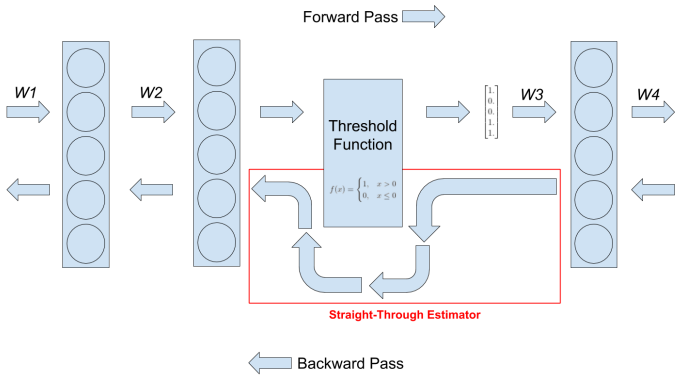


Figure 6.5: Illustration of straight-through estimator when applied to a binary threshold function in the middle of an MLP. From https://www.hassanaskary.com/python/pytorch/deep%20learning/2020/09/19/intuitive-explanation-of-straight-through-estimators.html. Used with kind permission of Hassan Askary.

6.6 Bound optimization (MM) algorithms

In this section, we consider a class of algorithms known as **bound optimization** or **MM** algorithms. In the context of minimization, MM stands for **majorize-minimize**. In the context of maximization, MM stands for **minorize-maximize**. There are many examples of MM algorithms, such as EM (Section 6.6.3), proximal gradient methods (Section 4.1), the mean shift algorithm for clustering [FH75; Che95; FT05], etc. For more details, see e.g., [HL04; Mai15; SBP17; Nad+19],

6.6.1 The general algorithm

In this section, we assume our goal is to *maximize* some function $\ell(\boldsymbol{\theta})$ wrt its parameters $\boldsymbol{\theta}$. The basic approach in MM algorithms is to construct a **surrogate function** $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t)$ which is a tight lowerbound to $\ell(\boldsymbol{\theta})$ such that $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \leq \ell(\boldsymbol{\theta})$ and $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$. If these conditions are met, we say that Q minorizes ℓ. We then perform the following update at each step:

$$\boldsymbol{\theta}^{t+1} = \operatorname*{argmax}_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \tag{6.117}$$

This guarantees us monotonic increases in the original objective:

$$\ell(\boldsymbol{\theta}^{t+1}) \geq Q(\boldsymbol{\theta}^{t+1}, \boldsymbol{\theta}^t) \geq Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t) \tag{6.118}$$

where the first inequality follows since $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}')$ is a lower bound on $\ell(\boldsymbol{\theta}^t)$ for any $\boldsymbol{\theta}'$; the second inequality follows from Equation (6.117); and the final equality follows the tightness property. As a consequence of this result, if you do not observe monotonic increase of the objective, you must have an error in your math and/or code. This is a surprisingly powerful debugging tool.

This process is sketched in Figure 6.6. The dashed red curve is the original function (e.g., the log-likelihood of the observed data). The solid blue curve is the lower bound, evaluated at $\boldsymbol{\theta}^t$; this

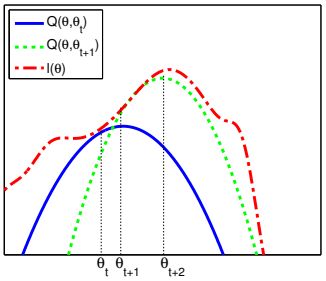


Figure 6.6: Illustration of a bound optimization algorithm. Adapted from Figure 9.14 of [Bis06]. Generated by em_log_likelihood_max.ipynb.

touches the objective function at $\boldsymbol{\theta}^t$. We then set $\boldsymbol{\theta}^{t+1}$ to the maximum of the lower bound (blue curve), and fit a new bound at that point (dotted green curve). The maximum of this new bound becomes $\boldsymbol{\theta}^{t+2}$, etc.

6.6.2 Example: logistic regression

If $\ell(\boldsymbol{\theta})$ is a concave function we want to maximize, then one way to obtain a valid lower bound is to use a bound on its Hessian, i.e., to find a matrix a negative definite matrix $\mathbf{B}$ such that $\mathbf{H}(\boldsymbol{\theta}) \succ \mathbf{B}$. In this case, one can show (see [BCN18, App. B]) that

$$\ell(\boldsymbol{\theta}) \geq \ell(\boldsymbol{\theta}^t) + (\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \boldsymbol{g}(\boldsymbol{\theta}^t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \mathbf{B}(\boldsymbol{\theta} - \boldsymbol{\theta}^t) \tag{6.119}$$

where $\boldsymbol{g}(\boldsymbol{\theta}^t) = \nabla \ell(\boldsymbol{\theta}^t)$. Therefore the following function is a valid lower bound:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \boldsymbol{\theta}^\mathsf{T}(\boldsymbol{g}(\boldsymbol{\theta}^t) - \mathbf{B}\boldsymbol{\theta}^t) + \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{B}\boldsymbol{\theta} \tag{6.120}$$

The corresponding update becomes

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \mathbf{B}^{-1}\boldsymbol{g}(\boldsymbol{\theta}^t) \tag{6.121}$$

This is similar to a Newton update, except we use $\mathbf{B}$, which is a fixed matrix, rather than $\mathbf{H}(\boldsymbol{\theta}^t)$, which changes at each iteration. This can give us some of the advantages of second order methods at lower computational cost.

For example, let us fit a multi-class logistic regression model using MM. (We follow the presentation of [Kri+05], who also consider the more interesting case of *sparse* logistic regression.) The probability that example n belongs to class $c \in \{1, \ldots, C\}$ is given by

$$p(y_n = c|\boldsymbol{x}_n, \boldsymbol{w}) = \frac{\exp(\boldsymbol{w}_c^\mathsf{T}\boldsymbol{x}_n)}{\sum_{i=1}^C \exp(\boldsymbol{w}_i^\mathsf{T}\boldsymbol{x}_n)} \tag{6.122}$$

Because of the normalization condition $\sum_{c=1}^C p(y_n = c|\boldsymbol{x}_n, \boldsymbol{w}) = 1$, we can set $\boldsymbol{w}_C = \mathbf{0}$. (For example, in binary logistic regression, where $C = 2$, we only learn a single weight vector.) Therefore the parameters $\boldsymbol{\theta}$ correspond to a weight matrix $\boldsymbol{w}$ of size $D(C-1)$, where $\boldsymbol{x}_n \in \mathbb{R}^D$.

If we let $\boldsymbol{p}_n(\boldsymbol{w}) = [p(y_n = 1|\boldsymbol{x}_n, \boldsymbol{w}), \ldots, p(y_n = C-1|\boldsymbol{x}_n, \boldsymbol{w})]$ and $\boldsymbol{y}_n = [\mathbb{I}(y_n = 1), \ldots, \mathbb{I}(y_n = C-1)]$, we can write the log-likelihood as follows:

$$\ell(\boldsymbol{w}) = \sum_{n=1}^{N} \left[\sum_{c=1}^{C-1} y_{nc} \boldsymbol{w}_c^\mathsf{T} \boldsymbol{x}_n - \log \sum_{c=1}^{C} \exp(\boldsymbol{w}_c^\mathsf{T} \boldsymbol{x}_n) \right] \tag{6.123}$$

The gradient is given by the following:

$$\boldsymbol{g}(\boldsymbol{w}) = \sum_{n=1}^{N} (\boldsymbol{y}_n - \boldsymbol{p}_n(\boldsymbol{w})) \otimes \boldsymbol{x}_n \tag{6.124}$$

where $\otimes$ denotes kronecker product (which, in this case, is just outer product of the two vectors). The Hessian is given by the following:

$$\mathbf{H}(\boldsymbol{w}) = -\sum_{n=1}^{N} (\mathrm{diag}(\boldsymbol{p}_n(\boldsymbol{w})) - \boldsymbol{p}_n(\boldsymbol{w}) \boldsymbol{p}_n(\boldsymbol{w})^\mathsf{T}) \otimes (\boldsymbol{x}_n \boldsymbol{x}_n^\mathsf{T}) \tag{6.125}$$

We can construct a lower bound on the Hessian, as shown in [Boh92]:

$$\mathbf{H}(\boldsymbol{w}) \succ -\frac{1}{2}[\mathbf{I} - \mathbf{1}\mathbf{1}^\mathsf{T}/C] \otimes (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\mathsf{T}) \triangleq \mathbf{B} \tag{6.126}$$

where $\mathbf{I}$ is a $(C-1)$-dimensional identity matrix, amd $\mathbf{1}$ is a $(C-1)$-dimensional vector of all 1s. In the binary case, this becomes

$$\mathbf{H}(\boldsymbol{w}) \succ -\frac{1}{2}(1 - \frac{1}{2})(\sum_{n=1}^{N} \boldsymbol{x}_n^\mathsf{T} \boldsymbol{x}_n) = -\frac{1}{4} \mathbf{X}^\mathsf{T} \mathbf{X} \tag{6.127}$$

This follows since $p_n \leq 0.5$ so $-(p_n - p_n^2) \geq -0.25$.

We can use this lower bound to construct an MM algorithm to find the MLE. The update becomes

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \mathbf{B}^{-1} \boldsymbol{g}(\boldsymbol{w}^t) \tag{6.128}$$

For example, let us consider the binary case, so $\boldsymbol{g}^t = \nabla \ell(\boldsymbol{w}^t) = \mathbf{X}^\mathsf{T}(\boldsymbol{y} - \boldsymbol{\mu}^t)$, where $\boldsymbol{\mu}^t = [p_n(\boldsymbol{w}^t), (1 - p_n(\boldsymbol{w}^t))]_{n=1}^N$. The update becomes

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - 4(\mathbf{X}^\mathsf{T} \mathbf{X})^{-1} \boldsymbol{g}^t \tag{6.129}$$

The above is faster (per step) than the IRLS (iteratively reweighted least squares) algorithm (i.e., Newton's method), which is the standard method for fitting GLMs. To see this, note that the Newton update has the form

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \mathbf{H}^{-1} \boldsymbol{g}(\boldsymbol{w}^t) = \boldsymbol{w}^t - (\mathbf{X}^\mathsf{T} \mathbf{S}^t \mathbf{X})^{-1} \boldsymbol{g}^t \tag{6.130}$$

where $\mathbf{S}^t = \mathrm{diag}(\boldsymbol{\mu}^t \odot (1 - \boldsymbol{\mu}^t))$. We see that Equation (6.129) is faster to compute, since we can precompute the constant matrix $(\mathbf{X}^\mathsf{T} \mathbf{X})^{-1}$.

6.6.3 The EM algorithm

In this section, we discuss the **expectation maximization** (**EM**) algorithm [DLR77; MK07], which is an algorithm designed to compute the MLE or MAP parameter estimate for probability models that have **missing data** and/or **hidden variables**. It is a special case of an MM algorithm.

The basic idea behind EM is to alternate between estimating the hidden variables (or missing values) during the **E step** (expectation step), and then using the fully observed data to compute the MLE during the **M step** (maximization step). Of course, we need to iterate this process, since the expected values depend on the parameters, but the parameters depend on the expected values.

In Section 6.6.3.1, we show that EM is a **bound optimization** algorithm, which implies that this iterative procedure will converge to a local maximum of the log likelihood. The speed of convergence depends on the amount of missing data, which affects the tightness of the bound [XJ96; MD97; SRG03; KKS20].

We now describe the EM algorithm for a generic model. We let $\boldsymbol{y}_n$ be the visible data for example n, and $\boldsymbol{z}_n$ be the hidden data.

6.6.3.1 Lower bound

The goal of EM is to maximize the log likelihood of the observed data:

$$\ell(\boldsymbol{\theta}) = \sum_{n=1}^{N_{\mathcal{D}}} \log p(\boldsymbol{y}_n|\boldsymbol{\theta}) = \sum_{n=1}^{N_{\mathcal{D}}} \log \left[\sum_{\boldsymbol{z}_n} p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta}) \right] \tag{6.131}$$

where $\boldsymbol{y}_n$ are the visible variables and $\boldsymbol{z}_n$ are the hidden variables. Unfortunately this is hard to optimize, since the log cannot be pushed inside the sum.

EM gets around this problem as follows. First, consider a set of arbitrary distributions $q_n(\boldsymbol{z}_n)$ over each hidden variable $\boldsymbol{z}_n$. The observed data log likelihood can be written as follows:

$$\ell(\boldsymbol{\theta}) = \sum_{n=1}^{N_{\mathcal{D}}} \log \left[\sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \right] \tag{6.132}$$

Using Jensen's inequality, we can push the log (which is a concave function) inside the expectation to get the following lower bound on the log likelihood:

$$\ell(\boldsymbol{\theta}) \geq \sum_n \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \tag{6.133}$$

$$= \sum_n \underbrace{\mathbb{E}_{q_n} \left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta}) \right] + \mathbb{H}(q_n)}_{Ł(\boldsymbol{\theta}, q_n|\boldsymbol{y}_n)} \tag{6.134}$$

$$= \sum_n Ł(\boldsymbol{\theta}, q_n|\boldsymbol{y}_n) \triangleq Ł(\boldsymbol{\theta}, \{q_n\}|\mathcal{D}) \tag{6.135}$$

where $\mathbb{H}(q)$ is the entropy of probability distribution q, and $Ł(\boldsymbol{\theta}, \{q_n\}|\mathcal{D})$ is called the **evidence lower bound** or **ELBO**, since it is a lower bound on the log marginal likelihood, $\log p(\boldsymbol{y}_{1:N}|\boldsymbol{\theta})$, also called the evidence. Optimizing this bound is the basis of variational inference, as we discuss in Section 10.1.

6.6.3.2 E step

We see that the lower bound is a sum of N terms, each of which has the following form:

$$\text{Ł}(\boldsymbol{\theta}, q_n|\boldsymbol{y}_n) = \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \tag{6.136}$$

$$= \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{z}_n|\boldsymbol{y}_n, \boldsymbol{\theta}) p(\boldsymbol{y}_n|\boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \tag{6.137}$$

$$= \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{z}_n|\boldsymbol{y}_n, \boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} + \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log p(\boldsymbol{y}_n|\boldsymbol{\theta}) \tag{6.138}$$

$$= -D_{\mathbb{KL}}\left(q_n(\boldsymbol{z}_n) \parallel p(\boldsymbol{z}_n|\boldsymbol{y}_n, \boldsymbol{\theta})\right) + \log p(\boldsymbol{y}_n|\boldsymbol{\theta}) \tag{6.139}$$

where $D_{\mathbb{KL}}\left(q \parallel p\right) \triangleq \sum_z q(z) \log \frac{q(z)}{p(z)}$ is the Kullback-Leibler divergence (or KL divergence for short) between probability distributions q and p. We discuss this in more detail in Section 5.1, but the key property we need here is that $D_{\mathbb{KL}}\left(q \parallel p\right) \geq 0$ and $D_{\mathbb{KL}}\left(q \parallel p\right) = 0$ iff $q = p$. Hence we can maximize the lower bound $\text{Ł}(\boldsymbol{\theta}, \{q_n\}|\mathcal{D})$ wrt $\{q_n\}$ by setting each one to $q_n^* = p(\boldsymbol{z}_n|\boldsymbol{y}_n, \boldsymbol{\theta})$. This is called the **E step**. This ensures the ELBO is a tight lower bound:

$$\text{Ł}(\boldsymbol{\theta}, \{q_n^*\}|\mathcal{D}) = \sum_n \log p(\boldsymbol{y}_n|\boldsymbol{\theta}) = \ell(\boldsymbol{\theta}|\mathcal{D}) \tag{6.140}$$

To see how this connects to bound optimization, let us define

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \text{Ł}(\boldsymbol{\theta}, \{p(\boldsymbol{z}_n|\boldsymbol{y}_n; \boldsymbol{\theta}^t)\}) \tag{6.141}$$

Then we have $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \leq \ell(\boldsymbol{\theta})$ and $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$, as required.

However, if we cannot compute the posteriors $p(\boldsymbol{z}_n|\boldsymbol{y}_n; \boldsymbol{\theta}^t)$ exactly, we can still use an approximate distribution $q(\boldsymbol{z}_n|\boldsymbol{y}_n; \boldsymbol{\theta}^t)$; this will yield a non-tight lower-bound on the log-likelihood. This generalized version of EM is known as variational EM [NH98b]. See Section 6.6.6.1 for details.

6.6.3.3 M step

In the M step, we need to maximize $\text{Ł}(\boldsymbol{\theta}, \{q_n^t\})$ wrt $\boldsymbol{\theta}$, where the q_n^t are the distributions computed in the E step at iteration t. Since the entropy terms $\mathbb{H}(q_n)$ are constant wrt $\boldsymbol{\theta}$, so we can drop them in the M step. We are left with

$$\ell^t(\boldsymbol{\theta}) = \sum_n \mathbb{E}_{q_n^t(\boldsymbol{z}_n)}\left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})\right] \tag{6.142}$$

This is called the **expected complete data log likelihood**. If the joint probability is in the exponential family (Section 2.3), we can rewrite this as

$$\ell^t(\boldsymbol{\theta}) = \sum_n \mathbb{E}\left[\mathcal{T}(\boldsymbol{y}_n, \boldsymbol{z}_n)^\mathsf{T}\boldsymbol{\theta} - A(\boldsymbol{\theta})\right] = \sum_n (\mathbb{E}\left[\mathcal{T}(\boldsymbol{y}_n, \boldsymbol{z}_n)\right]^\mathsf{T}\boldsymbol{\theta} - A(\boldsymbol{\theta})) \tag{6.143}$$

where $\mathbb{E}\left[\mathcal{T}(\boldsymbol{y}_n, \boldsymbol{z}_n)\right]$ are called the **expected sufficient statistics**.

In the M step, we maximize the expected complete data log likelihood to get

$$\boldsymbol{\theta}^{t+1} = \arg\max_{\boldsymbol{\theta}} \sum_n \mathbb{E}_{q_n^t}\left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})\right] \tag{6.144}$$

In the case of the exponential family, the maximization can be solved in closed-form by matching the moments of the expected sufficient statistics (Section 2.3.5).

We see from the above that the E step does not in fact need to return the full set of posterior distributions $\{q(\boldsymbol{z}_n)\}$, but can instead just return the sum of the expected sufficient statistics, $\sum_n \mathbb{E}_{q(\boldsymbol{z}_n)}\left[\mathcal{T}(\boldsymbol{y}_n, \boldsymbol{z}_n)\right]$.

A common application of EM is for fitting mixture models; we discuss this in the prequel to this book, [Mur22]. Below we give a different example.

6.6.4 Example: EM for an MVN with missing data

It is easy to compute the MLE for a multivariate normal when we have a fully observed data matrix: we just compute the sample mean and covariance. In this section, we consider the case where we have **missing data** or **partially observed data**. For example, we can think of the entries of $\mathbf{Y}$ as being answers to a survey; some of these answers may be unknown. There are many kinds of missing data, as we discuss in Section 21.3.4. In this section, we make the missing at random (MAR) assumption, for simplicity. Under the MAR assumption, the log likelihood of the visible data has the form

$$\log p(\mathbf{X}|\boldsymbol{\theta}) = \sum_n \log p(\boldsymbol{x}_n|\boldsymbol{\theta}) = \sum_n \log\left[\int p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\theta}) d\boldsymbol{z}_n\right] \tag{6.145}$$

where $\boldsymbol{x}_n$ are the visible variables in case n, $\boldsymbol{z}_n$ are the hidden variables, and $\boldsymbol{y}_n = (\boldsymbol{z}_n, \boldsymbol{x}_n)$ are all the variables. Unfortunately, this objective is hard to maximize. since we cannot push the log inside the expectation. Fortunately, we can easily apply EM, as we explain below.

6.6.4.1 E step

Suppose we have the parameters $\boldsymbol{\theta}^{t-1}$ from the previous iteration. Then we can compute the expected complete data log likelihood at iteration t as follows:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{t-1}) = \mathbb{E}\left[\sum_{n=1}^{N_{\mathcal{D}}} \log \mathcal{N}(\boldsymbol{y}_n|\boldsymbol{\mu}, \boldsymbol{\Sigma})|\mathcal{D}, \boldsymbol{\theta}^{t-1}\right] \tag{6.146}$$

$$= -\frac{N_{\mathcal{D}}}{2}\log|2\pi\boldsymbol{\Sigma}| - \frac{1}{2}\sum_n \mathbb{E}\left[(\boldsymbol{y}_n - \boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{y}_n - \boldsymbol{\mu})\right] \tag{6.147}$$

$$= -\frac{N_{\mathcal{D}}}{2}\log|2\pi\boldsymbol{\Sigma}| - \frac{1}{2}\mathrm{tr}(\boldsymbol{\Sigma}^{-1}\sum_n \mathbb{E}\left[(\boldsymbol{y}_n - \boldsymbol{\mu})(\boldsymbol{y}_n - \boldsymbol{\mu})^{\mathsf{T}}\right] \tag{6.148}$$

$$= -\frac{N_{\mathcal{D}}}{2}\log|\boldsymbol{\Sigma}| - \frac{N_{\mathcal{D}}D}{2}\log(2\pi) - \frac{1}{2}\mathrm{tr}(\boldsymbol{\Sigma}^{-1}\mathbb{E}\left[\mathbf{S}(\boldsymbol{\mu})\right]) \tag{6.149}$$

where

$$\mathbb{E}\left[\mathbf{S}(\boldsymbol{\mu})\right] \triangleq \sum_n \left(\mathbb{E}\left[\boldsymbol{y}_n\boldsymbol{y}_n^{\mathsf{T}}\right] + \boldsymbol{\mu}\boldsymbol{\mu}^{\mathsf{T}} - 2\boldsymbol{\mu}\mathbb{E}\left[\boldsymbol{y}_n\right]^{\mathsf{T}}\right) \tag{6.150}$$

(We drop the conditioning of the expectation on $\mathcal{D}$ and $\boldsymbol{\theta}^{t-1}$ for brevity.) We see that we need to compute $\sum_n \mathbb{E}[\boldsymbol{y}_n]$ and $\sum_n \mathbb{E}[\boldsymbol{y}_n \boldsymbol{y}_n^\mathsf{T}]$; these are the expected sufficient statistics.

To compute these quantities, we use the results from Section 2.2.5.3. We have

$$p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{z}_n|\boldsymbol{m}_n, \mathbf{V}_n) \tag{6.151}$$

$$\boldsymbol{m}_n \triangleq \boldsymbol{\mu}_h + \boldsymbol{\Sigma}_{hv}\boldsymbol{\Sigma}_{vv}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}_v) \tag{6.152}$$

$$\mathbf{V}_n \triangleq \boldsymbol{\Sigma}_{hh} - \boldsymbol{\Sigma}_{hv}\boldsymbol{\Sigma}_{vv}^{-1}\boldsymbol{\Sigma}_{vh} \tag{6.153}$$

where we partition $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ into blocks based on the hidden and visible indices h and v. Hence the expected sufficient statistics are

$$\mathbb{E}[\boldsymbol{y}_n] = (\mathbb{E}[\boldsymbol{z}_n]; \boldsymbol{x}_n) = (\boldsymbol{m}_n; \boldsymbol{x}_n) \tag{6.154}$$

To compute $\mathbb{E}[\boldsymbol{y}_n \boldsymbol{y}_n^\mathsf{T}]$, we use the result that $\mathrm{Cov}[\boldsymbol{y}] = \mathbb{E}[\boldsymbol{y}\boldsymbol{y}^\mathsf{T}] - \mathbb{E}[\boldsymbol{y}]\mathbb{E}[\boldsymbol{y}^\mathsf{T}]$. Hence

$$\mathbb{E}[\boldsymbol{y}_n \boldsymbol{y}_n^\mathsf{T}] = \mathbb{E}\left[\begin{pmatrix}\boldsymbol{z}_n \\ \boldsymbol{x}_n\end{pmatrix}\begin{pmatrix}\boldsymbol{z}_n^\mathsf{T} & \boldsymbol{x}_n^\mathsf{T}\end{pmatrix}\right] = \begin{pmatrix}\mathbb{E}[\boldsymbol{z}_n \boldsymbol{z}_n^\mathsf{T}] & \mathbb{E}[\boldsymbol{z}_n]\boldsymbol{x}_n^\mathsf{T} \\ \boldsymbol{x}_n \mathbb{E}[\boldsymbol{z}_n]^\mathsf{T} & \boldsymbol{x}_n \boldsymbol{x}_n^\mathsf{T}\end{pmatrix} \tag{6.155}$$

$$\mathbb{E}[\boldsymbol{z}_n \boldsymbol{z}_n^\mathsf{T}] = \mathbb{E}[\boldsymbol{z}_n]\mathbb{E}[\boldsymbol{z}_n]^\mathsf{T} + \mathbf{V}_n \tag{6.156}$$

6.6.4.2 M step

By solving $\nabla Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) = \mathbf{0}$, we can show that the M step is equivalent to plugging these ESS into the usual MLE equations to get

$$\boldsymbol{\mu}^t = \frac{1}{N_\mathcal{D}} \sum_n \mathbb{E}[\boldsymbol{y}_n] \tag{6.157}$$

$$\boldsymbol{\Sigma}^t = \frac{1}{N_\mathcal{D}} \sum_n \mathbb{E}[\boldsymbol{y}_n \boldsymbol{y}_n^\mathsf{T}] - \boldsymbol{\mu}^t (\boldsymbol{\mu}^t)^\mathsf{T} \tag{6.158}$$

Thus we see that EM is *not* equivalent to simply replacing variables by their expectations and applying the standard MLE formula; that would ignore the posterior variance and would result in an incorrect estimate. Instead we must compute the expectation of the sufficient statistics, and plug that into the usual equation for the MLE.

6.6.4.3 Initialization

To get the algorithm started, we can compute the MLE based on those rows of the data matrix that are fully observed. If there are no such rows, we can just estimate the diagonal terms of $\boldsymbol{\Sigma}$ using the observed marginal statistics. We are then ready to start EM.

6.6.4.4 Example

As an example of this procedure in action, let us consider an imputation problem, where we have $N_\mathcal{D} = 100$ 10-dimensional data cases, which we assume to come from a Gaussian. We generate synthetic data where 50% of the observations are missing at random. First we fit the parameters

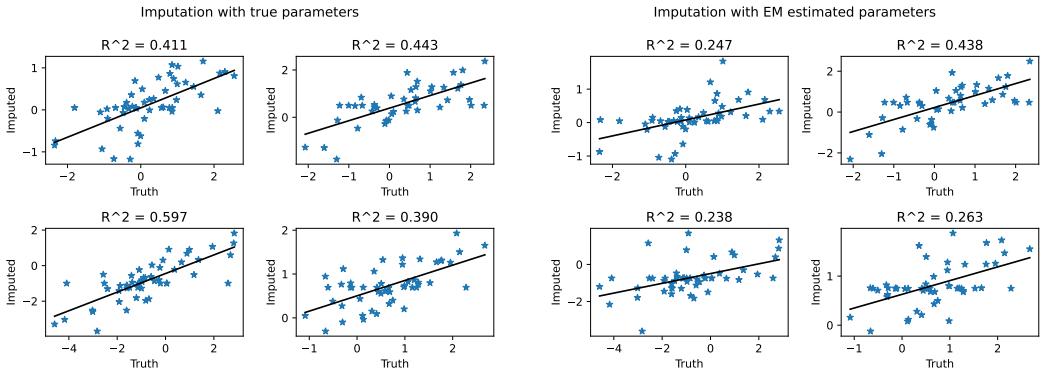


Figure 6.7: Illustration of data imputation using a multivariate Gaussian. (a) Scatter plot of true values vs imputed values using true parameters. (b) Same as (a), but using parameters estimated with EM. We just show the first four variables, for brevity. Generated by gauss_imputation_em_demo.ipynb.

using EM. Call the resulting parameters $\hat{\boldsymbol{\theta}}$. We can now use our model for predictions by computing $\mathbb{E}\left[\boldsymbol{z}_n|\boldsymbol{x}_n, \hat{\boldsymbol{\theta}}\right]$. Figure 6.7 indicates that the results obtained using the learned parameters are almost as good as with the true parameters. Not surprisingly, performance improves with more data, or as the fraction of missing data is reduced.

6.6.5 Example: robust linear regression using Student-t likelihood

In this section, we discuss how to use EM to fit a linear regression model that uses the Student distribution for its likelihood, instead of the more common Gaussian distribution, in order to achieve robustness, as first proposed in [Zel76]. More precisely, the likelihood is given by

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2, \nu) = \mathcal{T}(y|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}, \sigma^2, \nu) \tag{6.159}$$

At first blush it may not be apparent how to do this, since there is no missing data, and there are no hidden variables. However, it turns out that we can introduce "artificial" hidden variables to make the problem easier to solve; this is a common trick. The key insight is that we can represent the Student distribution as a Gaussian scale mixture, as we discussed in Section 28.2.3.1.

We can apply the GSM version of the Student distribution to our problem by associating a latent scale $z_n \in \mathbb{R}_+$ with each example. The complete data log likelihood is therefore given by

$$\log p(\boldsymbol{y}, \boldsymbol{z}|\mathbf{X}, \boldsymbol{w}, \sigma^2, \nu) = \sum_n -\frac{1}{2}\log(2\pi z_n\sigma^2) - \frac{1}{2z_n\sigma^2}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2 \tag{6.160}$$

$$+ (\frac{\nu}{2} - 1)\log(z_n) - z_n\frac{\nu}{2} + \text{const} \tag{6.161}$$

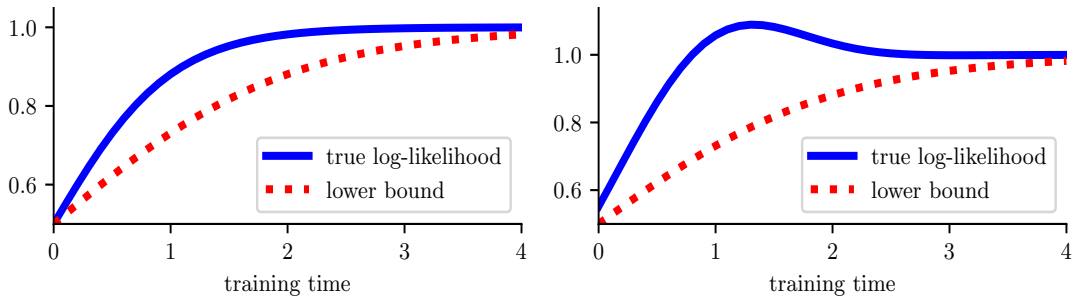


Figure 6.8: Illustration of possible behaviors of variational EM. (a) The lower bound increases at each iteration, and so does the likelihood. (b) The lower bound increases but the likelihood decreases. In this case, the algorithm is closing the gap between the approximate and true posterior. This can have a regularizing effect. Adapted from Figure 6 of [SJJ96]. Generated by var_em_bound.ipynb.

Ignoring terms not involving $\boldsymbol{w}$, and taking expectations, we have

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = -\sum_n \frac{\lambda_n}{2\sigma^2}(y_n - \boldsymbol{w}^T \boldsymbol{x}_n)^2 \tag{6.162}$$

where $\lambda_n^t \triangleq \mathbb{E}\left[1/z_n | y_n, \boldsymbol{x}_n, \boldsymbol{w}^t\right]$. We recognize this as a weighted least squares objective, with weight λ_n^t per data point .

We now discuss how to compute these weights. Using the results from Section 2.2.3.4, one can show that

$$p(z_n | y_n, \boldsymbol{x}_n, \boldsymbol{\theta}) = \mathrm{IG}(\frac{\nu+1}{2}, \frac{\nu+\delta_n}{2}) \tag{6.163}$$

where $\delta_n = \frac{(y_n - \boldsymbol{x}^T \boldsymbol{x}_n)^2}{\sigma^2}$ is the standardized residual. Hence

$$\lambda_n = \mathbb{E}\left[1/z_n\right] = \frac{\nu^t + 1}{\nu^t + \delta_n^t} \tag{6.164}$$

So if the residual δ_n^t is large, the point will be given low weight λ_n^t, which makes intuitive sense, since it is probably an outlier.

6.6.6 Extensions to EM

There are many variations and extensions of the EM algorithm, as discussed in [MK97]. We summarize a few of these below.

6.6.6.1 Variational EM

Suppose in the E step we pick $q_n^* = \mathrm{argmin}_{q_n \in \mathcal{Q}} D_{\mathbb{KL}}\left(q_n \parallel p(\boldsymbol{z}_n | \boldsymbol{x}_n, \boldsymbol{\theta})\right)$. Because we are optimizing over the space of functions, this is called variational inference (see Section 10.1 for details). If the

family of distributions $\mathcal{Q}$ is rich enough to contain the true posterior, $q_n = p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta})$, then we can make the KL be zero. But in general, we might choose a more restrictive class for computational reasons. For example, we might use $q_n(\boldsymbol{z}_n) = \mathcal{N}(\boldsymbol{z}_n|\boldsymbol{\mu}_n, \text{diag}(\boldsymbol{\sigma}_n))$ even if the true posterior is correlated.

The use of a restricted posterior family $\mathcal{Q}$ inside the E step of EM is called **variational EM** [NH98a]. Unlike regular EM, variational EM is not guaranteed to increase the actual log likelihood itself (see Figure 6.8), but it does monotonically increase the variational lower bound. We can control the tightness of this lower bound by varying the variational family $\mathcal{Q}$; in the limit in which $q_n = p_n$, corresponding to exact inference, we recover the same behavior as regular EM. See Section 10.2.5 for further discussion.

6.6.6.2 Hard EM

Suppose we use a degenerate posterior approximation in the context of variational EM, corresponding to a point estimate, $q(\boldsymbol{z}|\boldsymbol{x}_n) = \delta_{\hat{\boldsymbol{z}}_n}(\boldsymbol{z})$, where $\hat{\boldsymbol{z}}_n = \text{argmax}_{\boldsymbol{z}}\, p(\boldsymbol{z}|\boldsymbol{x}_n)$. This is equivalent to **hard EM**, where we ignore uncertainty about $\boldsymbol{z}_n$ in the E step.

The problem with this degenerate approach is that it is very prone to overfitting, since the number of latent variables is proportional to the number of datacases [WCS08].

6.6.6.3 Monte Carlo EM

Another approach to handling an intractable E step is to use a Monte Carlo approximation to the expected sufficient statistics. That is, we draw samples from the posterior, $\boldsymbol{z}_n^s \sim p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta}^t)$, and then compute the sufficient statistics for each completed vector, $(\boldsymbol{x}_n, \boldsymbol{z}_n^s)$, and then average the results. This is called **Monte Carlo EM** or **MCEM** [WT90; Nea12].

One way to draw samples is to use MCMC (see Chapter 12). However, if we have to wait for MCMC to converge inside each E step, the method becomes very slow. An alternative is to use stochastic approximation, and only perform "brief" sampling in the E step, followed by a partial parameter update. This is called **stochastic approximation EM** [DLM99] and tends to work better than MCEM.

6.6.6.4 Generalized EM

Sometimes we can perform the E step exactly, but we cannot perform the M step exactly. However, we can still monotonically increase the log likelihood by performing a "partial" M step, in which we merely increase the expected complete data log likelihood, rather than maximizing it. For example, we might follow a few gradient steps. This is called the **generalized EM** or **GEM** algorithm [MK07]. (This is an unfortunate term, since there are many ways to generalize EM, but it is the standard terminology.) For example, [Lan95a] proposes to perform one Newton-Raphson step:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{H}_t^{-1} \boldsymbol{g}_t \tag{6.165}$$

where $0 < \eta_t \leq 1$ is the step size, and

$$\boldsymbol{g}_t = \frac{\partial}{\partial \boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_t)|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \tag{6.166}$$

$$\mathbf{H}_t = \frac{\partial^2}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^\mathsf{T}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_t)|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \tag{6.167}$$

If $\eta_t = 1$, [Lan95a] cals this the **gradient EM algorithm**. However, it is possible to use a larger step size to speed up the algorithm, as in the **quasi-Newton EM algorithm** of [Lan95b]. This method also replaces the Hessian in Equation (6.167), which may not be negative definite (for non exponential family models), with a BFGS approximation. This ensures the overall algorithm is an ascent algorithm. Note, however, when the M step cannot be computed in closed form, EM loses some of its appeal over directly optimizing the marginal likelihood with a gradient based solver.

6.6.6.5 ECM algorithm

The **ECM** algorithm stands for "expectation conditional maximization", and refers to optimizing the parameters in the M step sequentially, if they turn out to be dependent. The **ECME** algorithm, which stands for "ECM either" [LR95], is a variant of ECM in which we maximize the expected complete data log likelihood (the Q function) as usual, or the observed data log likelihood, during one or more of the conditional maximization steps. The latter can be much faster, since it ignores the results of the E step, and directly optimizes the objective of interest. A standard example of this is when fitting the Student T distribution. For fixed ν, we can update $\boldsymbol{\Sigma}$ as usual, but then to update ν, we replace the standard update of the form $\nu^{t+1} = \arg\max_\nu Q((\boldsymbol{\mu}^{t+1}, \boldsymbol{\Sigma}^{t+1}, \nu), \boldsymbol{\theta}^t)$ with $\nu^{t+1} = \arg\max_\nu \log p(\mathcal{D}|\boldsymbol{\mu}^{t+1}, \boldsymbol{\Sigma}^{t+1}, \nu)$. See [MK97] for more information.

6.6.6.6 Online EM

When dealing with large or streaming datasets, it is important to be able to learn online, as we discussed in Section 19.7.5. There are two main approaches to **online EM** in the literature. The first approach, known as **incremental EM** [NH98a], optimizes the lower bound $Q(\boldsymbol{\theta}, q_1, \ldots, q_N)$ one q_n at a time; however, this requires storing the expected sufficient statistics for each data case.

The second approach, known as **stepwise EM** [SI00; LK09; CM09], is based on stochastic gradient descent. This optimizes a local upper bound on $\ell_n(\boldsymbol{\theta}) = \log p(\boldsymbol{x}_n|\boldsymbol{\theta})$ at each step. (See [Mai13; Mai15] for a more general discussion of stochastic and incremental bound optimization algorithms.)

6.7 The Bayesian learning rule

in this section, we discuss the "**Bayesian learning rule**" [KR21a], which provides a unified framework for deriving many standard (and non-standard) optimization and inference algorithms used in the ML community.

To motivate the BLR, recall the standard **empirical risk minimzation** or **ERM** problem, which has the form $\boldsymbol{\theta}_* = \text{argmin}_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta})$, where

$$\bar{\ell}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) + R(\boldsymbol{\theta}) \tag{6.168}$$

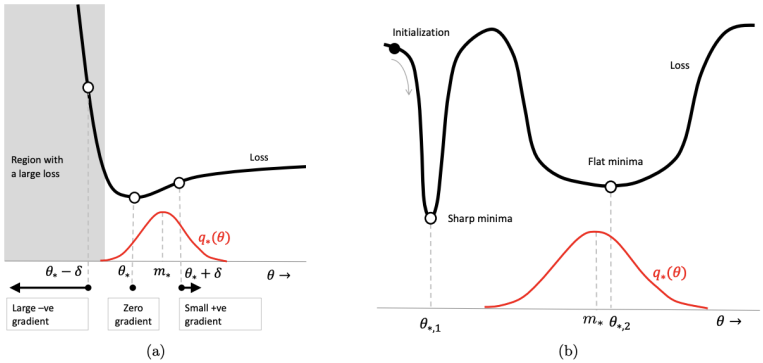


Figure 6.9: Illustration of the robustness obtained by using a Bayesian approach to parameter estimation. (a) When the minimum $\boldsymbol{\theta}_*$ lies next to a "wall", the Bayesian solution shifts away from the boundary to avoid large losses due to perturbations of the parameters. (b) The Bayesian solution prefers flat minima over sharp minima, to avoid large losses due to perturbations of the parameters. From Figure 1 of [KR21a]. Used with kind permission of Emtiyaz Khan.

where $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ is a prediction function, $\ell(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is a loss function, and $R(\boldsymbol{\theta})$ is some kind of regularizer.

Although the regularizer can prevent overfitting, the ERM method can still result in parameter estimates that are not robust. A better approach is to fit a *distribution* over possible parameter values, $q(\boldsymbol{\theta})$. If we minimize the expected loss, we will find parameter settings that will work well even if they are slightly perturbed, as illustrated in Figure 6.9, which helps with robustness and generalization. Of course, if the distribution q collapses to a single delta function, we will end up with the ERM solution. To prevent this, we add a penalty term, that measures the KL divergence from $q(\boldsymbol{\theta})$ to some prior $\pi_0(\boldsymbol{\theta}) \propto \exp(R(\boldsymbol{\theta}))$. This gives rise to the following BLR objective:

$$\mathcal{L}(q) = \mathbb{E}_{q(\boldsymbol{\theta})}\left[\sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right] + D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}) \parallel \pi_0(\boldsymbol{\theta})\right) \tag{6.169}$$

We can rewrite the KL term as

$$D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}) \parallel \pi_0(\boldsymbol{\theta})\right) = \mathbb{E}_{q(\boldsymbol{\theta})}\left[R(\boldsymbol{\theta})\right] - \mathbb{H}(q(\boldsymbol{\theta})) \tag{6.170}$$

and hence can rewrite the BLR objective as follows:

$$\mathcal{L}(q) = \mathbb{E}_{q(\boldsymbol{\theta})}\left[\bar{\ell}(\boldsymbol{\theta})\right] - \mathbb{H}(q(\boldsymbol{\theta})) \tag{6.171}$$

Below we show that different approximations to this objective recover a variety of different methods in the literature.

6.7.1 Deriving inference algorithms from BLR

In this section we show how to derive several different inference algorithms from BLR. (We discuss such algorithms in more detail in Chapter 10.)

6.7.1.1 Bayesian inference as optimization

The BLR objective includes standard exact Bayesian inference as a special case, as first shown in [Opt88]. To see this, let us assume the loss function is derived from a log-likelihood:

$$\ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})) = -\log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x})) \tag{6.172}$$

Let $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) : n = 1 : N\}$ be the data we condition on. The Bayesian posterior can be written as

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z(\mathcal{D})}\pi_0(\boldsymbol{\theta}) \prod_{n=1}^{N} p(\boldsymbol{y}_n|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \tag{6.173}$$

This can be derived by minimizing the BLR, since

$$\mathcal{L}(q) = -\mathbb{E}_{q(\boldsymbol{\theta})}\left[\sum_{n=1}^{N} \log p(\boldsymbol{y}_n|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right] + D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}) \parallel \pi_0(\boldsymbol{\theta})\right) \tag{6.174}$$

$$= \mathbb{E}_{q(\boldsymbol{\theta})}\left[\log \frac{q(\boldsymbol{\theta})}{\frac{\pi_0(\boldsymbol{\theta})}{Z(\mathcal{D})}\prod_{n=1}^{N} p(\boldsymbol{y}_n|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}\right] - \log Z(\mathcal{D}) \tag{6.175}$$

$$= D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}) \parallel p(\boldsymbol{\theta}|\mathcal{D})\right) - \log Z(\mathcal{D}) \tag{6.176}$$

Since $Z(\mathcal{D})$ is a constant, we can minimize the loss by by setting $q(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D})$.

Of course, we can use other kinds of loss, not just log likelihoods. This results in a framework known as **generalized Bayesian inference** [BHW16; KJD19; KJD21]. See Section 14.1.3 for more discussion.

6.7.1.2 Optimization of BLR using natural gradient descent

In general, we cannot compute the exact posterior $q(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D})$, so we seek an approximation. We will assume that $q(\boldsymbol{\theta})$ is an exponential family distibution, such as a multivariate Gaussian, where the mean represents the standard point estimate of $\boldsymbol{\theta}$ (as in ERM), and the covariance represents our uncertainty (as in Bayes). Hence q can be written as follows:

$$q(\boldsymbol{\theta}) = h(\boldsymbol{\theta}) \exp[\boldsymbol{\lambda}^\mathsf{T}\mathcal{T}(\boldsymbol{\theta}) - A(\boldsymbol{\lambda})] \tag{6.177}$$

where $\boldsymbol{\lambda}$ are the natural parameters, $\mathcal{T}(\boldsymbol{\theta})$ are the sufficient statistics, $A(\boldsymbol{\lambda})$ is the log partition function, and $h(\boldsymbol{\theta})$ is the base measure, which is usually a constant. The BLR loss becomes

$$\mathcal{L}(\boldsymbol{\lambda}) = \mathbb{E}_{q_{\boldsymbol{\lambda}}(\boldsymbol{\theta})}\left[\overline{\ell}(\boldsymbol{\theta})\right] - \mathbb{H}(q_{\boldsymbol{\lambda}}(\boldsymbol{\theta})) \tag{6.178}$$

We can optimize this using natural gradient descent (Section 6.4). The update becomes

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t - \eta_t \tilde{\nabla}_{\boldsymbol{\lambda}}\left[\mathbb{E}_{q_{\boldsymbol{\lambda}_t}(\boldsymbol{\theta})}\left[\overline{\ell}(\boldsymbol{\theta})\right] - \mathbb{H}(q_{\boldsymbol{\lambda}_t})\right] \tag{6.179}$$

where $\tilde{\nabla}_{\boldsymbol{\lambda}}$ denotes the natural gradient. We discuss how to compute these natural gradients in Section 6.4.5. In particular, we can convert it to regular gradients wrt the moment parameters $\boldsymbol{\mu}_t = \boldsymbol{\mu}(\boldsymbol{\lambda}_t)$. This gives

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t - \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}_t}(\boldsymbol{\theta})}\left[\overline{\ell}(\boldsymbol{\theta})\right] + \eta_t \nabla_{\boldsymbol{\mu}}\, \mathbb{H}(q_{\boldsymbol{\mu}_t}) \tag{6.180}$$

From Equation (6.84) we have

$$\nabla_{\boldsymbol{\mu}} \mathbb{H}(q) = -\boldsymbol{\lambda} - \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\log h(\boldsymbol{\theta})\right] \tag{6.181}$$

Hence the update becomes

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t - \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}_t}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] - \eta_t \boldsymbol{\lambda}_t - \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\log h(\boldsymbol{\theta})\right] \tag{6.182}$$

$$= (1 - \eta_t)\boldsymbol{\lambda}_t - \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta}) + \log h(\boldsymbol{\theta})\right] \tag{6.183}$$

For distributions q with constant base measure $h(\boldsymbol{\theta})$, this simplifies to

$$\boldsymbol{\lambda}_{t+1} = (1 - \eta_t)\boldsymbol{\lambda}_t - \eta_t \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.184}$$

Hence at the fixed point we have

$$\boldsymbol{\lambda}_* = (1 - \eta)\boldsymbol{\lambda}_* - \eta \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.185}$$

$$\boldsymbol{\lambda}_* = \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[-\overline{\ell}(\boldsymbol{\theta})\right] = \tilde{\nabla}_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\boldsymbol{\theta})} \left[-\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.186}$$

6.7.1.3 Conjugate variational inference

In Section 7.3 we showed how to do exact inference in conjugate models. We can derive Equation (7.11) from the BLR by using the fixed point condition in Equation (6.186) to write

$$\boldsymbol{\lambda}_* = \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_*} \left[-\overline{\ell}(\boldsymbol{\theta})\right] = \boldsymbol{\lambda}_0 + \sum_{i=1}^{N} \underbrace{\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q*} \left[\log p(\boldsymbol{y}_i|\boldsymbol{\theta})\right]}_{\tilde{\boldsymbol{\lambda}}_i(\boldsymbol{y}_i)} \tag{6.187}$$

where $\tilde{\boldsymbol{\lambda}}_i(\boldsymbol{y}_i)$ are the sufficient statistics for the i'th likelihood term.

For models where the joint distribution over the latents factorizes (using a graphical model), we can further decompose this update into a series of local terms. This gives rise to the variational message passing scheme discussed in Section 10.2.7.

6.7.1.4 Partially conjugate variational inference

In Supplementary Section 10.3.1, we discuss CVI, which performs variational inference for partially conjugate models, using gradient updates for the non-conjugate parts, and exact Bayesian inference for the conjugate parts.

6.7.2 Deriving optimization algorithms from BLR

In this section we show how to derive several different optimization algorithms from BLR. Recall that in BLR, instead of directly minimizing the loss

$$\overline{\ell}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) + R(\boldsymbol{\theta}) \tag{6.188}$$

we will instead minimize

$$\mathcal{L}(\boldsymbol{\lambda}) = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})} \left[\overline{\ell}(\boldsymbol{\theta})\right] - \mathbb{H}(q(\boldsymbol{\theta}|\boldsymbol{\lambda})) \tag{6.189}$$

Below we show that different approximations to this objective recover a variety of different optimization methods that are used in the literature. (We discuss such algorithms in more detail in Chapter 6.)

6.7.2.1 Gradient descent

In this section, we show how to derive gradient descent as a special case of BLR. We use as our approximate posterior $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{m}, \mathbf{I})$. In this case the natural and moment parameters are equal, $\boldsymbol{\mu} = \boldsymbol{\lambda} = \boldsymbol{m}$. The base measure satisfies the following (from Equation (2.180)):

$$2 \log h(\boldsymbol{\theta}) = -D \log(2\pi) - \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\theta} \tag{6.190}$$

Hence

$$\nabla_{\boldsymbol{\mu}} \mathbb{E}_q \left[\log h(\boldsymbol{\theta})\right] = \nabla_{\boldsymbol{\mu}}(-D \log(2\pi) - \boldsymbol{\mu}^\mathsf{T} \boldsymbol{\mu} - D) = -\boldsymbol{\mu} = -\boldsymbol{\lambda} = -\boldsymbol{m} \tag{6.191}$$

Thus from Equation (6.183) the BLR update becomes

$$\boldsymbol{m}_{t+1} = (1 - \eta_t)\boldsymbol{m}_t + \eta_t \boldsymbol{m}_t - \eta_t \nabla_{\boldsymbol{m}} \mathbb{E}_{q_{\boldsymbol{m}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.192}$$

We can remove the expectation using the first order delta method (Section 6.5.5):

$$\nabla_{\boldsymbol{m}} \mathbb{E}_{q_{\boldsymbol{m}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \approx \nabla_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}} \tag{6.193}$$

Putting these together gives the gradient descent update:

$$\boldsymbol{m}_{t+1} = \boldsymbol{m}_t - \eta_t \nabla_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}_t} \tag{6.194}$$

6.7.2.2 Newton's method

In this section, we show how to derive Newton's second order optimization method as a special case of BLR, as first shown in [Kha+18].

Suppose we assume $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{m}, \mathbf{S}^{-1})$. The natural parameters are

$$\boldsymbol{\lambda}^{(1)} = \mathbf{S}\boldsymbol{m}, \; \boldsymbol{\lambda}^{(2)} = -\frac{1}{2}\mathbf{S} \tag{6.195}$$

The mean (moment) parameters are

$$\boldsymbol{\mu}^{(1)} = \boldsymbol{m}, \; \boldsymbol{\mu}^{(2)} = \mathbf{S}^{-1} + \boldsymbol{m}\boldsymbol{m}^\mathsf{T} \tag{6.196}$$

Since the base measure is constant (see Equation (2.193)), from Equation (6.184) we have

$$\mathbf{S}_{t+1}\boldsymbol{m}_{t+1} = (1 - \eta_t)\mathbf{S}_t \boldsymbol{m}_t - \eta_t \nabla_{\boldsymbol{\mu}^{(1)}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.197}$$

$$\mathbf{S}_{t+1} = (1 - \eta_t)\mathbf{S}_t + 2\eta_t \nabla_{\boldsymbol{\mu}^{(2)}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] \tag{6.198}$$

In Section 6.4.5.1 we show that

$$\nabla_{\boldsymbol{\mu}^{(1)}} \mathbb{E}_{q(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] = \mathbb{E}_{q(\boldsymbol{\theta})} \left[\nabla_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})\right] - \mathbb{E}_{q(\boldsymbol{\theta})} \left[\nabla^2_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})\right] \boldsymbol{m} \tag{6.199}$$

$$\nabla_{\boldsymbol{\mu}^{(2)}} \mathbb{E}_{q(\boldsymbol{\theta})} \left[\overline{\ell}(\boldsymbol{\theta})\right] = \frac{1}{2} \mathbb{E}_{q(\boldsymbol{\theta})} \left[\nabla^2_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})\right] \tag{6.200}$$

Hence the update for the precision matrix becomes

$$\mathbf{S}_{t+1} = (1 - \eta_t)\mathbf{S}_t + \eta_t \mathbb{E}_{q_t} \left[\nabla^2_{\boldsymbol{\theta}} \overline{\ell}(\boldsymbol{\theta})\right] \tag{6.201}$$

For the precision weighted mean, we have

$$\mathbf{S}_{t+1}\boldsymbol{m}_{t+1} = (1-\eta_t)\mathbf{S}_t\boldsymbol{m}_t - \eta_t\mathbb{E}_{q_t}\left[\nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] + \eta_t\mathbb{E}_{q_t}\left[\nabla^2_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right]\boldsymbol{m}_t \tag{6.202}$$

$$= \mathbf{S}_{t+1}\boldsymbol{m}_t - \eta_t\mathbb{E}_{q_t}\left[\nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] \tag{6.203}$$

Hence

$$\boldsymbol{m}_{t+1} = \boldsymbol{m}_t - \eta_t\mathbf{S}^{-1}_{t+1}\mathbb{E}_{q_t}\left[\nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] \tag{6.204}$$

We can recover Newton's method in three steps. First set the learning rate to $\eta_t = 1$, based on an assumption that the objective is convex. Second, treat the iterate as $\boldsymbol{m}_t = \boldsymbol{\theta}_t$. Third, apply the delta method to get

$$\mathbf{S}_{t+1} = \mathbb{E}_{q_t}\left[\nabla^2_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] \approx \nabla^2_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}_t} \tag{6.205}$$

and

$$\mathbb{E}_q\left[\nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] \approx \nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{m}_t} \tag{6.206}$$

This gives Newton's update:

$$\boldsymbol{m}_{t+1} = \boldsymbol{m}_t - [\nabla^2_{\boldsymbol{m}}\bar{\ell}(\boldsymbol{m}_t)]^{-1}[\nabla_{\boldsymbol{m}}\bar{\ell}(\boldsymbol{m}_t)] \tag{6.207}$$

6.7.2.3 Variational online Gauss-Newton

In this section, we describe the **Variational Online Gauss-Newton** or **VOGN** method of [Kha+18]. This is an approximate second order optimization method that can be derived from the BLR in several steps, as we show below.

First, we use a diagonal Gaussian approximation to the posterior, $q_t(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\theta}_t, \mathbf{S}_t^{-1})$, where $\mathbf{S}_t = \mathrm{diag}(\boldsymbol{s}_t)$ is a vector of precisions. Following Section 6.7.2.2, we get the following updates:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t\frac{1}{\boldsymbol{s}_{t+1}} \odot \mathbb{E}_{q_t}\left[\nabla_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta})\right] \tag{6.208}$$

$$\boldsymbol{s}_{t+1} = (1-\eta_t)\boldsymbol{s}_t + \eta_t\mathbb{E}_{q_t}\left[\mathrm{diag}(\nabla^2_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta}))\right] \tag{6.209}$$

where $\odot$ is elementwise multiplication, and the division by $\boldsymbol{s}_{t+1}$ is also elementwise.

Second, we use the delta approximation to replace expectations by plugging in the mean. Third we use a minibatch approximation to the gradient and diagonal Hessian:

$$\hat{\nabla}_{\boldsymbol{\theta}}\bar{\ell}(\boldsymbol{\theta}) = \frac{N}{M}\sum_{i\in\mathcal{M}}\nabla_{\boldsymbol{\theta}}\ell(y_i, f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)) + \nabla_{\boldsymbol{\theta}}R(\boldsymbol{\theta}) \tag{6.210}$$

$$\hat{\nabla}^2_{\theta_j}\bar{\ell}(\boldsymbol{\theta}) = \frac{N}{M}\sum_{i\in\mathcal{M}}\nabla^2_{\theta_j}\ell(y_i, f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)) + \nabla^2_{\theta_j}R(\boldsymbol{\theta}) \tag{6.211}$$

where M is the minibatch size.

For some non-convex problems, such as DNNs, the Hessian may be not be positive definite, so we can get better results using a Gauss-Newton approximation, based on the squared gradients instead of the Hessian:

$$\hat{\nabla}^2_{\theta_j} \bar{\ell}(\boldsymbol{\theta}) \approx \frac{N}{M} \sum_{i \in \mathcal{M}} [\nabla_{\theta_j} \ell(y_i, f_{\boldsymbol{\theta}}(\boldsymbol{x}_i))]^2 + \nabla^2_{\theta_j} R(\boldsymbol{\theta}) \tag{6.212}$$

This is also faster to compute.

Putting all this together gives rise to the **Online Gauss-Newton** or **OGN** method of [Osa+19a].

If we drop the delta approximation, and work with expectations. we get the **Variational Online Gauss-Newton** or **VOGN** method of [Kha+18]. We can approximate the expectations by sampling. In particular, VOGN uses the following **weight perturbation** method

$$\mathbb{E}_{q_t}\left[\hat{\nabla}_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta})\right] \approx \hat{\nabla}_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t + \boldsymbol{\epsilon}_t) \tag{6.213}$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \text{diag}(\boldsymbol{s}_t))$. It also also possible to approximate the Fisher information matrix directly; this results in the **Variational Online Generalized Gauss-Newton** or **VOGGN** method of [Osa+19a].

6.7.2.4 Adaptive learning rate SGD

In this section, we show how to derive an update rule which is very similar to the **RMSprop** [Hin14] method, which is widely used in deep learning. The approach we take is similar to that VOGN in Section 6.7.2.3. We use the same diagonal Gaussian approximation, $q_t(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\theta}_t, \mathbf{S}_t^{-1})$, where $\mathbf{S}_t = \text{diag}(\boldsymbol{s}_t)$ is a vector of precisions. We then use the delta method to eliminate expectations:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{1}{\boldsymbol{s}_{t+1}} \odot \nabla_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t) \tag{6.214}$$

$$\boldsymbol{s}_{t+1} = (1 - \eta_t)\boldsymbol{s}_t + \eta_t \text{diag}(\nabla^2_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t)) \tag{6.215}$$

where $\odot$ is elementwise multiplication. If we allow for different learning rates we get

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \frac{1}{\boldsymbol{s}_{t+1}} \odot \nabla_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t) \tag{6.216}$$

$$\boldsymbol{s}_{t+1} = (1 - \beta_t)\boldsymbol{s}_t + \beta_t \text{diag}(\hat{\nabla}^2_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t)) \tag{6.217}$$

Now suppose we replace the diagonal Hessian approximation with the sum of the squares per-sample gradients:

$$\text{diag}(\nabla^2_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t)) \approx \hat{\nabla} \bar{\ell}(\boldsymbol{\theta}_t) \odot \hat{\nabla} \bar{\ell}(\boldsymbol{\theta}_t) \tag{6.218}$$

If we also change some scaling factors we can get the RMSprop updates:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{1}{\sqrt{\boldsymbol{v}_{t+1}} + c\mathbf{1}} \odot \hat{\nabla}_{\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{\theta}_t) \tag{6.219}$$

$$\boldsymbol{v}_{t+1} = (1 - \beta)\boldsymbol{v}_t + \beta[\hat{\nabla} \bar{\ell}(\boldsymbol{\theta}_t) \odot \hat{\nabla} \bar{\ell}(\boldsymbol{\theta}_t)] \tag{6.220}$$

This allows us to use standard deep learning optimizers to get a Gaussian approximation to the posterior for the parameters [Osa+19a].

It is also possible to derive the Adam optimizer [KB15] from BLR by adding a momentum term to RMSprop. See [KR21a; Ait18] for details.

6.7.3 Variational optimization

Consider an objective defined in terms of discrete variables. Such objectives are not differentiable and so are hard to optimize. One advantage of BLR is that it optimizes the parameters of a probability distribution, and such expected loss objectives are usually differentiable and smooth. This is called "**variational optimization**" [Bar17], since we are optimizing over a probability distribution.

For example, consider the case of a **binary neural network** where $\theta_d \in \{0, 1\}$ indicates if weight d is used or not, we can optimize over the parameters of a Bernoulli distribution, $q(\boldsymbol{\theta}|\boldsymbol{\lambda}) = \prod_{d=1}^{D} \text{Ber}(\theta_d|p_d)$, where $p_d \in [0, 1]$ and $\lambda_d = 1/2 \log(p_d/(1 - p_d)$ is the log odds. This is the basis of the **BayesBiNN** approach [MBK20].

If we ignore the entropy and regularizer term, we get the following simplified objective:

$$\mathcal{L}(\boldsymbol{\lambda}) = \int \bar{\ell}(\boldsymbol{\theta}) q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} \tag{6.221}$$

This method has various names: **stochastic relaxation** [SB12; SB13; MMP13], **stochastic approximation** [HHC12; Hu+12], etc. It is closely related to **evolutionary strategies**, which we discuss in Section 6.9.6.

In the case of functions with continuous domains, we can use a Gaussian for $q(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The resulting integral in Equation (6.221) can then sometimes be solved in closed form, as explained in [Mob16]. By starting with a broad variance, and gradually reducing it, we hope the method can avoid poor local optima, similar to simulated annealing (Section 12.9.1. However, we generally get better results by including the entropy term, because then we can automatically learn to adapt the variance. In addition, we can often work with natural gradients, which results in faster convergence.

6.8 Bayesian optimization

In this section, we discuss **Bayesian optimization** or **BayesOpt**, which is a model-based approach to black-box optimization, designed for the case where the objective function $f : \mathcal{X} \to \mathbb{R}$ is expensive to evaluate (e.g., if it requires running a simulation, or training and testing a particular neural net architecture).

Since the true function f is expensive to evaluate, we want to make as few function calls (i.e., make as few **queries** $\boldsymbol{x}$ to the **oracle** f) as possible. This suggests that we should build a **surrogate function** (also called a **response surface model**) based on the data collected so far, $\mathcal{D}_n = \{(\boldsymbol{x}_i, y_i) : i = 1 : n\}$, which we can use to decide which point to query next. There is an inherent tradeoff between picking the point $\boldsymbol{x}$ where we think $f(\boldsymbol{x})$ is large (we follow the convention in the literature and assume we are trying to maximize f), and picking points where we are uncertain about $f(\boldsymbol{x})$ but where observing the function value might help us improve the surrogate model. This is another instance of the exploration-exploitation dilemma.

In the special case where the domain we are optimizing over is finite, so $\mathcal{X} = \{1, \ldots, A\}$, the BayesOpt problem becomes similar to the **best arm identification** problem in the bandit literature (Section 34.4). An important difference is that in bandits, we care about the cost of every action we take, whereas in optimization, we usually only care about the cost of the final solution we find. In other words, in bandits, we want to minimize cumulative regret, whereas in optimization we want to minimize simple or final regret.

Another related topic is **active learning**. Here the goal is to identify the whole function f with as few queries as possible, whereas in BayesOpt, the goal is just to identify the maximum of the function.

Bayesian optimization is a large topic, and we only give a brief overview below. For more details, see e.g., [Sha+16; Fra18; Gar22]. (See also https://distill.pub/2020/bayesian-optimization/ for an interactive tutorial.)

6.8.1 Sequential model-based optimization

BayesOpt is an instance of a strategy known as sequential model-based optimization (**SMBO**) [HHLB11]. In this approach, we alternate between querying the function at a point, and updating our estimate of the surrogate based on the new data. More precisely, at each iteration n, we have a labeled dataset $\mathcal{D}_n = \{(\boldsymbol{x}_i, y_i) : i = 1 : n\}$, which records points $\boldsymbol{x}_i$ that we have queried, and the corresponding function values $y_i = f(\boldsymbol{x}_i) + \epsilon_i$, where ϵ_i is an optional noise term. We use this data to estimate a probability distribution over the true function f; we will denote this by $p(f|\mathcal{D}_n)$. We then choose the next point to query $\boldsymbol{x}_{n+1}$ using an **acquisition function** $\alpha(\boldsymbol{x}; \mathcal{D}_n)$, which computes the expected utility of querying $\boldsymbol{x}$. (We discuss acquisition functions in Section 6.8.3). After we observe $y_{n+1} = f(\boldsymbol{x}_{n+1}) + \epsilon_{n+1}$, we update our beliefs about the function, and repeat. See Algorithm 5 for some pseudocode.

Algorithm 5: Bayesian optimization

```
Collect initial dataset D_0 = {(x_i, y_i) :} from random queries x_i
Initialize model p(f|D_0)
for n = 1, 2, ... until convergence do
    Choose next query point x_{n+1} = argmax_{x∈X} α(x; D_n)
    Measure function value, y_{n+1} = f(x_{n+1}) + ε_n
    Augment dataset, D_{n+1} = {D_n, (x_{n+1}, y_{n+1})}
    Update model, p(f|D_{n+1}) ∝ p(f|D_n) p(y_{n+1}|x_{n+1}, f)
```

This method is illustrated in Figure 6.10. The goal is to find the global optimum of the solid black curve. In the first row, we show the 2 previously queried points, x_1 and x_2, and their corresponding function values. $y_1 = f(x_1)$ and $y_2 = f(x_2)$. Our uncertainty about the value of f at those locations is 0 (if we assume no observation noise), as illustrated by the posterior credible interval (shaded blue are) becoming "pinched". Consequently the acquisition function (shown in green at the bottom) also has value 0 at those previously queried points. The red triangle represents the maximum of the acquisition function, which becomes our next query, x_3. In the second row, we show the result of observing $y_3 = f(x_3)$; this further reduces our uncertainty about the shape of the function. In the third row, we show the result of observing $y_4 = f(x_4)$. This process repeats until we run out of time, or until we are confident there are no better unexplored points to query.

The two main "ingredients" that we need to provide to a BayesOpt algorithm are (1) a way to represent and update the posterior surrogate $p(f|\mathcal{D}_n)$, and (2) a way to define and optimize the acquisition function $\alpha(\boldsymbol{x}; \mathcal{D}_n)$. We discuss both of these topics below.

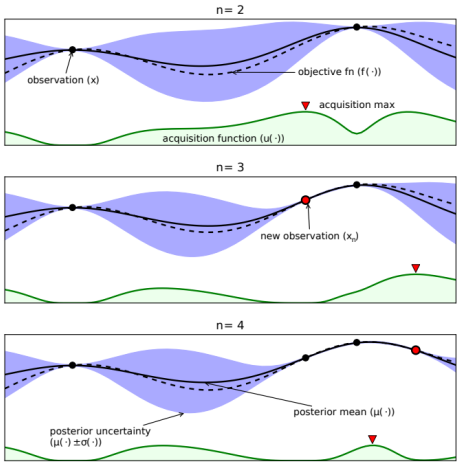


Figure 6.10: Illustration of sequential Bayesian optimization over three iterations. The rows correspond to a training set of size $t = 2, 3, 4$. The solid black line is the true, but unknown, function $f(x)$. The dotted black line is the posterior mean, $\mu(x)$. The shaded blue intervals are the 95% credible interval derived from $\mu(x)$ and $\sigma(x)$. The solid black dots correspond to points whose function value has already been computed, i.e., x_n for which $f(x_n)$ is known. The green curve at the bottom is the acquisition function. The red dot is the proposed next point to query, which is the maximum of the acquisition function. From Figure 1 of [Sha+16]. Used with kind permission of Nando de Freitas.

6.8.2 Surrogate functions

In this section, we discuss ways to represent and update the posterior over functions, $p(f|\mathcal{D}_n)$.

6.8.2.1 Gaussian processes

In BayesOpt, it is very common to use a Gaussian process or GP for our surrogate. GPs are explained in detail in Chapter 18, but the basic idea is that they represent $p(f(\boldsymbol{x})|\mathcal{D}_n)$ as a Gaussian, $p(f(\boldsymbol{x})|\mathcal{D}_n) = \mathcal{N}(f|\mu_n(\boldsymbol{x}), \sigma_n^2(\boldsymbol{x}))$, where $\mu_n(\boldsymbol{x})$ and $\sigma_n(\boldsymbol{x})$ are functions that can be derived from the training data $\mathcal{D}_n = \{(\boldsymbol{x}_i, y_i) : i = 1 : n\}$ using a simple closed-form equation. The GP requires specifying a kernel function $\mathcal{K}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}')$, which measures similarities between input points $\boldsymbol{x}, \boldsymbol{x}'$. The intuition is that if two inputs are similar, so $\mathcal{K}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}')$ is large, then the corresponding function values are also likely to be similar, so $f(\boldsymbol{x})$ and $f(\boldsymbol{x}')$ should be positively correlated. This allows us to interpolate the function between the labeled training points; in some cases, it also lets us extrapolate beyond them.

GPs work well when we have little training data, and they support closed form Bayesian updating. However, exact updating takes $O(N^3)$ for N samples, which becomes too slow if we perform many function evaluations. There are various methods (Section 18.5.3) for reducing this to $O(NM^2)$ time, where M is a parameter we choose, but this sacrifices some of the accuracy.

In addition, the performance of GPs depends heavily on having a good kernel. We can estimate the kernel parameters $\boldsymbol{\theta}$ by maximizing the marginal likelihood, as discussed in Section 18.6.1. However,

since the sample size is small (by assumption), we can often get better performance by marginalizing out $\boldsymbol{\theta}$ using approximate Bayesian inference methods, as discussed in Section 18.6.2. See e.g., [WF16] for further details.

6.8.2.2 Bayesian neural networks

A natural alternative to GPs is to use a parametric model. If we use linear regression, we can efficiently perform exact Bayesian inference, as shown in Section 15.2. If we use a nonlinear model, such as a DNN, we need to use approximate inference methods. We discuss Bayesian neural networks in detail in Chapter 17. For their application to BayesOpt, see e.g. [Spr+16].

6.8.2.3 Other models

We are free to use other forms of regression model. [HHLB11] use an ensemble of random forests; such models can easily handle conditional parameter spaces, as we discuss in Section 6.8.4.2, although bootstrapping (which is needed to get uncertainty estimates) can be slow.

6.8.3 Acquisition functions

In BayesOpt, we use an **acquisition function** (also called a **merit function**) to evaluate the expected utility of each possible point we could query: $\alpha(\boldsymbol{x}|\mathcal{D}_n) = \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D}_n)}\left[U(\boldsymbol{x}, y; \mathcal{D}_n)\right]$, where $y = f(\boldsymbol{x})$ is the unknown value of the function at point $\boldsymbol{x}$, and $U()$ is a utility function. Different utility functions give rise to different acquisition functions, as we discuss below. We usually choose functions so that the utility of picking a point that has already been queried is small (or 0, in the case of noise-free observations), in order to encourage exploration.

6.8.3.1 Probability of improvement

Let us define $V_n = \max_{i=1}^{n} y_i$ to be the best value observed so far (known as the **incumbent**). (If the observations are noisy, using the highest mean value $\max_i \mathbb{E}\left[f(\boldsymbol{x}_i)\right]$ is a reasonable alternative [WF16].) Then we define the utility of some new point $\boldsymbol{x}$ using $U(\boldsymbol{x}, y; \mathcal{D}_n) = \mathbb{I}\left(y > V_n\right)$. This gives reward iff the new value is better than the incumbent. The corresponding acquisition function is then given by the expeced utility, $\alpha_{PI}(\boldsymbol{x}; \mathcal{D}_n) = p(f(\boldsymbol{x}) > V_n|\mathcal{D}_n)$. This is known as the **probability of improvement** [Kus64]. If $p(f|\mathcal{D}_n)$ is a GP, then this quantity can be computed in closed form, as follows:

$$\alpha_{PI}(\boldsymbol{x}; \mathcal{D}_n) = p(f(\boldsymbol{x}) > V_n|\mathcal{D}_n) = \Phi(\gamma(\boldsymbol{x}, V_n)) \tag{6.222}$$

where Φ is the cdf of the $\mathcal{N}(0, 1)$ distribution and

$$\gamma(\boldsymbol{x}, \tau) = \frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})} \tag{6.223}$$

6.8.3.2 Expected improvement

The problem with PI is that all improvements are considered equally good, so the method tends to exploit quite aggressively [Jon01]. A common alternative takes into account the amount of

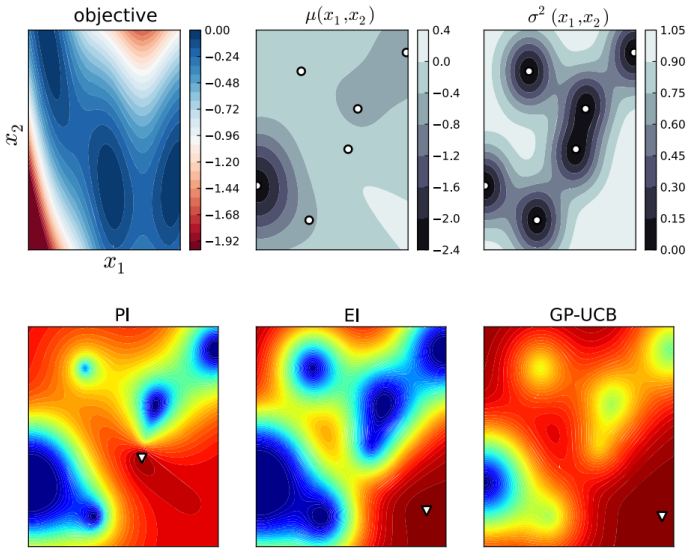


Figure 6.11: The first row shows the objective function, (the Branin function defined on $\mathbb{R}^2$), and its posterior mean and variance using a GP estimate. White dots are the observed data points. The second row shows 3 different acquisition functions (probability of improvement, expected improvement, and upper confidence bound); the white triangles are the maxima of the corresponding acquisition functions. From Figure 6 of [BCF10]. Used with kind permission of Nando de Freitas.

improvement by defining $U(\boldsymbol{x}, y; \mathcal{D}_n) = (y - V_n)\mathbb{I}(y > V_n)$ and

$$\alpha_{EI}(\boldsymbol{x}; \mathcal{D}_n) = \mathbb{E}_{\mathcal{D}_n}[U(\boldsymbol{x}, y)] = \mathbb{E}_{\mathcal{D}_n}[(f(\boldsymbol{x}) - V_n)\mathbb{I}(f(\boldsymbol{x}) > V_n)] \tag{6.224}$$

This acquisition function is known as the **expected improvement** (**EI**) criterion [Moc+96]. In the case of a GP surrogate, this has the following closed form expression:

$$\alpha_{EI}(\boldsymbol{x}; \mathcal{D}_n) = (\mu_n(\boldsymbol{x}) - \tau)\Phi(\gamma(\boldsymbol{x})) + \sigma_n(\boldsymbol{x})\phi(\gamma(\boldsymbol{x})) \tag{6.225}$$

where $\phi()$ is the pdf of the $\mathcal{N}(0,1)$ distribution. The first term encourages exploitation (evaluating points with high mean) and the second term encourages exploration (evaluating points with high variance). This is illustrated in Figure 6.10.

6.8.3.3 Upper confidence bound (UCB)

An alternative approach is to compute an **upper confidence bound** or **UCB** on the function, at some confidence level β_n, and then to define the acquisition function as follows: $\alpha_{UCB}(\boldsymbol{x}; \mathcal{D}_n) = \mu_n(\boldsymbol{x}) + \beta_n \sigma_n(\boldsymbol{x})$. This is the same as in the contextual bandit setting, discussed in Section 34.4.5, except we are optimizing over $\boldsymbol{x} \in \mathcal{X}$, rather than a finite set of arms $a \in \{1, \ldots, A\}$. If we use a GP for our surrogate, the method is known as **GP-UCB** [Sri+10].

6.8.3.4 Thompson sampling

We introduced **Thompson sampling** in Section 34.4.6 in the context of multi-armed bandits, where the state space is finite, $\mathcal{X} = \{1, \ldots, A\}$, and the acquisition function $\alpha(a; \mathcal{D}_n)$ corresponds to the probability that arm a is the best arm. We can generalize this to real-valued input spaces $\mathcal{X}$ using

$$\alpha(\boldsymbol{x}; \mathcal{D}_n) = \mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D}_n)}\left[\mathbb{I}\left(\boldsymbol{x} = \operatorname*{argmax}_{\boldsymbol{x}'} f_{\boldsymbol{\theta}}(\boldsymbol{x}')\right)\right] \tag{6.226}$$

We can compute a single sample approximation to this integral by sampling $\tilde{\boldsymbol{\theta}} \sim p(\boldsymbol{\theta}|\mathcal{D}_n)$. We can then pick the optimal action as follows:

$$\boldsymbol{x}_{n+1} = \operatorname*{argmax}_{\boldsymbol{x}} \alpha(\boldsymbol{x}; \mathcal{D}_n) = \operatorname*{argmax}_{\boldsymbol{x}} \mathbb{I}\left(\boldsymbol{x} = \operatorname*{argmax}_{\boldsymbol{x}'} f_{\tilde{\boldsymbol{\theta}}}(\boldsymbol{x}')\right) = \operatorname*{argmax}_{\boldsymbol{x}} f_{\tilde{\boldsymbol{\theta}}}(\boldsymbol{x}) \tag{6.227}$$

In other words, we greedily maximize the sampled surrogate.

For continuous spaces, Thompson sampling is harder to apply than in the bandit case, since we can't directly compute the best "arm" $\boldsymbol{x}_{n+1}$ from the sampled function. Furthermore, when using GPs, there are some subtle technical difficulties with sampling a function, as opposed to sampling the parameters of a parametric surroagate model (see [HLHG14] for discussion).

6.8.3.5 Entropy search

Since our goal in BayesOpt is to find $\boldsymbol{x}^* = \operatorname{argmax}_{\boldsymbol{x}} f(\boldsymbol{x})$, it makes sense to try to directly minimize our uncertainty about the location of $\boldsymbol{x}^*$, which we denote by $p_*(\boldsymbol{x}|\mathcal{D}_n)$. We will therefore define the utility as follows:

$$U(\boldsymbol{x}, y; \mathcal{D}_n) = \mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n\right) - \mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n \cup \{(\boldsymbol{x}, y)\}\right) \tag{6.228}$$

where $\mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n\right) = \mathbb{H}\left(p_*(\boldsymbol{x}|\mathcal{D}_n)\right)$ is the entropy of the posterior distribution over the location of the optimum. This is known as the information gain criterion; the difference from the objective used in active learning is that here we want to gain information about $\boldsymbol{x}^*$ rather than about f for all $\boldsymbol{x}$. The corresponding acquisition function is given by

$$\alpha_{ES}(\boldsymbol{x}; \mathcal{D}_n) = \mathbb{E}_{p(y|\boldsymbol{x}, \mathcal{D}_n)}\left[U(\boldsymbol{x}, y; \mathcal{D}_n)\right] = \mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n\right) - \mathbb{E}_{p(y|\boldsymbol{x}, \mathcal{D}_n)}\left[\mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n \cup \{(\boldsymbol{x}, y)\}\right)\right] \tag{6.229}$$

This is known as **entropy search** [HS12].

Unfortunately, computing $\mathbb{H}\left(\boldsymbol{x}^*|\mathcal{D}_n\right)$ is hard, since it requires a probability model over the input space. Fortunately, we can leverage the symmetry of mutual information to rewrite the acquisition function in Equation (6.229) as follows:

$$\alpha_{PES}(\boldsymbol{x}; \mathcal{D}_n) = \mathbb{H}\left(y|\mathcal{D}_n, \boldsymbol{x}\right) - \mathbb{E}_{\boldsymbol{x}^*|\mathcal{D}_n}\left[\mathbb{H}\left(y|\mathcal{D}_n, \boldsymbol{x}, \boldsymbol{x}^*\right)\right] \tag{6.230}$$

where we can approximate the expectation from $p(\boldsymbol{x}^*|\mathcal{D}_n)$ using Thompson sampling. Now we just have to model uncertainty about the output space y. This is known as **predictive entropy search** [HLHG14].

6.8.3.6 Knowledge gradient

So far the acquisition functions we have considered are all greedy, in that they only look one step ahead. The **knowledge gradient** acquisition function, proposed in [FPD09], looks two steps ahead by considering the improvement we might expect to get if we query $\boldsymbol{x}$, update our posterior, and then exploit our knowledge by maximizing wrt our new beliefs. More precisely, let us define the best value we can find if we query one more point:

$$V_{n+1}(\boldsymbol{x}, y) = \max_{\boldsymbol{x}'} \mathbb{E}_{p(f|\boldsymbol{x},y,\mathcal{D}_n)}\left[f(\boldsymbol{x}')\right] \tag{6.231}$$

$$V_{n+1}(\boldsymbol{x}) = \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D}_n)}\left[V_{n+1}(\boldsymbol{x}, y)\right] \tag{6.232}$$

We define the KG acquisition function as follows:

$$\alpha_{KG}(\boldsymbol{x}; \mathcal{D}_n) = \mathbb{E}_{\mathcal{D}_n}\left[(V_{n+1}(\boldsymbol{x}) - V_n)\mathbb{I}\left(V_{n+1}(\boldsymbol{x}) > V_n\right)\right] \tag{6.233}$$

Compare this to the EI function in Equation (6.224).) Thus we pick the point $\boldsymbol{x}_{n+1}$ such that observing $f(\boldsymbol{x}_{n+1})$ will give us knowledge which we can then exploit, rather than directly trying to find a better point with better f value.

6.8.3.7 Optimizing the acquisition function

The acquisition function $\alpha(\boldsymbol{x})$ is often multimodal (see e.g., Figure 6.11), since it will be 0 at all the previously queried points (assuming noise-free observations). Consequently maximizing this function can be a hard subproblem in itself [WHD18; Rub+20].

In the continuous setting, it is common to use multi-restart BFGS or grid search. We can also use the cross-entropy method (Section 6.9.5), using mixtures of Gaussians [BK10] or VAEs [Fau+18] as the generative model over $\boldsymbol{x}$. In the discrete, combinatorial setting (e.g., when optimizing biological sequences), [Bel+19] use regularized evolution, (Section 6.9.3), and [Ang+20] use proximial policy optimization (Section 35.3.4). Many other combinations are possible.

6.8.4 Other issues

There are many other issues that need to be tackled when using Bayesian optimization, a few of which we briefly mention below.

6.8.4.1 Parallel (batch) queries

In some cases, we want to query the objective function at multiple points in parallel; this is known as **batched Bayesian optimization**. Now we need to optimize over a set of possible queries, which is computationally even more difficult than the regular case. See [WHD18; DBB20] for some recent papers on this topic.

6.8.4.2 Conditional parameters

BayesOpt is often applied to hyper-parameter optimization. In many applications, some hyperparameters are only well-defined if other ones take on specific values. For example, suppose we are trying to automatically tune a classifier, as in the **Auto-Sklearn** system [Feu+15], or the **Auto-Weka**

system [Kot+17]. If the method chooses to use a neural network, it also needs to specify the number of layers, and number of hidden units per layer; but if it chooses to use a decision tree, it instead should specify different hyperparameters, such as the maximum tree depth.

We can formalize such problems by defining the search space in terms of a tree or DAG (directed acyclic graph), where different subsets of the parameters are defined at each leaf. Applying GPs to this setting requires non-standard kernels, such as those discussed in [Swe+13; Jen+17]. Alternatively, we can use other forms of Bayesian regression, such as ensembles of random forests [HHLB11], which can easily handle conditional parameter spaces.

6.8.4.3 Multi-fidelity surrogates

In some cases, we can construct surrogate functions with different levels of accuracy, each of which may take variable amounts of time to compute. In particular, let $f(\boldsymbol{x}, s)$ be an approximation to the true function at $\boldsymbol{x}$ with fidelity s. The goal is to solve $\max_{\boldsymbol{x}} f(\boldsymbol{x}, 0)$ by observing $f(\boldsymbol{x}, s)$ at a sequence of $(\boldsymbol{x}_i, s_i)$ values, such that the total cost $\sum_{i=1}^{n} c(s_i)$ is below some budget. For example, in the context of hyperparameter selection, s may control how long we run the parameter optimizer for, or how large the validation set is.

In addition to choosing what fidelity to use for an experiment, we may choose to terminate expensive trials (queries) early, if the results of their cheaper proxies suggest they will not be worth running to completion (see e.g., [Str19; Li+17c; FKH17]). Alternatively, we may choose to resume an earlier aborted run, to collect more data on it, as in the **freeze-thaw algorithm** [SSA14].

6.8.4.4 Constraints

If we want to maximize a function subject to known constraints, we can simply build the constraints into the acquisition function. But if the constraints are unknown, we need to estimate the support of the feasible set in addition to estimating the function. In [GSA14], they propose the weighted EI criterion, given by $\alpha_{wEI}(\boldsymbol{x}; \mathcal{D}_n) = \alpha_{EI}(\boldsymbol{x}; \mathcal{D}_n) h(\boldsymbol{x}; \mathcal{D}_n)$, where $h(\boldsymbol{x}; \mathcal{D}_n)$ is a GP with a Bernoulli observation model that specifies if $\boldsymbol{x}$ is feasible or not. Of course, other methods are possible. For example, [HL+16b] propose a method based on predictive entropy search.

6.9 Derivative free optimization

Derivative free optimization or **DFO** refers to a class of techniques for optimizing functions without using derivatives. This is useful for blackbox function optimization as well as discrete optimization. If the function is expensive to evaliate, we can use Bayesian optimization (Section 6.8). If the function is cheap to evaluate, we can use stochastic local search methods or evolutionary search methods, as we discuss below.

6.9.1 Local search

In this section, we discuss heuristic optimization algorithms that try to find the global maximum in a discrete, unstructured search space. These algorithms replace the local gradient based update, which has the form $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \boldsymbol{d}_t$, with the following discrete analog:

$$\boldsymbol{x}_{t+1} = \operatorname*{argmax}_{\boldsymbol{x} \in \text{nbr}(\boldsymbol{x}_t)} \mathcal{L}(\boldsymbol{x}) \tag{6.234}$$

where $\text{nbr}(\boldsymbol{x}_t) \subseteq \mathcal{X}$ is the set of **neighbors** of $\boldsymbol{x}_t$. This is called **hill climbing**, **steepest ascent**, or **greedy search**.

If the "neighborhood" of a point contains the entire space, Equation (6.234) will return the global optimum in one step, but usually such a global neighborhood is too large to search exhaustively. Consequently we usually define local neighborhoods. For example, consider the **8-queens problem**. Here the goal is to place queens on an 8×8 chessboard so that they don't attack each other (see Figure 6.14). The state space has the form $\mathcal{X} = 64^8$, since we have to specify the location of each queen on the grid. However, due to the constraints, there are only $8^8 \approx 17M$ feasible states. We define the neighbors of a state to be all possible states generated by moving a single queen to another square in the same column, so each node has $8 \times 7 = 56$ neighbors. According to [RN10, p.123], if we start at a randomly generated 8-queens state, steepest ascent gets stuck at a local maximum 86% of the time, so it only solves 14% of problem instances. However, it is fast, taking an average of 4 steps when it succeeds and 3 when it gets stuck.

In the sections below, we discuss slightly smarter algorithms that are less likely to get stuck in local maxima.

6.9.1.1 Stochastic local search

Hill climbing is greedy, since it picks the best point in its local neighborhood, by solving Equation (6.234) exactly. One way to reduce the chance of getting stuck in local maxima is to approximately maximize this objective at each step. For example, we can define a probability distribution over the uphill neighbors, proportional to how much they improve, and then sample one at random. This is called **stochastic hill climbing**. If we gradually decrease the entropy of this probability distribution (so we become greedier over time), we get a method called simulated annealing, which we discuss in Section 12.9.1.

Another simple technique is to use greedy hill climbing, but then whenever we reach a local maximum, we start again from a different random starting point. This is called **random restart hill climbing**. To see the benefit of this, consider again the 8-queens problem. If each hill-climbing search has a probability of $p \approx 0.14$ of success, then we expect to need $R = 1/p \approx 7$ restarts until we find a valid solution. The expected number of total steps can be computed as follows. Let $N_1 = 4$ be the average number of steps for successful trials, and $N_0 = 3$ be the average number of steps for failures. Then the total number of steps on average is $N_1 + (R-1)N_0 = 4 + 6 \times 3 = 22$. Since each step is quick, the overall method is very fast. For example, it can solve an n-queens problem with n =1M in under a minute.

Of course, solving the n-queens problem is not the most useful task in practice. However, it is typical of several real-world **boolean satisfiability problems**, which arise in problems ranging from AI planning to model checking (see e.g., [SLM92]). In such problems, simple **stochastic local search** (**SLS**) algorithms of the kind we have discussed work surprisingly well (see e.g., [HS05]).

6.9.1.2 Tabu search

Hill climbing will stop as soon as it reaches a local maximum or a plateau. Obviously one can perform a random restart, but this would ignore all the information that had been gained up to this point. A more intelligent alternative is called **tabu search** [GL97]. This is like hill climbing, except it allows moves that decrease (or at least do not increase) the scoring function, provided the move is to a new

Algorithm 6: Tabu search.

```
t := 0 // counts iterations
c := 0 // counts number of steps with no progress
Initialize x_0
x* := x_0 // current best incumbent
while c < c_max do
    x_{t+1} = argmax_{x ∈ nbr(x_t) \ {x_{t-τ}, ..., x_{t-1}}} f(x)
    if f(x_t) > f(x*) then
        x* := x_t
        c := 0
    else
        c := c + 1
    t := t + 1
return x*
```

state that has not been seen before. We can enforce this by keeping a tabu list which tracks the τ most recently visited states. This forces the algorithm to explore new states, and increases the chances of escaping from local maxima. We continue to do this for up to $c_{\max}$ steps (known as the "tabu tenure"). The pseudocode can be found in Algorithm 6. (If we set $c_{\max} = 1$, we get greedy hill climbing.)

For example, consider what happens when tabu search reaches a hill top, $\boldsymbol{x}_t$. At the next step, it will move to one of the neighbors of the peak, $\boldsymbol{x}_{t+1} \in \text{nbr}(\boldsymbol{x}_t)$, which will have a lower score. At the next step, it will move to the neighbor of the previous step, $\boldsymbol{x}_{t+2} \in \text{nbr}(\boldsymbol{x}_{t+1})$; the tabu list prevents it cycling back to $\boldsymbol{x}_t$ (the peak), so it will be forced to pick a neighboring point at the same height or lower. It continues in this way, "circling" the peak, possibly being forced downhill to a lower level-set (an inverse **basin flooding** operation), until it finds a ridge that leads to a new peak, or until it exceeds a maximum number of non-improving moves.

According to [RN10, p.123], tabu search increases the percentage of 8-queens problems that can be solved from 14% to 94%, although this variant takes an average of 21 steps for each successful instance and 64 steps for each failed instance.

6.9.1.3 Random search

A surprisingly effective strategy in problems where we know nothing about the objective is to use **random search**. In this approach, each iterate $\boldsymbol{x}_{t+1}$ is chosen uniformly at random from $\mathcal{X}$. This should always be tried as a baseline.

In [BB12], they applied this technique to the problem of hyper-parameter optimization for some ML models, where the objective is performance on a validation set. In their examples, the search space is continuous, $\Theta = [0, 1]^D$. It is easy to sample from this at random. The standard alternative approach is to quantize the space into a fixed set of values, and then to evaluate them all; this is known as **grid search**. (Of course, this is only feasible if the number of dimensions D is small.) They found that random search outperformed grid search. The intuitive reason for this is that many

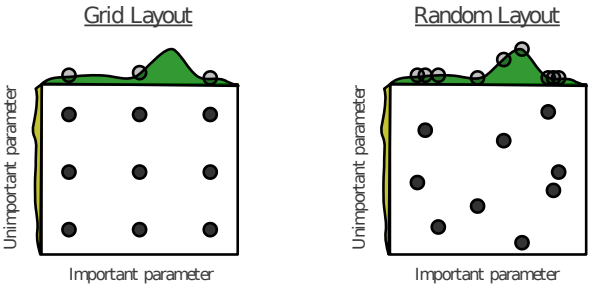


Figure 6.12: Illustration of grid search (left) vs random search (right). From Figure 1 of [BB12]. Used with kind permission of James Bergstra.

hyper-parameters do not make much difference to the objective function, as illustrated in Figure 6.12. Consequently it is a waste of time to place a fine grid along such unimportant dimensions.

RS has also been used to optimize the parameters of MDP policies, where the objective has the form $f(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{x}}}[R(\boldsymbol{\tau})]$ is the expected reward of trajectories generated by using a policy with parameters $\boldsymbol{x}$. For policies with few free parameters, RS can outperform more sophisticated reinforcement learning methods described in Chapter 35, as shown in [MGR18]. In cases where the policy has a large number of parameters, it is sometimes possible to project them to a lower dimensional random subspace, and perform optimization (either grid search or random search) in this subspace [Li+18a].

6.9.2 Simulated annealing

Simulated annealing [KJV83; LA87] is a **stochastic local search** algorithm (Section 6.9.1.1) that attempts to find the global minimum of a black-box function $\mathcal{E}(\boldsymbol{x})$, where $\mathcal{E}()$ is known as the **energy function**. The method works by converting the energy to an (unnormalized) probability distribution over states by defining $p(\boldsymbol{x}) = \exp(-\mathcal{E}(\boldsymbol{x}))$, and then using a variant of the **Metropolis Hastings** algorithm to sample from a set of probability distributions, designed so that at the final step, the method samples from one of the modes of the distribution, i.e., it finds one of the most likely states, or lowest energy states. This approach can be used for both discrete and continuous optimization. See Section 12.9.1 for details.

6.9.3 Evolutionary algorithms

Stochastic local search (SLS) maintains a single "best guess" at each step, $\boldsymbol{x}_t$. If we run this for T steps, and restart K times, the total cost is TK. A natural alternative is to maintain a set or **population** of K good candidates, $\mathcal{S}_t$, which we try to improve at each step. This is called an an **evolutionary algorithm** (**EA**). If we run this for T steps, it also takes TK time; however, it can often get better results than multi-restart SLS, since the search procedure explores more of the space in parallel, and information from different members of the population can be shared. Many versions of EA are possible, as we discuss below.

Since EA algorithms draw inspiration from the biological process of evolution, they also borrow

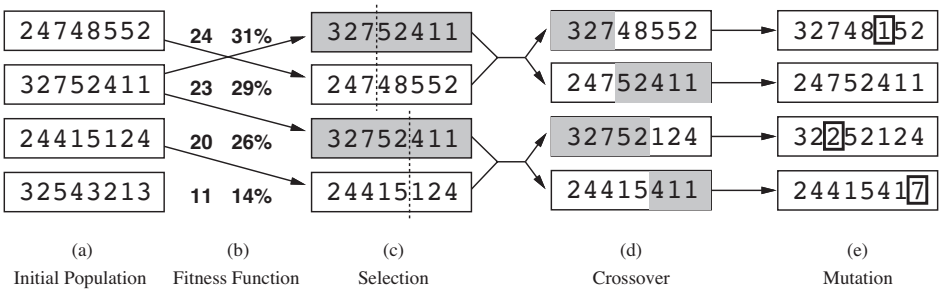


Figure 6.13: Illustration of a genetic algorithm applied to the 8-queens problem. (a) Initial population of 4 strings. (b) We rank the members of the population by fitness, and then compute their probability of mating. Here the integer numbers represent the number of nonattacking pairs of queens, so the global maximum has a value of 28. We pick an individual $\boldsymbol{\theta}$ with probability $p(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta})/Z$, where $Z = \sum_{\boldsymbol{\theta} \in \mathcal{P}} \mathcal{L}(\boldsymbol{\theta})$ sums the total fitness of the population. For example, we pick the first individual with probability $24/78 = 0.31$, the second with probability $23/78 = 0.29$, etc. In this example, we pick the first individual once, the second twice, the third one once, and the last one does not get to breed. (c) A split point on the "chromosome" of each parent is chosen at random. (d) The two parents swap their chromosome halves. (e) We can optionally apply pointwise mutation. From Figure 4.6 of [RN10]. Used with kind permission of Peter Norvig.

a lot of its terminology. The **fitness** of a member of the population is the value of the objective function (possibly normalized across population members). The members of the population at step $t+1$ are called the **offspring**. These can be created by randomly choosing a **parent** from $\mathcal{S}_t$ and applying a random **mutation** to it. This is like asexual reproduction. Alternatively we can create an offspring by choosing two parents from $\mathcal{S}_t$, and then combining them in some way to make a child, as in sexual reproduction; combining the parents is called **recombination**. (It is often followed by mutation.)

The procedure by which parents are chosen is called the **selection function**. In **truncation selection**, each parent is chosen from the fittest K members of the population (known as the **elite set**). In **tournament selection**, each parent is the fittest out of K randomly chosen members. In **fitness proportionate selection**, also called **roulette wheel selection**, each parent is chosen with probability proportional to its fitness relative to the others. We can also "kill off" the oldest members of the population, and then select parents based on their fitness; this is called **regularized evolution** [Rea+19]).

In addition to the selection rule for patents, we need to specify the recombination and mutation rules. There are many possible choices for these heuristics. We briefly mention a few of them below.

- In a **genetic algorithm** (**GA**) [Gol89; Hol92], we use mutation and a particular recombination method based on **crossover**. To implement crossover, we assume each individual is represented as a vector of integers or binary numbers, by analogy to **chromosomes**. We pick a split point along the chromosome for each of the two chosen parents, and then swap the strings, as illustrated in Figure 6.13.

- In **genetic programming** [Koz92], we use use a tree-structured representation of individuals, instead of a bit string. This representation ensures that all crossovers result in valid children,

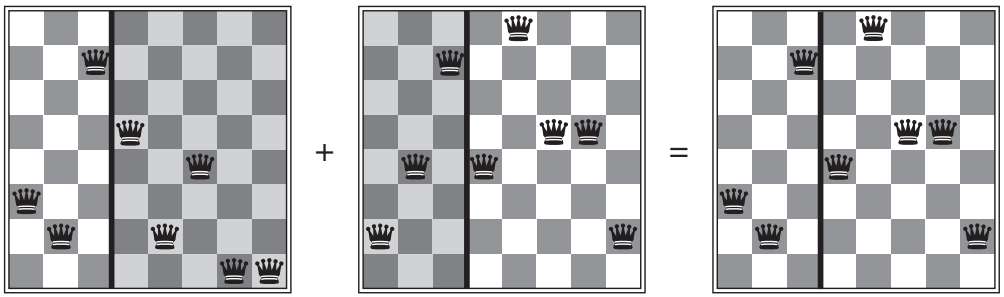


Figure 6.14: The 8-queens states corresponding to the first two parents in Figure 6.13(c) and their first child in Figure 6.13(d). We see that the encoding 32752411 *means that the first queen is in row 3 (counting from the bottom left), the second queen is in row 2, etc. The shaded columns are lost in the crossover, but the unshaded columns are kept. From Figure 4.7 of [RN10]. Used with kind permission of Peter Norvig.*

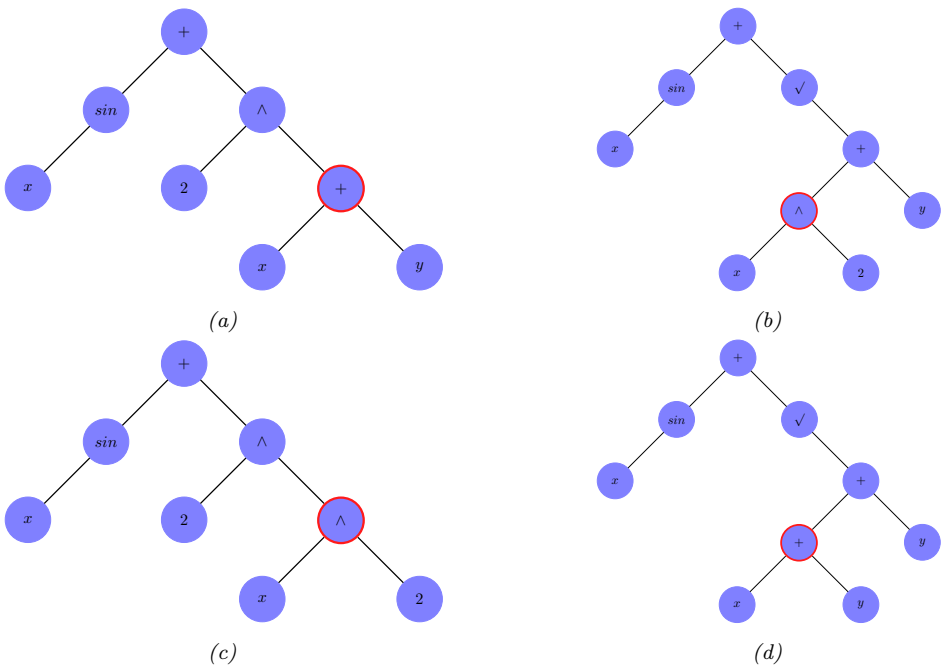


Figure 6.15: Illustration of crossover operator in a genetic program. (a-b) the two parents, representing $\sin(x) + (x+y)^2$ *and* $\sin(x) + \sqrt{x^2+y}$*. The red circles denote the two crossover points. (c-d) the two children, representing* $\sin(x) + (x^2)^2$ *and* $\sin(x) + \sqrt{x+y+y}$*. Adapted from Figure 9.2 of [Mit97]*

as illustrated in Figure 6.15. Genetic programming can be useful for finding good programs as well as other structured objects, such as neural networks. In **evolutionary programming**, the structure of the tree is fixed and only the numerical parameters are evolved.

- In **surrogate assisted EA**, a surrogate function $\hat{f}(s)$ is used instead of the true objective function $f(s)$ in order to speed up the evaluation of members of the population (see [Jin11] for a survey). This is similar to the use of response surface models in Bayesian optimization (Section 6.8), except it does not deal with the explore-exploit tradeoff.
- In a **memetic algorithm** [MC03], we combine mutation and recombination with standard local search.

Evolutionary algorithms have been applied to a large number of applications, including training neural networks (this combination is known as **neuroevolution** [Sta+19]). An efficient JAX-based library for (neuro)-evolution can be found at https://github.com/google/evojax.

6.9.4 Estimation of distribution (EDA) algorithms

EA methods maintain a population of good candidate solutions, which can be thought of as an implicit (nonparametric) density model over states with high fitness. [BC95] proposed to "remove the genetics from GAs", by explicitly learning a probabilistic model over the configuration space that puts its mass on high scoring solutions. Thay is, the population becomes the set of parameters of a generative model, $\boldsymbol{\theta}_t$.

One way to learn such as model is as follows. We start by creating a sample of $K' > K$ candidate solutions from the current model, $\mathcal{S}_t = \{\boldsymbol{x}_k \sim p(\boldsymbol{x}|\boldsymbol{\theta}_t)\}$. We then rank the samples using the fitness function, and then pick the most promising subset $\mathcal{S}_t^*$ of size K using a selection operator (this is known as **truncation selection**). Finally, we fit a new probabilistic model $p(\boldsymbol{x}|\boldsymbol{\theta}_{t+1})$ to $\mathcal{S}_t^*$ using maximum likelihood estimation. This is called the **estimation of distribution** or **EDA** algorithm (see e.g., [LL02; PSCP06; Hau+11; PHL12; Hu+12; San17; Bal17]).

Note that EDA is equivalent to minimizing the cross-entropy between the empirical distribution defined by $\mathcal{S}_t^*$ and the model distribution $p(\boldsymbol{x}|\boldsymbol{\theta}_{t+1})$. Thus EDA is related to the **cross entropy method**, as described in Section 6.9.5, although CEM usually assumes the special case where $p(\boldsymbol{x}|\boldsymbol{w}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. EDA is also closely related to the EM algorithm, as discussed in [Bro+20a].

As a simple example, suppose the configuration space is bit strings of length D, and the fitness function is $f(\boldsymbol{x}) = \sum_{d=1}^{D} x_d$, where $x_d \in \{0, 1\}$ (this is called the **one-max** function in the EA literature). A simple probabilistic model for this is a fully factored model of the form $p(\boldsymbol{x}|\boldsymbol{\theta}) = \prod_{d=1}^{D} \text{Ber}(x_d|\theta_d)$. Using this model inside of DBO results in a method called univariate marginal distribution algorithm or **UMDA**.

We can estimate the parameters of the Bernoulli model by setting θ_d to the fraction of samples in $\mathcal{S}_t^*$ that have bit d turned on. Alternatively, we can incrementally adjust the parameters. The population-based incremental learning (**PBIL**) algorithm [BC95] applies this idea to the factored Bernoulli model, resulting in the following update:

$$\hat{\theta}_{d,t+1} = (1 - \eta_t)\hat{\theta}_{d,t} + \eta_t \overline{\theta}_{d,t} \tag{6.235}$$

where $\overline{\theta}_{d,t} = \frac{1}{N_t} \sum_{k=1}^{K} \mathbb{I}(x_{k,d} = 1)$ is the MLE estimated from the $K = |\mathcal{S}_t^*|$ samples generated in the current iteration, and η_t is a learning rate.

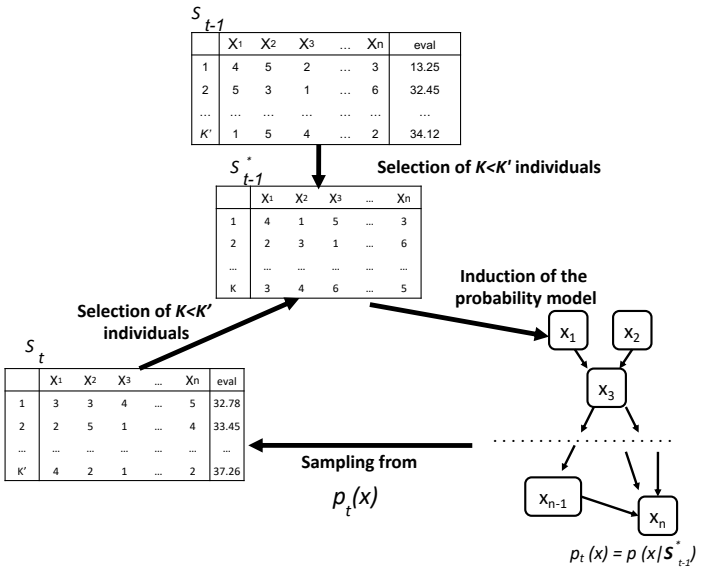


Figure 6.16: Illustration of the BOA algorithm (EDA applied to a generative model structured as a Bayes net). Adapted from Figure 3 of [PHL12].

It is straightforward to use more expressive probability models that capture dependencies between the parameters (these are known as **building blocks** in the EA literature). For example, in the case of real-valued parameters, we can use a multivariate Gaussian, $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The resulting method is called the **estimation of multivariate normal algorithm** or **EMNA**, [LL02]. (See also Section 6.9.5.)

For discrete random variables, it is natural to use probabilistic graphical models (Chapter 4) to capture dependencies between the variables. [BD97] learn a tree-structured graphical model using the Chow-Liu algorithm (Supplementary Section 30.1.1); [BJV97] is a special case of this where the graph is a tree. We can also learn more general graphical model structures (see e.g., [LL02]). We typically use a Bayes net (Section 4.2), since we can use ancestral sampling (Section 4.2.5) to easily generate samples; the resulting method is therefore called the **Bayesian Optimization Algorithm** (**BOA**) [PGCP00].[4] The hierarchical BOA (**hBOA**) algorithm [Pel05] extends this by using decision trees and decision graphs to represent the local CPTs in the Bayes net (as in [CHM97]), rather than using tables. In general, learning the structure of the probability model for use in EDA is called **linkage learning**, by analogy to how genes can be linked together if they can be co-inherited as a building block.

We can also use deep generative models to represent the distribution over good candidates. For example, [CSF16] use denoising autoencoders and NADE models (Section 22.2), [Bal17] uses a DNN regressor which is then inverted using gradient descent on the inputs, [PRG17] uses RBMs

4. This should not be confused with the Bayesian optimization methods we discuss in Section 6.8, that uses response surface modeling to model $p(f(\boldsymbol{x}))$ rather than $p(\boldsymbol{x}^*)$.

(Section 4.3.3.2), [GSM18] uses VAEs (Section 21.2), etc. Such models might take more data to fit (and therefore more function calls), but can potentially model the probability landscape more faithfully. (Whether that translates to better optimization performance is not clear, however.)

6.9.5 Cross-entropy method

The **cross-entropy method** [Rub97; RK04; Boe+05] is a special case of EDA (Section 6.9.4) in which the population is represented by a multivariate Gaussian. In particular, we set $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\Sigma}_{t+1}$ to the empirical mean and covariance of $\mathcal{S}^*_{t+1}$, which are the top K samples. This is closely related to the SMC algorithm for sampling rare events discussed in Section 13.6.4.

The CEM is sometimes used for model-based RL (Section 35.4), since it is simple and can find reasonably good optima of multi-modal objectives. It is also sometimes used inside of Bayesian optimization (Section 6.8), to optimize the multi-modal acquisition function (see [BK10]).

6.9.5.1 Differentiable CEM

The **differentiable CEM** method of [AY19] replaces the top K operator with a soft, differentiable approximation, which allows the optimizer to be used as part of an end-to-end differentiable pipeline. For example, we can use this to create a differentiable model predictive control (MPC) algorithm (Section 35.4.1), as described in Section 35.4.5.2.

The basic idea is as follows. Let $\mathcal{S}_t = \{\boldsymbol{x}_{t,i} \sim p(\boldsymbol{x}|\boldsymbol{\theta}_t) : i = 1 : K'\}$ represent the current population, with fitness values $v_{t,i} = f(\boldsymbol{x}_{t,i})$. Let $v^*_{t,K}$ be the K'th smallest value. In CEM, we compute the set of top K samples, $\mathcal{S}^*_t = \{i : v_{t,i} \geq v^*_{t,K}\}$, and then update the model based on these: $\boldsymbol{\theta}_{t+1} = \text{argmax}_{\boldsymbol{\theta}} \sum_{i \in \mathcal{S}_t} p_t(i) \log p(\boldsymbol{x}_{t,i}|\boldsymbol{\theta})$, where $p_t(i) = \mathbb{I}\left(i \in \mathcal{S}^*_t\right) / |\mathcal{S}^*_t|$. In the differentiable version, we replace the sparse distribution $\boldsymbol{p}_t$ with the "soft" dense distribution $\boldsymbol{q}_t = \Pi(\boldsymbol{p}_t; \tau, K)$, where

$$\Pi(\boldsymbol{p}; \tau, K) = \underset{\mathbf{0} \leq \boldsymbol{q} \leq \mathbf{1}}{\text{argmin}} -\boldsymbol{p}^\mathsf{T}\boldsymbol{q} - \tau\, \mathbb{H}(\boldsymbol{q}) \;\; \text{s.t.} \;\; \mathbf{1}^\mathsf{T}\boldsymbol{q} = K \tag{6.236}$$

projects the distribution $\boldsymbol{p}$ onto the polytope of distributions which sum to K. (Here $\mathbb{H}(\boldsymbol{q}) = -\sum_i q_i \log(q_i) + (1 - q_i) \log(1 - q_i)$ is the entropy, and $\tau > 0$ is a temperature parameter.) This projection operator (and hence the whole DCEM algorithm) can be backpropagated through using implicit differentiation [AKZK19].

6.9.6 Evolutionary strategies

Evolution strategies [Wie+14] are a form of distribution-based optimization in which the distribution over the population is represented by a Gaussian, $p(\boldsymbol{x}|\boldsymbol{\theta}_t)$ (see e.g., [Sal+17b]). Unlike CEM, the parameters are updated using gradient ascent applied to the expected value of the objective, rather than using MLE on a set of elite samples. More precisely, consider the smoothed objective $\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p(\boldsymbol{x}|\boldsymbol{\theta})}\left[f(\boldsymbol{x})\right]$. We can use the REINFORCE estimator (Section 6.5.3) to compute the gradient of this objective as follows:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p(\boldsymbol{x}|\boldsymbol{\theta})}\left[f(\boldsymbol{x}) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}|\boldsymbol{\theta})\right] \tag{6.237}$$

This can be approximated by drawing Monte Carlo samples. We discuss how to compute this gradient below.

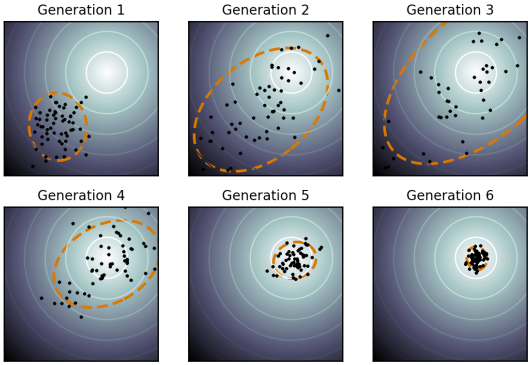


Figure 6.17: Illustration of the CMA-ES method applied to a simple 2d function. The dots represent members of the population, and the dashed orange ellipse represents the multivariate Gaussian. From https://en.wikipedia.org/wiki/CMA-ES. Used with kind permission of Wikipedia author Sentewolf.

6.9.6.1 Natural evolutionary strategies

If the probability model is in the exponential family, we can compute the natural gradient (Section 6.4), rather than the "vanilla" gradient, which can result in faster convergence. Such methods are called **natural evolution strategies** [Wie+14].

6.9.6.2 CMA-ES

The **CMA-ES** method of [Han16], which stands for "covariance matrix adaptation evolution strategy" is a kind of NES. It is very similar to CEM except it updates the parameters in a special way. In particular, instead of computing the new mean and covariance using unweighted MLE on the elite set, we attach weights to the elite samples based on their rank. We then set the new mean to the weighted MLE of the elite set.

The update equations for the covariance are more complex. In particular, "evolutionary paths" are also used to accumulate the search directions across successive generations, and these are used to update the covariance. It can be show that the resulting updates approximate the natural gradient of $\mathcal{L}(\boldsymbol{\theta})$ without explicitly modeling the Fisher information matrix [Oll+17].

Figure 6.17 illustrates the method in action.

6.10 Optimal Transport

This section is written by Marco Cuturi.

In this section, we focus on **optimal transport** theory, a set of tools that have been proposed, starting with work by [Mon81], to compare two probability distributions. We start from a simple example involving only matchings, and work from there towards various extensions.

6.10.1 Warm-up: Matching optimally two families of points

Consider two families $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$, each consisting in $n > 1$ distinct points taken from a set $\mathcal{X}$. A *matching* between these two families is a bijective mapping that assigns to each point $\mathbf{x}_i$ another point $\mathbf{y}_j$. Such an assignment can be encoded by pairing indices $(i, j) \in \{1, \ldots, n\}^2$ such that they define a *permutation* σ in the symmetric group $\mathcal{S}_n$. With that convention and given a permuation σ, $\mathbf{x}_i$ would be assigned to $\mathbf{y}_{\sigma_i}$, the σ_i-th element in the second family.

Matchings costs. When matching a family with another, it is natural to consider the cost incurred when pairing any point $\mathbf{x}_i$ with another point $\mathbf{y}_j$, for all possible pairs $(i, j) \in \{1, \ldots, n\}^2$. For instance, $\mathbf{x}_i$ might contain information on the current location of a taxi driver i, and $\mathbf{y}_j$ that of a user j who has just requested a taxi; in that case, $C_{ij} \in \mathbb{R}$ may quantify the cost (in terms of time, fuel or distance) required for taxi driver i to reach user j. Alternatively, $\mathbf{x}_i$ could represent a vector of skills held by a job seeker i and $\mathbf{y}_j$ a vector quantifying desirable skills associated with a job posting j; in that case C_{ij} could quantify the numbers of hours required for worker i to carry out job j. We will assume without loss of generality that the values C_{ij} are obtained by evaluating a cost *function* $c : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ on the pair $(\mathbf{x}_i, \mathbf{y}_j)$, namely $C_{ij} = c(\mathbf{x}_i, \mathbf{y}_j)$. In many applications of optimal transport, such cost functions have a geometric interpretation and are typically distance functions on $\mathcal{X}$ as in Fig. 6.18, in which $\mathcal{X} = \mathbb{R}^2$, or as will be later discussed in §6.10.2.4.

Least-cost Matchings. Equipped with a cost function c, the *optimal* matching (or assignment) problem is that of finding a permutation that reaches the smallest total cost, as defined by the function

$$\min_{\sigma} E(\sigma) = \sum_{i=1}^{n} c(\mathbf{x}_i, \mathbf{y}_{\sigma_j}) \,. \tag{6.238}$$

The optimal matching problem is arguably one of the simplest combinatorial optimization problems, tackled as early as the 19th century [JB65]. Although a naive enumeration of all permutations would require evaluating objective E a total of $n!$ times, the Hungarian algorithm [Kuh55] was shown to provide the optimal solution in polynomial time [Mun57], and later refined to require in the worst case $O(n^3)$ operations.

6.10.2 From Optimal Matchings to Kantorovich and Monge formulations

The optimal matching problem is relevant to many applications, but it suffers from a few limitations. One could argue that most of the optimal transport literature arises from the necessity to overcome these limitations and extend (6.238) to more general settings. An obvious issue arises when the number of points available in both familites is not the same. The second limitation arises when considering a continuous setting, namely when trying to match (or morph) two probability densities, rather than families of atoms (discrete measures).

6.10.2.1 Mass splitting

Suppose again that all points $\mathbf{x}_i$ and $\mathbf{y}_j$ describe skills, respectively held by a worker i and needed for a task j to be fulfilled in a factory. Since finding a matching is equivalent to finding a permutation in $\{1, \ldots, n\}$, problem (6.238) cannot handle cases in which the number of workers is larger (or smaller) than the number of tasks. More problematically, the assumption that every single task is indivisible,

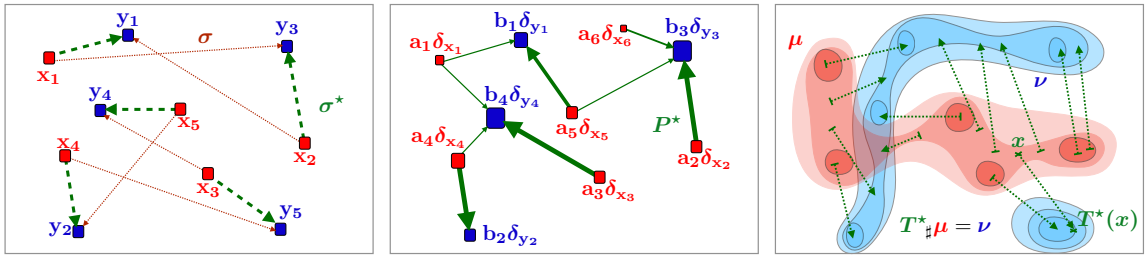


Figure 6.18: (left) Matching a family of 5 points to another is equivalent to considering a permutation in $\{1, \dots, n\}$. *When to each pair* $(\mathbf{x}_i, \mathbf{y}_j) \in \mathbb{R}^2$ *is associated a cost equal to the distance* $\|\mathbf{x}_i - \mathbf{y}_j\|$, *the optimal matching problem involves finding a permutation* σ *that minimizes* $\|\mathbf{x}_i - \mathbf{y}_{\sigma_i}\|$ *for* i *in* $\{1, 2, 3, 4, 5\}$. *(middle) The Kantorovich formulation of optimal transport generalizes optimal matchings, and arises when comparing discrete measures, that is families of weighted points that do not necessarily share the same size but do share the same total mass. The relevant variable is a matrix* P *of size* $n \times m$, *which must satisfy row-sum and column-sum constraints, and which minimizes its dot product with matrix* C_{ij}. *(right) another direct extension of the matching problem lies when, intuitively, the number* n *of points that is described is such that the considered measures become continuous densities. In that setting, and unlike the Kantorovich setting, the goal is to seek a map* $T : \mathcal{X} \to \mathcal{X}$ *which, to any point* x *in the support of the input measure* μ *is associated a point* $y = T(x)$ *in the support of* ν. *The push-forward constraint* $T_\sharp \mu = \nu$ *ensures that* ν *is recovered by applying map* T *to all points in the support of* μ; *the optimal map* $T^\star$ *is that which minimizes the distance between* x *and* $T(x)$, *averaged over* μ.

or that workers are only able to dedicate themselves to a single task, is hardly realistic. Indeed, certain tasks may require more (or less) dedication than that provided by a single worker, whereas some workers may only be able to work part-time, or, on the contrary, be willing to put extra hours. The rigid machinery of permutations falls short of handling such cases, since permutations are by definition one-to-one associations. The Kantorovich formulation allows for *mass-splitting*, the idea that the effort provided by a worker or needed to complete a given task can be split. In practice, to each of the n workers is associated, in addition to $\mathbf{x}_i$, a positive number $\mathbf{a}_i > 0$. That number represents the amount of time worker i is able to provide. Similarly, we introduce numbers $\mathbf{b}_j > 0$ describing the amount of time needed to carry out each of the m tasks (n and m do not necessarily coincide). Worker i is therefore described as a pair $(\mathbf{a}_i, \mathbf{x}_i)$, mathematically equivalent to a *weighted Dirac measure* $\mathbf{a}_i \delta_{\mathbf{x}_i}$. The overall workforce available to the factory is described as a discrete measure $\sum_i \mathbf{a}_i \delta_{\mathbf{x}_i}$, whereas its tasks are described in $\sum_j \mathbf{b}_j \delta_{\mathbf{y}_j}$. If one assumes further that the factory has a balanced workload, namely that $\sum_i \mathbf{a}_i = \sum_j \mathbf{b}_j$, then the Kantorovich [Kan42] formulation of optimal transport is:

$$\mathrm{OT}_C(\mathbf{a}, \mathbf{b}) \triangleq \min_{P \in \mathbf{R}_+^{n \times m}, P\mathbf{1}_m = \mathbf{a}, P^T \mathbf{1}_m = \mathbf{b}} \langle P, C \rangle \triangleq \sum_{i,j} P_{ij} C_{ij}. \tag{K}$$

The interpretation behind such matrices is simple: each coefficient P_{ij} describes an allocation of time for worker i to spend on task j. The i-th row-sum must be equal to the total $\mathbf{a}_i$ for the time constraint of worker i to be satisfied, whereas the j-th column-sum must be equal to $\mathbf{b}_j$, reflecting that the time needed to complete task j has been budgeted.

6.10.2.2 Monge formulation and optimal push-forward maps

By introducing mass-splitting, the Kantorovich formulation of optimal transport allows for a far more general comparison between discrete measures of different sizes and weights (middle plot of Fig. 6.18). Naturally, this flexibility comes with a downside: one can no longer associate to each point $\mathbf{x}_i$ another point $\mathbf{y}_j$ to which it is uniquely associated, as was the case with the classical matching problem. Interestingly, this property can be recovered in the limit where the measures become densities. Indeed, the Monge [Mon81] formulation of optimal transport allows to recover precisely that property, on the condition (loosely speaking) that measure μ admits a density. In that setting, the analogous mathematical object guaranteeing that μ is mapped onto ν is that of *push-forward* maps morphing μ to ν, namely maps T such that for any measurable set $A \subset \mathcal{X}$, $\mu(T^{-1}(A)) = \nu(A)$. When T is differentiable, and μ, ν have densities p and q w.r.t. the Lebesgue measure in $\mathbb{R}^d$, this statement is equivalent, thanks to the change of variables formula, to ensuring almost everywhere that:

$$q(T(x)) = p(x)|J_T(x)|\,, \tag{6.239}$$

where $|J_T(x)|$ stands for the determinant of the Jacobian matrix of T evaluated at x.

Writing $T_\sharp \mu = \nu$ when T does satisfy these conditions, the Monge [Mon81] problem consists in finding the best map T that minimizes the average cost between $\mathbf{x}$ and its displacement $T(\mathbf{x})$,

$$\inf_{T : T_\sharp \mu = \nu} \int_{\mathcal{X}} c(\mathbf{x}, T(\mathbf{x}))\, \mu(\mathrm{d}\mathbf{x}). \tag{M}$$

T is therefore a map that pushes forward μ to ν globally, but which results, on average, in the smallest average cost. While very intuitive, the Monge problem turns out to be extremely difficult to solve in practice, since it is non-convex. Indeed, one can easily check that the constraint $\{T_\sharp \mu = \nu\}$ is not convex, since one can easily find counter-examples for which $T_\sharp \mu = \nu$ and $T'_\sharp \nu$ yet $(\frac{1}{2}T + \frac{1}{2}T')_\sharp \mu \neq \nu$. Luckily, Kantorovich's approach also works for continuous measures, and yields a comparatively much simpler linear program.

6.10.2.3 Kantorovich formulation

The Kantovorich problem (K) can also be extended to a continuous setting: Instead of optimizing over a subset of matrices in $\mathbb{R}^{n \times m}$, consider $\Pi(\mu, \nu)$, the subset of joint probability distributions $\mathcal{P}(\mathcal{X} \times \mathcal{X})$ with marginals μ and ν, namely

$$\Pi(\mu, \nu) \triangleq \{\pi \in \mathcal{P}(\mathcal{X}^2) : \forall A \subset \mathcal{X}, \pi(A \times \mathcal{X}) = \mu(A) \text{ and } \pi(\mathcal{X} \times A) = \nu(A)\}. \tag{6.240}$$

Note that $\Pi(\mu, \nu)$ is not empty since it always contains the product measure $\mu \otimes \nu$. With this definition, the continuous formulation of (K) can be obtained as

$$\mathrm{OT}_c(\mu, \nu) \triangleq \inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X}^2} c \, \mathrm{d}\pi\,. \tag{K2}$$

Notice that (K2) subsumes directly (K), since one can check that that they coincide when μ, ν are discrete measures, with respective probability weights $\mathbf{a}, \mathbf{b}$ and locations $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and $(\mathbf{y}_1, \ldots, \mathbf{y}_m)$.

6.10.2.4 Wasserstein distances

When c is equal to a metric d exponentiated by an integer, the optimal value of the Kantorovich problem is called the Wasserstein *distance* between μ and ν:

$$W_p(\mu,\nu) \triangleq \left(\inf_{\pi \in \Pi(\mu,\nu)} \int_{\mathcal{X}^2} d(\mathbf{x},\mathbf{y})^p \, \mathrm{d}\pi(\mathbf{x},\mathbf{y}) \right)^{1/p} . \tag{6.241}$$

While the symmetry and the fact that $W_p(\mu,\nu) = 0 \Rightarrow \mu = \nu$ are relatively easy to prove provided d is a metric, proving the triangle inequality is slightly more challenging, and builds on a result known as the gluing lemma ([Vil08, p.23]). The p-th power of $W_p(\mu,\nu)$ is often abbreviated as $W_p^p(\mu,\nu)$.

6.10.3 Solving optimal transport

6.10.3.1 Duality and cost concavity

Both (K) and (K2) are linear programs: their constraints and objective functions only involve summations. In that sense they admit a dual formulation (here, again, (DK2) subsumes (DK)):

$$\max_{\substack{\mathbf{f}\in\mathbb{R}^n, \mathbf{g}\in\mathbb{R}^m \\ \mathbf{f}\oplus\mathbf{g}\leq C}} \mathbf{f}^T\mathbf{a} + \mathbf{g}^T\mathbf{b} \tag{DK}$$

$$\sup_{f\oplus g \leq c} \int_{\mathcal{X}} f \, \mathrm{d}\mu + \int_{\mathcal{X}} g \, \mathrm{d}\nu; \tag{DK2}$$

where the sign $\oplus$ denotes tensor addition for vectors, $\mathbf{f}\oplus\mathbf{g} = [\mathbf{f}_i + \mathbf{g}_j]_{ij}$, or functions, $f \oplus g : \mathbf{x},\mathbf{y} \mapsto f(\mathbf{x}) + g(\mathbf{y})$. In other words, the dual problem looks for a pair of vectors (or functions) that attain the highest possible expectation when summed against $\mathbf{a}$ and $\mathbf{b}$ (or integrated against μ,ν), pending the constraint that they do not differ too much across points $\mathbf{x},\mathbf{y}$, as measured by c.

The dual problems in (K) and (K2) have two variables. Focusing on the continuous formulation, a closer inspection shows that it is possible, given a function f for the first measure, to compute the best possible candidate for function g. That function g should be as large as possible, yet satisfy the constraint that $g(\mathbf{y}) \leq c(\mathbf{x},\mathbf{y}) - f(\mathbf{x})$ for all $\mathbf{x},\mathbf{y}$, making

$$\forall \mathbf{y} \in \mathcal{X}, \overline{f}(\mathbf{y}) \triangleq \inf_{\mathbf{x}} c(\mathbf{x},\mathbf{y}) - f(\mathbf{x}) \,, \tag{6.242}$$

the optimal choice. $\overline{f}$ is called the c-transform of f. Naturally, one may choose to start instead from g, to define an alternative c-transform:

$$\forall \mathbf{x} \in \mathcal{X}, \widetilde{g}(\mathbf{x}) \triangleq \inf_{\mathbf{y}} c(\mathbf{x},\mathbf{y}) - g(\mathbf{y}) \,. \tag{6.243}$$

Since these transformations can only improve solutions, one may even think of applying alternatively these transformations to an arbitrary f, to define $\overline{f}$, $\widetilde{\overline{f}}$ and so on. One can show, however, that this has little interest, since

$$\widetilde{\overline{\widetilde{\overline{f}}}} = \overline{f} \,. \tag{6.244}$$

This remark allows, nonetheless, to narrow down the set of candidate functions to those that have already undergone such transformations. This reasoning yields the so-called set of c-concave functions, $\mathcal{F}_c \triangleq \{f \,|\exists g : \mathcal{X} \to \mathbb{R}, f = \widetilde{g}\}$, which can be shown, equivalently, to be the set of functions f such that $f = \widetilde{\overline{f}}$. One can therefore focus our attention to c-concave functions to solve (DK2) using a so-called semi-dual formulation,

$$\sup_{f \in \mathcal{F}_c} \int_{\mathcal{X}} f \, \mathrm{d}\mu + \int_{\mathcal{X}} \overline{f} \, \mathrm{d}\nu \,. \tag{DK2}$$

Going from (DK2) to (DK2), we have removed a dual variable g and narrowed down the feasible set to $\mathcal{F}_c$, at the cost of introducing the highly non-linear transform $\overline{f}$. This reformulation is, however, very useful, in the sense that it allows to restrict our attention on c-concave functions, notably for two important classes of cost functions c: distances and squared-Euclidean norms.

6.10.3.2 Kantorovich-Rubinstein duality and Lipschitz potentials

A striking result illustrating the interest of c-concavity is provided when c is a metric d, namely when $p = 1$ in (6.241). In that case, one can prove (exploiting notably the triangle inequality of the d) that a d-concave function f is 1-Lipschitz (one has $|f(\mathbf{x}) - f(\mathbf{y})| \leq d(\mathbf{x}, \mathbf{y})$ for any $\mathbf{x}, \mathbf{y}$) and such that $\overline{f} = -f$. This result translates therefore in the following identity:

$$W_1(\mu, \nu) = \sup_{f 1\text{-Lipschitz}} \int_{\mathcal{X}} f \, (\mathrm{d}\mu - \mathrm{d}\nu). \tag{6.245}$$

This result has numerous practical applications. This supremum over 1-Lipschitz functions can be efficiently approximated using Wavelet coefficients of densities in low-dimensions [SJ08], or heuristically in more general cases by training neural networks parameterized to be 1-Lipschitz [ACB17] using ReLU activation functions, and bounds on the entries of the weight matrices.

6.10.3.3 Monge maps as gradients of convex functions: the Brenier theorem

Another application of c-concavity lies in the case $c(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|\|^2$, which corresponds, up to the factor $\frac{1}{2}$, to the squared W_2 distance used between densities in an Euclidean space. The remarkable result, shown first by [Bre91], is that the Monge map solving (M) between two measures for that cost (taken for granted μ is regular enough, here assumed to have a density w.r.t. Lebesgue measure) exists and is necessarily the gradient of a convex function. In loose terms, one can show that

$$T^\star = \arg \min_{T : T_\sharp \mu = \nu} \int_{\mathcal{X}} \tfrac{1}{2} \|\mathbf{x} - T(\mathbf{x})\|_2^2 \, \mu(\mathrm{d}\mathbf{x}). \tag{M}$$

exists, and is the gradient of a convex function $u : \mathbb{R}^d \to \mathbb{R}$, namely $T^\star = \nabla u$. Conversely, for any convex function u, the optimal transport map between μ and the displacement $\nabla u_{\#}\mu$ is necessarily equal to ∇u.

We provide a sketch of the proof: one can always exploit, for any reasonable cost function c (e.g. lower bounded and lower semi continuous), primal-dual relationships: Consider an optimal coupling $P^\star$ for (K2), as well as an optimal c-concave dual function $f^\star$ for (DK2). This implies in particular that $(f^\star, g^\star = \overline{f^\star})$ is optimal for (DK2). Complementary slackness conditions for this pair

of linear programs imply that if $\mathbf{x}_0, \mathbf{y}_0$ is in the support of $P^\star$, then necessarily (and sufficiently) $f^\star(\mathbf{x}_0) + \overline{f^\star}(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0)$. Suppose therefore that $\mathbf{x}_0, \mathbf{y}_0$ is indeed in the support of $P^\star$. From the equality $f^\star(\mathbf{x}_0) + \overline{f^\star}(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0)$ one can trivially obtain that $\overline{f^\star}(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0) - f^\star(\mathbf{x}_0)$. Yet, recall also that, by definition, $\overline{f^\star}(\mathbf{y}_0) = \inf_{\mathbf{x}} c(\mathbf{x}, \mathbf{y}_0) - f^\star(\mathbf{x})$. Therefore, $\mathbf{x}_0$ has the special property that it *minimizes* $\mathbf{x} \to c(\mathbf{x}, \mathbf{y}_0) - f^\star(\mathbf{x})$. If, at this point, one recalls that c is assumed in this section to be $c(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|\|^2$, one has therefore that $\mathbf{x}_0$ verifies

$$\mathbf{x}_0 \in \operatorname*{argmin}_{\mathbf{x}} \tfrac{1}{2}\|\mathbf{x} - \mathbf{y}_0\|^2 - f^\star(\mathbf{x}). \tag{6.246}$$

Assuming $f^\star$ is differentiable, which one can prove by c-concavity, this yields the identity

$$\mathbf{y}_0 - \mathbf{x}_0 - \nabla f^\star(\mathbf{x}_0) = 0 \Rightarrow \mathbf{y}_0 = \mathbf{x}_0 - \nabla f^\star(\mathbf{x}_0) = \nabla\left(\tfrac{1}{2}\|\cdot\|^2 - f^\star\right)(\mathbf{x}_0). \tag{6.247}$$

Therefore, if $(\mathbf{x}_0, \mathbf{y}_0)$ is in the support of $P^\star$, $\mathbf{y}_0$ is uniquely determined, which proves $P^\star$ is in fact a Monge map "disguised" as a coupling, namely

$$P^\star = \left(\mathrm{Id}, \nabla\left(\tfrac{1}{2}\|\cdot\|^2 - f^\star\right)\right)_\sharp \mu\,. \tag{6.248}$$

The end of the proof can be worked out as follows: For any function $h : \mathcal{X} \to \mathbf{R}$, one can show, using the definitions of c-transforms and the Legendre transform, that $\frac{1}{2}\|\cdot\|^2 - h$ is convex if and only if h is c-concave. An intermediate step in that proof relies on showing that $\frac{1}{2}\|\cdot\|^2 - \overline{h}$ is equal to the Legendre transform of $\frac{1}{2}\|\cdot\|^2 - h$. The function $\frac{1}{2}\|\cdot\|^2 - f^\star$ above is therefore convex, by c-concavity of $f^\star$, and the optimal transport map is itself the gradient of a convex function.

Knowing that an optimal transport map for the squared-Euclidean cost is necessarily the gradient of a convex function can prove very useful to solve (DK2). Indeed, this knowledge can be leveraged to restrict estimation to relevant families of functions, namely gradients of input-convex neural networks[AXK17], as proposed in [Mak+20] or [Kor+20], as well as arbitrary convex functions with desirable smoothness and strong-convexity constants [PdC20].

6.10.3.4 Closed forms for univariate and Gaussian distributions

Many metrics between probability distributions have closed form expressions for simple cases. The Wasserstein distance is no exception, and can be computed in close form in two important scenarios. When distributions are univariate and the cost $c(\mathbf{x}, \mathbf{y})$ is either a convex function of the difference $\mathbf{x} - \mathbf{y}$, or when $\partial c / \partial \mathbf{x} \partial \mathbf{y} < 0$ a.e., then the Wasserstein distance is essentially a comparison between the quantile functions of μ and ν. Recall that for a measure ρ, its quantile function Q_ρ is a function that takes values in $[0, 1]$ and is valued in the support of ρ, and corresponds to the (generalized) inverse map of F_ρ, the cumulative distribution function (cdf) of ρ. With these notations, one has that

$$\mathrm{OT}_c(\mu, \nu) = \int_{[0,1]} c\left(Q_\mu(u), Q_\nu(u)\right) \mathrm{d}u \tag{6.249}$$

In particular, when c is $\mathbf{x}, \mathbf{y} \mapsto |\mathbf{x} - \mathbf{y}|$ then $\mathrm{OT}_c(\mu, \nu)$ corresponds to the Kolmogorov-Smirnov statistic, namely the area between the cdf of μ and that of ν. If c is $\mathbf{x}, \mathbf{y} \mapsto (\mathbf{x} - \mathbf{y})^2$, we recover simply the squared-Euclidean norm between the quantile functions of μ and ν. Note finally that the Monge map is also available in closed form, and is equal to $Q_\nu \circ F_\mu$.

The second closed form applies to so-called elliptically contoured distributions, chiefly among them Gaussian multivariate distributions[Gel90]. For two Gaussians $\mathcal{N}(\mathbf{m}_1, \Sigma_1)$ and $\mathcal{N}(\mathbf{m}_2, \Sigma_2)$ their 2-Wasserstein distance decomposes as

$$W_2^2(\mathcal{N}(\mathbf{m}_1, \Sigma_1), \mathcal{N}(\mathbf{m}_2, \Sigma_2) = \|\mathbf{m}_1 - \mathbf{m}_2\|^2 + \mathcal{B}^2(\Sigma_1, \Sigma_2) \tag{6.250}$$

where the Bures metric $\mathcal{B}$ reads:

$$\mathcal{B}^2(\Sigma_1, \Sigma_2) = \text{trace}\left(\Sigma_1 + \Sigma_2 - 2\left(\Sigma_1^{\frac{1}{2}}\Sigma_2\Sigma_1^{\frac{1}{2}}\right)^{\frac{1}{2}}\right). \tag{6.251}$$

Notice in particular that these quantities are well-defined even when the covariance matrices are not invertible, and that they collapse to the distance between means as both covariances become 0. When the first covariance matrix is invertible, one has that the optimal Monge map is given by

$$T \triangleq \mathbf{x} \mapsto A(\mathbf{x} - \mathbf{m}_1) + \mathbf{m}_2, \text{ where } A \triangleq \Sigma_1^{-\frac{1}{2}}\left(\Sigma_1^{\frac{1}{2}}\Sigma_2\Sigma_1^{\frac{1}{2}}\right)^{\frac{1}{2}}\Sigma_1^{-\frac{1}{2}} \tag{6.252}$$

It is easy to show that $T^\star$ is indeed optimal: The fact that $T_\sharp\mathcal{N}(\mathbf{m}_1, \Sigma_1) = \mathcal{N}(\mathbf{m}_2, \Sigma_2)$ follows from the knowledge that the affine push-forward of a Gaussian is another Gaussian. Here T is designed to push precisely the first Gaussian onto the second (and A designed to recover random variables with variance Σ_2 when starting from random variables with variance Σ_1). The optimality of T can be recovered by simply noticing that is the gradient of a convex quadratic form, since A is positive definite, and closing this proof using the Brenier theorem above.

6.10.3.5 Exact evaluation using linear program solvers

We have hinted, using duality and c-concavity, that methods based on stochastic optimization over 1-Lipschitz or convex neural networks can be employed to estimate Wasserstein distances when c is the Euclidean distance or its square. These approaches are, however, non-convex and can only reach local optima. Apart from these two cases, and the closed forms provided above, the only reliable approach to compute Wasserstein distances appears when both μ and ν are discrete measures: In that case, one can instantiate and solve the discrete (K) problem, or its dual (DK) formulation. The primal problem is a canonical example of network flow problems, and can be solved with the network-simplex method in $O(nm(n+m)\log(n+m))$ complexity [AMO88], or, alternatively, with the comparable auction algorithm [BC89]. These approaches suffer from computational limitations: their cubic cost is intractable for large scale scenarios; their combinatorial flavor makes it harder to solve to parallelize simultaneously the computation of multiple optimal transport problems with a common cost matrix C.

An altogether different issue, arising from statistics, should discourage further users from using these LP formulations, notably in high-dimensional settings. Indeed, the bottleneck practitioners will most likely encounter when using (K) is that, in most scenarios, their goal will be to approximate the distance between two continuous measures μ, ν using only i.i.d samples contained in empirical measures $\hat{\mu}_n, \hat{\nu}_n$. Using (K) to approximate the corresponding (K2) is doomed to fail, as various results [FG15] have shown in relevant settings (notably for measures in $\mathbb{R}^q$) that the *sample complexity*

of the estimator provided by (K) to approximate (K2) is of order $1/n^{1/q}$. In other words, the gap between $W_2(\mu,\nu)$ and $W_2(\hat{\mu}_n,\hat{\nu}_n)$ is large on expectation, decreases extremely slowly as n increases in high dimensions, and solving exactly (K2) between these samples is compute power that is mostly wasted on overfitting. To address this curse of dimensionality, it is therefore extremely important in practice to approach (K2) using a more careful strategy, one that involves regularizations that can leverage prior assumptions on μ and ν. While all approaches outlined above using neural networks can be interpreted under this light, we focus in the following on a specific approach that results in a convex problem that is relatively simple to implement, embarassingly parallel and with quadratic complexity.

6.10.3.6 Obtaining smoothness using entropic regularization

A computational approach to speed-up the resolution of (K) was proposed in [Cut13], building on earlier contributions [Wil69; KY94] and a filiation to the Schrödinger bridge problem in the special case where $c = d^2$ [Léo14]. The idea rests upon regularizing the transportation cost by the Kullback-Leibler divergence of the coupling to the product measure of μ, ν,

$$W_{c,\gamma}(\mu,\nu) \triangleq \inf_{\pi\in\Pi(\mu,\nu)} \int_{\mathcal{X}^2} d(\mathbf{x},\mathbf{y})^p \, \mathrm{d}\pi(\mathbf{x},\mathbf{y}) + \gamma D_{\mathbb{KL}}(\pi \| \mu\otimes\nu). \tag{6.253}$$

When instantiated on discrete measures, this problem is equivalent to the following γ-strongly convex problem on the set of transportation matrices (which should be compared to (K))

$$\mathrm{OT}_{C,\gamma}(\mathbf{a},\mathbf{b}) = \min_{P\in\mathbf{R}_+^{n\times m}, P\mathbf{1}_m=\mathbf{a}, P^T\mathbf{1}_m=\mathbf{b}} \langle P, C\rangle \triangleq \sum_{i,j} P_{ij}C_{ij} - \gamma\mathbb{H}(P) + \gamma\left(\mathbb{H}(\mathbf{a}) + \mathbb{H}(\mathbf{b})\right), \tag{6.254}$$

Where and is itself equivalent to the following dual problem (which should be compared to (DK))

$$\mathrm{OT}_{C,\gamma}(\mathbf{a},\mathbf{b}) = \max_{\mathbf{f}\in\mathbb{R}^n, \mathbf{g}\in\mathbb{R}^m} \mathbf{f}^T\mathbf{a} + \mathbf{g}^T\mathbf{b} - \gamma(e^{\mathbf{f}/\gamma})^T K e^{\mathbf{g}/\gamma}) + \gamma\left(1 + \mathbb{H}(\mathbf{a}) + \mathbb{H}(\mathbf{b})\right) \tag{6.255}$$

and $K \triangleq e^{-C/\gamma}$ is the elementwise exponential of $-C/\gamma$. This regularization has several benefits. Primal-dual relationships show an explicit link between the (unique) solution $P^\star_\gamma$ and a pair of optimal dual variables $(\mathbf{f}^\star, \mathbf{g}^\star)$ as

$$P^\star_\gamma = \mathrm{diag}(e^{\mathbf{f}/\gamma}) K \mathrm{diag}(e^{\mathbf{g}/\gamma}) \tag{6.256}$$

Problem(6.255) can be solved using a fairly simple strategy that has proved very sturdy in practice: a simple block-coordinate ascent (optimizing alternatively the objective in $\mathbf{f}$ and then $\mathbf{g}$), resulting in the famous Sinkhorn algorithm [Sin67], here expressed with log-sum-exp updates, starting from an arbitrary initialization for $\mathbf{g}$, to carry out these two updates sequentially, until they converge:

$$\mathbf{f} \leftarrow \gamma\log\mathbf{a} - \gamma\log Ke^{\mathbf{g}/\gamma} \qquad \mathbf{g} \leftarrow \gamma\log\mathbf{b} - \gamma\log K^T e^{\mathbf{f}/\gamma} \tag{6.257}$$

The convergence of this algorithm has been amply studied (see [CK21] and references therein). Convergence is naturally slower as γ decreases, reflecting the hardness of approaching LP solutions, as studied in [AWR17]. This regularization also has statistical benefits since, as argued in [Gen+19], the sample complexity of the regularized Wasserstein distance improves to a $O(1/\sqrt{n})$ regime, with, however, a constant in $1/\gamma^{q/2}$ that deteriorates as dimension grows.

6.11 Submodular optimization

This section was written by Jeff Bilmes.

This section provides a brief overview of submodularity in machine learning.[5] Submodularity has an extremely simple definition. However, the "simplest things are often the most complicated to understand fully" [Sam74], and while submodularity has been studied extensively over the years, it continues to yield new and surprising insights and properties, some of which are extremely relevant to data science, machine learning, and artificial intelligence. A submodular function operates on subsets of some finite *ground set*, V. Finding a guaranteed good subset of V would ordinarily require an amount of computation exponential in the size of V. Submodular functions, however, have certain properties that make optimization either tractable or approximable where otherwise neither would be possible. The properties are quite natural, however, so submodular functions are both flexible and widely applicable to real problems. Submodularity involves an intuitive and natural diminishing returns property, stating that adding an element to a smaller set helps more than adding it to a larger set. Like convexity, submodularity allows one to efficiently find provably optimal or near-optimal solutions. In contrast to convexity, however, where little regarding maximization is guaranteed, submodular functions can be both minimized and (approximately) maximized. Submodular maximization and minimization, however, require very different algorithmic solutions and have quite different applications. It is sometimes said that submodular functions are a discrete form of convexity. This is not quite true, as submodular functions are like both convex and concave functions, but also have properties that are similar simultaneously to both convex and concave functions at the same time, but then some properties of submodularity are like neither convexity nor concavity. Convexity and concavity, for example, can be conveyed even as univariate functions. This is impossible for submodularity, as submodular functions are defined based only on the response of the function to changes amongst different variables in a multidimensional discrete space.

6.11.1 Intuition, Examples, and Background

Let us define a *set function* $f : 2^V \to \mathbb{R}$ as one that assigns a value to every subset of V. The notation 2^V is the power set of V, and has size $2^{|V|}$ which means that f lives in space $\mathbb{R}^{2^n}$ — i.e., since there are 2^n possible subsets of V, f can return 2^n distinct values. We use the notation $X + v$ as shorthand for $X \cup \{v\}$. Also, the value of an element in a given context is so widely used a concept, we have a special notation for it — the incremental value *gain* of v in the context if X is defined as $f(v|X) = f(X + v) - f(X)$. Thus, while $f(v)$ is the value of element v, $f(v|X)$ is the value of element v if you already have X. We also define the gain of set X in the context of Y as $f(X|Y) = f(X \cup Y) - f(Y)$.

6.11.1.1 Coffee, Lemon, Milk, Tea

As a simple example, will explore the manner in which the value of everyday items may interact and combine, namely coffee, lemon, milk, and tea. Consider the value relationships amongst the four items coffee (c), lemon (l), milk (m), and tea (t) as shown in Figure 6.19. Suppose you just woke up,

5. A greatly extended version of the material in this section may be found at [Bil22].

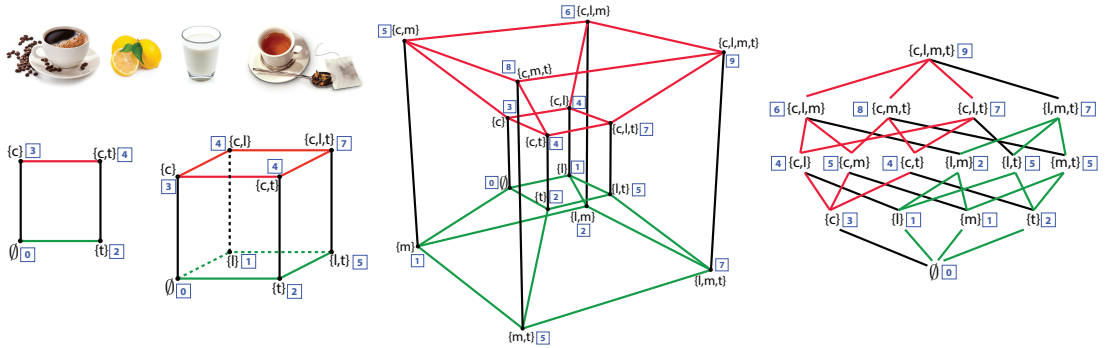


Figure 6.19: The value relationships between coffee (c), lemon (l), milk (m), and tea (t). On the left, we first see a simple square showing the relationships between coffee and tea, and see that they are substitutive (or submodular). In this, and all of the shapes, the vertex label set is indicated in curly braces and the value at that vertex is a blue integer in a box. We next see a three-dimensional cube that adds lemon to coffee and tea set. We see that tea and lemon are complementary (supermodular) but coffee and lemon are additive (modular, or independent). We next see a four-dimensional hypercube (tesseract) showing all of the value relationships described in the text. The four-dimensional hypercube is also shown as a lattice (on the right) showing the same relationships as well as two (red and green, also shown in the tesseract) of the eight three-dimensional cubes contained within.

and there is a function $f : 2^V \to \mathbb{R}$ that provides the average valuation for any subset of the items in V where $V = \{c, l, m, t\}$. You can think of this function as giving the average price a typical person would be willing to pay for any subset of items. Since nothing should cost nothing, we would expect that $f(\emptyset) = 0$. Clearly, one needs either coffee or tea in the morning, so $f(c) > 0$ and $f(t) > 0$, and coffee is usually more expensive than tea, so that $f(c) > f(t)$ pound for pound. Also more items cost more, so that, for example, $0 < f(c) < f(c, m) < f(c, m, t) < f(c, l, m, t)$. Thus, the function f is strictly *monotone*, or $f(X) < f(Y)$ whenever $X \subset Y$.

The next thing we note is that coffee and tea may substitute for each other – they both have the same effect, waking you up. They are mutually redundant, and they decrease each other's value since once you have had a cup of coffee, a cup of tea is less necessary and less desirable. Thus, $f(c, t) < f(c) + f(t)$, which is known as a *subadditive* relationship, the whole is less than the sum of the parts. On the other hand, some items complement each other. For example, milk and coffee are better combined together than when both are considered in isolation, or $f(m, c) > f(m) + f(c)$, a *superadditive* relationship, the whole is more than the sum of the parts. A few of the items do not affect each others' price. For example, lemon and milk cost the same together as apart, so $f(l, m) = f(l) + f(m)$, an *additive* or *modular* relationship — such a relationship is perhaps midway between a subadditive and a superadditive relationship, and can be seen as a form of independence.

Things become more interesting when we consider three or more items together. For example, once you have tea, lemon becomes less valuable when you acquire milk since there might be those that prefer milk to lemon in their tea. Similarly, milk becomes less valuable once you have acquired lemon since there are those who prefer lemon in their tea to milk. So, once you have tea, lemon and milk are substitutive, you would never use both as the lemon would only curdle the milk. These are *submodular* relationships, $f(l|m, t) < f(l|t)$ and $f(m|l, t) < f(m|t)$ each of which implies that

$f(l,t) + f(m,t) > f(l,m,t) + f(t)$. The value of lemon (respectively milk) with tea decreases in the larger context of having milk (respectively lemon) with tea, typical of submodular relationships.

Not all of the items are in a submodular relationship, as sometimes the presence of an item can increase the value of another item. For example, once you have milk, then tea becomes still more valuable when you also acquire lemon, since tea with the choice of either lemon or milk is more valuable than tea with the option only of milk. Similarity, once you have milk, lemon becomes more valuable when you acquire tea, since lemon with milk alone is not nearly as valuable as lemon with tea, even if milk is at hand. This means that $f(t|l,m) > f(t|m)$ and $f(l|t,m) > f(l|m)$ implying $f(l,m) + f(m,t) < f(l,m,t) + f(m)$. These are known as *supermodular* relationships, where the value increases as the context increases.

We have asked for a set of relationships amongst various subsets of the four items $V = \{c, l, m, t\}$, Is there a function that offers a value to each $X \subseteq V$ that satisfies all of the above relationships? Figure 6.19 in fact shows such a function. On the left, we see a two-dimensional square whose vertices indicate the values over subsets of $\{c, t\}$ and we can quickly verify that the sum of the blue boxes on north-west (corresponding to $f(\{c\})$) and south-east corners (corresponding to $f(\{t\})$) is greater than the sum of the north-east and south-west corners, expressing the required submodular relationship. Next on the right is a three-dimensional cube that adds the relationship with lemon. Now we have six squares and we see that the values at each of the vertices all satisfy the above requirements — we verify this by considering the valuations at the four corners of every one of the six faces of the cube. Since $|V| = 4$ we need a four-dimensional hypercube to show all values and this may be shown in two ways. It is first shown as a tesseract, a well-known three-dimensional projection of a four-dimensional hypercube. In the figure, all vertices are labeled both with subsets of V as well as the function value $f(X)$ as the blue number in a box. The figure on the right shows a *lattice* version of the four-dimensional hypercube, where corresponding three-dimensional cubes are shown in green and red.

We thus see that a set function is defined for all subsets of a ground set, and that they correspond to valuations at all vertices of the hypercube. For the particular function over valuations of subsets of coffee, lemon, milk, and tea, we have seen submodular, supermodular, and modular relationships all in one function. Therefore, the overall function f defined in Figure 6.19 is neither submodular, supermodular, nor modular. For combinatorial auctions, there is often a desire to have a diversity of such manners of relationships [LLN06] — representation of these relationships can be handled by a difference of submodular functions [NB05; IB12] or a sum of a submodular and supermodular function [BB18] (further described below). In machine learning, however, most of the time we are interested in functions that are submodular (or modular, or supermodular) everywhere.

6.11.2 Submodular Basic Definitions

For a function to be submodular, it must satisfy the submodular relationship for all subsets. We arrive at the following definition.

Definition 6.11.1 (**Submodular** Function)**.** *A given set function $f : 2^V \to \mathbb{R}$ is submodular if for all $X, Y \subseteq V$, we have the following inequality:*

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y) \tag{6.258}$$

There are also many other equivalent definitions of submodularity [Bil22] some of which are more intuitive and easier to understand. For example, submodular functions are those set functions that satisfy the property of diminishing returns. If we think of a function $f(X)$ as measuring the value of a set X that is a subset of a larger set of data items $X \subseteq V$, then the submodular property means that the incremental "value" of adding a data item v to set X decreases as the size of X grows. This gives us a second classic definition of submodularity.

Definition 6.11.2 (Submodular Function via Diminishing Returns)**.** *A given set function $f : 2^V \to \mathbb{R}$ is submodular if for all $X, Y \subseteq V$, where $X \subseteq Y$ and for all $v \notin Y$, we have the following inequality:*

$$f(X + v) - f(X) \geq f(Y + v) - f(Y) \tag{6.259}$$

The property that the incremental value of lemon with tea is less than the incremental value of lemon once milk is already in the tea is equivalent to Equation 6.258 if we set $X = \{m, t\}$ and $Y = \{l, t\}$ (i.e. $f(m, t) + f(l, t) > f(l, m, t) + f(t)$). It is naturally also equivalent to Equation 6.259 if we set $X = \{t\}$, $Y = \{m, t\}$, and with $v = l$ (i.e., $f(l|m, t) < f(l|t)$).

There are many functions that are submodular, one famous one being Shannon entropy seen as a function of subsets of random variables. We first point out that there are non-negative (i.e., $f(A) \geq 0, \forall A$), monotone non-decreasing (i.e., $f(A) \leq f(B)$ whenever $A \subseteq B$) submodular functions that are not entropic [Yeu91b; ZY97; ZY98], so submodularity is not just a trivial restatement of the class of entropy functions. When a function is monotone non-decreasing, submodular, and *normalized* so that $f(\emptyset) = 0$, it is often referred to as a **polymatroid function**. Thus, while the entropy function is a polymatroid function, it does not encompass all polymatroid functions even though all polymatroid functions satisfy the properties Claude Shannon mentioned as being natural for an "information" function (see Section 6.11.7).

A function f is supermodular if and only if $-f$ is submodular. If a function is both submodular and supermodular, it is known as a *modular* function. It is always the case that modular functions may take the form of a vector-scalar pair (m, c) where $m : 2^V \to \mathbb{R}$ and where $c \in \mathbb{R}$ is a constant, and where for any $A \subseteq V$, we have that $m(A) = c + \sum_{v \in A} m_v$. If the modular function is normalized, so that $m(\emptyset) = 0$, then $c = 0$ and the modular function can be seen simply as a vector $m \in \mathbb{R}^V$. Hence, we sometimes say that the modular function $x \in \mathbb{R}^V$ offers a value for set A as the partial sum $x(A) = \sum_{v \in A} x(v)$. Many combinatorial problems use modular functions as objectives. For example, the graph cut problem uses modular function defined over the edges, judges a cut in a graph as the modular function applied to the edges that comprise the cut.

As can be seen from the above, and by considering Figure 6.19, a submodular function, and in fact any set function, $f : 2^V \to \mathbb{R}$ can be seen as a function defined only on the vertices of the n-dimensional unit hypercube $[0, 1]^n$. Given any set $X \subseteq V$, we define $\mathbf{1}_X \in \{0, 1\}^V$ to be the characteristic vector of set X defined as $\mathbf{1}_X(v) = 1$ if $v \in X$ and $\mathbf{1}_X(v) = 0$ otherwise. This gives us a way to map from any set $X \subseteq V$ to a binary vector $\mathbf{1}_X$. We also see that $\mathbf{1}_X$ is itself a modular function since $\mathbf{1}_X \in \{0, 1\}^V \subset \mathbb{R}^V$.

Submodular functions share a number of properties in common with both convex and concave functions [Lov83], including wide applicability, generality, multiple representations, and closure under a number of common operators (including mixtures, truncation, complementation, and certain convolutions). There is one important submodular closure property that we state here — that is that if we take a non-negative weighted (or conical) combinations of submodular functions, we preserve submodularity. In other words, if we have a set of k submodular functions, $f_i : 2^V \to \mathbb{R}$, $i \in [k]$, and

we form $f(X) = \sum_{i=1}^{k} \omega_i f_i(X)$ where $\omega_i \geq 0$ for all i, then Definition 6.11.1 immediately implies that f is also submodular. When we consider Definition 6.11.1, we see that submodular functions live in a cone in 2^n-dimensional space defined by the intersection of an exponential number of half-spaces each one of which is defined by one of the inequalities of the form $f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$. Each submodular function is therefore a point in that cone. It is therefore not surprising that taking conical combinations of such points stays within this cone.

6.11.3 Example Submodular Functions

As mentioned above, there are many functions that are submodular besides entropy. Perhaps the simplest such function is $f(A) = \sqrt{|A|}$ which is the composition of the square-root function (which is concave) with the cardinality $|A|$ of the set A. The gain function is $f(A + v) - f(A) = \sqrt{k+1} - \sqrt{k}$ if $|A| = k$, which we know to be a decreasing in k, thus establishing the submodularity of f. In fact, if $\phi : \mathbb{R} \to \mathbb{R}$ is any concave function, then $f(A) = \phi(|A|)$ will be submodular for the same reason.[6] Generalizing this slightly further, a function defined as $f(A) = \phi(\sum_{a \in A} m(a))$ is also submodular, whenever $m(a) \geq 0$ for all $a \in V$. This yields a composition of a concave function with a modular function $f(A) = \phi(m(A))$ since $\sum_{a \in A} m(a) = m(A)$. We may take sums of such functions as well as add a final modular function without losing submodularity, leading to $f(A) = \sum_{u \in U} \phi_u(\sum_{a \in A} m_u(a)) + \sum_{a \in A} m_{\pm}(a)$ where ϕ_u can be a distinct concave function for each u, $m_{u,a}$ is a non-negative real value for all u and a, and $m_{\pm}(a)$ is an arbitrary real number. Therefore, $f(A) = \sum_{u \in U} \phi_u(m_u(A)) + m_{\pm}(A)$ where m_u is a u-specific non-negative modular function and $m_{\pm}$ is an arbitrary modular function. Such functions are sometimes known as **feature-based** submodular functions [BB17] because U can be a set of non-negative features (in the machine-learning "bag-of-words" sense) and this function measures a form of dispersion over A as determined by the set of features U.

A function such as $f(A) = \sum_{u \in U} \phi_u(m_u(A))$ tends to award high diversity to a set A that has a high valuation by a distinct set of the features U. The reason is that, due to the concave nature of ϕ_u, any addition to the argument $m_u(A)$ by adding, say, v to A would diminish as A gets larger. In order to produce a set larger than A that has a much larger valuation, one must use a feature $u' \neq u$ that has not yet diminished as much.

Facility location is another well-known submodular function — perhaps an appropriate nickname would be the "k-means of submodular functions," due to its applicability, utility, ease-of-use (it needs only an affinity matrix), and similarity to k-medoids problems. The facility location function is defined using an affinity matrix as follows: $f(A) = \sum_{v \in V} \max_{a \in A} \text{sim}(a, v)$ where $\text{sim}(a, v)$ is a non-negative measure of the affinity (or similarity) between element a and v. Here, every element $v \in V$ must have a representative within the set A and the representative for each $v \in V$ is chosen to be the element $a \in A$ most similar to v. This function is also a form of dispersion or diversity function because, in order to maximize it, every element $v \in V$ must have some element similar to it in A. The overall score is then the sum of the similarity between each element $v \in V$ and v's representative. This function is monotone (since as A includes more elements to become $B \supseteq A$, it is possible only to find an element in B more similar to a given v than an element in A).

While the facility location looks quite different from a feature based function, it is possible

6. While we will not be extensively discussing supermodular functions in this section, $f(A) = \phi(|A|)$ is supermodular for any convex function ϕ.

to precisely represent any facility location function with a feature based function. Consider just $\max_{a \in A} x_a$ and, without loss of generality, assume that $0 \leq x_1 \leq x_2 \leq \cdots \leq x_n$. Then $\max_{a \in A} x_a = \sum_{i=1}^{n} y_i \min(|A \cap \{i, i+1, \ldots, n\}|, 1)$ where $y_i = x_i - x_{i-1}$ and we set $x_0 = 0$. We note that this is a sum of weighted concave composed with modular functions since $\min(\alpha, 1)$ is concave in α, and $|A \cap \{i, i+1, \ldots, n\}|$ is a modular function in A. Thus, the facility location function, a sum of these, is merely a feature based function.

Feature based functions, in fact, are quite expressive, and can be used to represent many different submodular functions including set cover and graph-based functions. For example, we can define a *set cover function*, given a set of sets $\{U_v\}_{v \in V}$, via $f(X) = \left|\bigcup_{v \in X} U_v\right|$. If $f(X) = |U|$ where $U = \bigcup_{v \in V} U_v$ then X indexes a set that fully covers U. This can also be represented as $f(X) = \sum_{u \in U} \min(1, m_u(X))$ where $m_u(X)$ is a modular function where $m_u(v) = 1$ if and only if $u \in U_v$ and otherwise $m_u(v) = 0$. We see that this is a feature based submodular function since $\min(1, x)$ is concave in x, and U is a set of features.

This construct can be used to produce the vertex cover function if we set $U = V$ to be the set of vertices in a graph, and set $m_u(v) = 1$ if and only if vertices u and v are adjacent in the graph and otherwise set $m_u(v) = 1$. Similarly, the edge cover function can be expressed by setting V to be the set of edges in a graph, U to be the set of vertices in the graph, and $m_u(v) = 1$ if and only edge v is incident to vertex u.

A generalization of the set cover function is the *probabilistic coverage* function. Let $\mathsf{P}[B_{u,v} = 1]$ be the probability of the presence of feature (or concept) u within element v. Here, we treat $B_{u,v}$ as a Bernoulli random variable for each element v and feature u so that $\mathsf{P}[B_{u,v} = 1] = 1 - \mathsf{P}[B_{u,v} = 0]$. Then we can define the probabilistic coverage function as $f(X) = \sum_{u \in U} f_u(X)$ where, for feature u, we have $f_u(X) = 1 - \prod_{v \in X}(1 - \mathsf{P}[B_{u,v} = 1])$ which indicates the degree to which feature u is "covered" by X. If we set $\mathsf{P}[B_{u,v} = 1] = 1$ if and only if $u \in U_v$ and otherwise $\mathsf{P}[B_{u,v} = 1] = 0$, then $f_u(X) = \min(1, m_u(X))$ and the set cover function can be represented as $\sum_{u \in U} f_u(X)$. We can generalize this in two ways. First, to make it softer and more probabilistic we allow $\mathsf{P}[B_{u,v} = 1]$ to be any number between zero and one. We also allow each feature to have a non-negative weight. This yields the general form of the probabilistic coverage function, which is defined by taking a weighted combination over all features: $f_u(X) = \sum_{u \in U} \omega_u f_u(X)$ where $\omega_u \geq 0$ is a weight for feature u. Observe that $1 - \prod_{v \in X}(1 - \mathsf{P}[B_{u,v} = 1]) = 1 - \exp(-m_u(X)) = \phi(m_u(X))$ where m_u is a modular function with evaluation $m_u(X) = \sum_{v \in X} \log\big(1/(1 - \mathsf{P}[B_{u,v} = 1])\big)$ and for $z \in \mathbb{R}$, $\phi(z) = 1 - \exp(-z)$ is a concave function. Thus, the probabilistic coverage function (and its set cover specialization) is also feature based function.

Another common submodular function is the graph cut function. Here, we measure the value of a subset of V by the edges that cross between a set of nodes and all but that set of nodes. We are given an undirected non-negative weighted graph $\mathcal{G} = (V, E, w)$ where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $w \in \mathbb{R}_+^E$ are non-negative edge weights corresponding to symmetric matrix (so $w_{i,j} = w_{j,i}$). For any $e \in E$, we have $e = \{i, j\}$ for some $i, j \in V$ with $i \neq j$, the graph cut function $f : 2^V \to \mathbb{R}$ is defined as $f(X) = \sum_{i \in X, j \in \bar{X}} w_{i,j}$ where $w_{i,j} \geq 0$ is the weight of edge $e = \{i, j\}$ ($w_{i,j} = 0$ if the edge does not exist), and where $\bar{X} = V \setminus X$ is the complement of set X. Notice that

we can write the graph cut function as follows:

$$f(X) = \sum_{i \in X, j \in \bar{X}} w_{i,j} = \sum_{i,j \in V} w_{i,j} \mathbf{1}\{i \in X, j \in \bar{X}\} \tag{6.260}$$

$$= \frac{1}{2} \sum_{i,j \in V} w_{i,j} \min(|X \cap \{i,j\}|, 1) + \frac{1}{2} \sum_{i,j \in V} w_{i,j} \min(|(V \setminus X) \cap \{i,j\}|, 1) - \frac{1}{2} \sum_{i,j \in V} w_{i,j} \tag{6.261}$$

$$= \tilde{f}(X) + \tilde{f}(V \setminus X) - \tilde{f}(V) \tag{6.262}$$

where $\tilde{f}(X) = \frac{1}{2} \sum_{i,j \in V} w_{i,j} \min(|X \cap \{i,j\}|, 1)$. Therefore, since $\min(\alpha, 1)$ is concave, and since $m_{i,j}(X) = |X \cap \{i,j\}|$ is modular, $\tilde{f}(X)$ is submodular for all i, j. Also, since $\tilde{f}(X)$ is submodular, so is $\tilde{f}(V \setminus X)$ (in X). Therefore, the graph cut function can be expressed as a sum of non-normalized feature-based functions. Note that here the second modular function is not normalized and is non-increasing, and also we subtract the constant $\tilde{f}(V)$ to achieve equality.

Another way to view the graph cut function is to consider the non-negative weights as a modular function defined over the edges. That is, we view $w \in \mathbb{R}_+^E$ as a modular function $w : 2^E \to \mathbb{R}_+$ where for every $A \subseteq E$, $w(A) = \sum_{e \in A} w(e)$ is the weight of the edges A where $w(e)$ is the weight of edge e. Then the graph cut function becomes $f(X) = w(\{(a,b) \in E : a \in X, b \in X \setminus X\})$. We view $\{(a,b) \in E : a \in X, b \in X \setminus X\}$ as a set-to-set mapping function, that maps subsets of nodes to subsets of edges, and the edge weight modular function w measures the weight of the resulting edges. This immediately suggests that other functions can measure the weight of the resulting edges as well, including non-modular functions. One example is to use a polymatroid function itself leading $h(X) = g(\{(a,b) \in E : a \in X, b \in X \setminus X\})$ where $g : 2^E \to \mathbb{R}_+$ is a submodular function defined on subsets of edges. The function h is known as the **cooperative cut** function, and it is neither submodular nor supermodular in general but there are many useful and practical algorithms that can be used to optimize it [JB16] thanks to its internal yet exposed and thus available to exploit submodular structure.

While feature based functions are flexible and powerful, there is a strictly broader class of submodular functions, unable to be expressed by feature-based functions, and that are related to deep neural networks. Here, we create a recursively nested composition of concave functions with sums of compositions of concave functions. An example is $f(A) = \phi(\sum_{u \in U} \omega_u \phi_u(\sum_{a \in A} m_{u,a}))$, where ϕ is an outer concave function composed with a feature based function, with $m_{u,a} \geq 0$ and $\omega_u \geq 0$. This is known as a two-layer **deep submodular function** (DSF). A three-layer DSF has the form $f(A) = \phi(\sum_{c \in C} \omega_c \phi_c(\sum_{u \in U} \omega_{u,c} \phi_u(\sum_{a \in A} m_{u,a})))$. DSFs strictly expand the class of submodular functions beyond feature based functions, meaning that there are feature based functions that can not[BB17] represent deep submodular functions, even simple ones.

6.11.4 Submodular Optimization

Submodular functions, while discrete, would not be very useful if it was not possible to optimize over them efficiently. There are many natural problems in machine learning that can be cast as submodular optimization and that can be addressed relatively efficiently.

When one wishes to encourage diversity, information, spread, high complexity, independence, coverage, or dispersion, one usually will maximize a submodular function, in the form of $\max_{A \in \mathcal{C}} f(A)$ where $\mathcal{C} \subseteq 2^V$ is a constraint set, a set of subsets we are willing to accept as feasible solutions (more on this below).

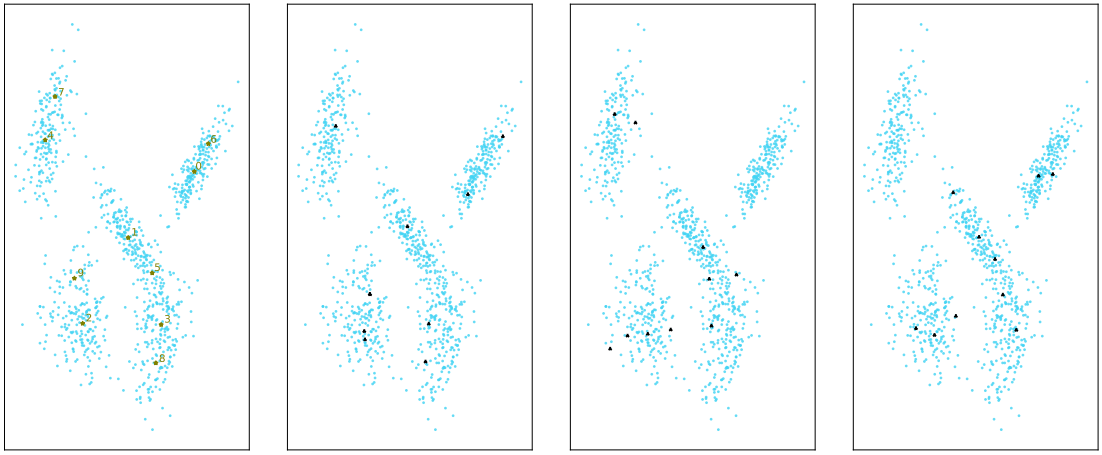


Figure 6.20: Far Left: cardinality constrained (to ten) submodular maximization of a facility location function over 1000 points in two dimensions. Similarities are based on a Gaussian kernel $sim(a, v) = \exp(-d(a, v))$ where $d(\cdot, \cdot)$ is a distance. Selected points are green stars and the greedy order is also shown next to each selected point. Right three plots: different uniformly-at-random subsets of size ten.

Why is submodularity, in general, a good model for diversity? Submodular functions are such that once you have some elements, any other elements not in your possession but that are similar to, explained by, or represented by the elements in your possession become less valuable. Thus, in order to maximize the function, one must choose other elements that are dissimilar to, or not well represented by, the ones you already have. That is, the elements similar to the ones you own are diminished in value relative to their original values, while the elements dissimilar to the ones you have do not have diminished value relative to their original values. Thus, maximizing a submodular function successfully involves choosing elements that are jointly dissimilar amongst each other, which is a definition of diversity. Diversity in general is a critically important aspect in machine learning and artificial intelligence. For example, bias in data science and machine learning can often be seen as some lack of diversity somewhere. Submodular functions have the potential to encourage (and even ensure) diversity, enhance balance, and reduce bias in artificial intelligence.

Note that in order for a submodular function to appropriately model diversity, it is important for it to be instantiated appropriately. Figure 6.20 shows an example in two dimensions. The plot compares the ten points chosen according to a facility location instantiated with a Gaussian kernel, along with the random samples of size ten. We see that the facility location selected points are more diverse and tend to cover the space much better than any of the randomly selected points each of which miss large regions of the space and/or show cases where points near each other are jointly selected.

When one wishes for homogeneity, conformity, low complexity, coherence, or cooperation, one will usually minimize a submodular function, in the form of $\min_{A \in \mathcal{C}} f(A)$. For example, if V is a set of pixels in an image, one might wish to choose a subset of pixels corresponding to a particular object over which the properties (i.e., color, luminance, texture) are relatively homogeneous. Finding a set X of size k, even if k is large, need not have a large valuation $f(X)$, in fact it could even have the

least valuation. Thus, semantic image segmentation could work even of the object being segmented and isolated consists of the majority of image pixels.

6.11.4.1 Submodular Maximization

While the cardinality constrained submodular maximization problem is NP complete [Fei98], it was shown in [NWF78; FNW78] that the very simple and efficient greedy algorithm finds an approximate solution guaranteed to be within $1-1/e \approx 0.63$ of the optimal solution. Moreover, the approximation ratio achieved by the simple greedy algorithm is provably the best achievable in polynomial time, assuming $P \neq NP$ [Fei98]. The greedy algorithm proceeds as follows: Starting with $X_0 = \emptyset$, we repeat the following greedy step for $i = 0 \dots (k-1)$:

$$X_{i+1} = X_i \cup (\operatorname*{argmax}_{v \in V \setminus X_i} f(X_i \cup \{v\})) \tag{6.263}$$

What the above approximation result means is that if $X^* \in \operatorname{argmax}\{f(X) : |X| \leq k\}$, and if $\tilde{X}$ is the result of the greedy procedure, then $f(\tilde{X}) \geq (1-1/e) f(X^*)$.

The $1-1/e$ guarantee is a powerful constant factor approximation result since it holds regardless of the size of the initial set V and regardless of which polymatroid function f is being optimized. It is possible to make this algorithm run extremely fast using various acceleration tricks [FNW78; NWF78; Min78].

A minor bit of additional information about a polymatroid function, however, can improve the approximation guarantee. Define the total curvature if the polymatroid function f as $\kappa = 1 - \min_{v \in V} f(v|V-v)/f(v)$ where we assume $f(v) > 0$ for all v (if not, we may prune them from the ground set since such elements can never improve a polymatroid function valuation). We thus have $0 \leq \kappa \leq 1$, and [CC84] showed that the greedy algorithm gives a guarantee of $\frac{1}{\kappa}(1-e^{-\kappa}) \geq 1-1/e$. In fact, this is an equality (and we get the same bound) when $\kappa = 1$, which is the fully curved case. As κ gets smaller, the bound improves, until we reach the $\kappa = 0$ case and the bound becomes unity. Observe that $\kappa = 0$ if and only if the function is modular, in which case the greedy algorithm is optimal for the cardinality constrained maximization problem. In some cases, non-submodular functions can be decomposed into components that each might be more amenable to approximation. We see below that any set function can be written as a difference of submodular [NB05; IB12] functions, and sometimes (but not always) a given h can be composed into a monotone submodular plus a monotone supermodular function, or a BP function [BB18], i.e., $h = f + g$ where f is submodular and g is supermodular. g has an easily computed quantity called the supermodular curvature $\kappa^g = 1 - \min_{v \in V} g(v)/g(v|V-v)$ that, together with the submodular curvature, can be used to produce an approximation ratio having the form $\frac{1}{\kappa}(1-e^{-\kappa(1-\kappa^g)})$ for greedily maximization of h.

6.11.4.2 Discrete Constraints

There are many other types of constraints one might desire besides a cardinality limitation. The next simplest constraint allows each element v to have a non-negative cost, say $m(v) \in \mathbb{R}_+$. In fact, this means that the costs are modular, i.e., the cost of any set X is $m(X) = \sum_{v \in X} m(v)$. A submodular maximization problem subject to a *knapsack constraint* then takes the form $\max_{X \subseteq V : m(X) \leq b} f(X)$ where b is a non-negative budget. While the greedy algorithm does not solve this problem directly, a

slightly modified cost-scaled version of the greedy algorithm [Svi04] does solve this problem for any set of knapsack costs. This has been used for various multi-document summarization tasks [LB11; LB12].

There is no single direct analogy for a convex set when one is optimizing over subsets of the set V, but there are a few forms of discrete constraints that are both mathematically interesting and that often occur repeatedly in applications.

The first form are the independent subsets of a matroids. The independent sets of a matroid are useful to represent a constraint set for submodular maximization [Cal+07; LSV09; Lee+10], $\max_{X \in \mathcal{I}} f(X)$, and this can be useful in many ways. We can see this by showing a simple example of what is known as a *partition matroid*. Consider a partition $V = \{V_1, V_2, \ldots, V_m\}$ of V into m mutually disjoint subsets that we call blocks. Suppose also that for each of the m blocks, there is a positive integer limit ℓ_i for $i \in [m]$. Consider next the set of sets formed by taking all subsets of V such that each subset has intersection with V_i no more than ℓ_i for each i. I.e., consider

$$\mathcal{I}_{\mathrm{p}} = \{X : \forall i \in [m], |V_i \cap X| \leq \ell_i\}. \tag{6.264}$$

Then $(V, \mathcal{I}_{\mathrm{p}})$ is a matroid. The corresponding submodular maximization problem is a natural generalization of the cardinality constraint in that, rather than having a fixed number of elements beyond which we are uninterested, the set of elements V is organized into groups, and here we have a fixed per-group limit beyond which we are uninterested. This is useful for fairness applications since the solution must be distributed over the blocks of the matroid. Still, there are many much more powerful types of matroids that one can use [Oxl11; GM12].

Regardless of the matroid, the problem $\max_{X \in \mathcal{I}} f(X)$ can be solved, with a $1/2$ approximation factor, using the same greedy algorithm as above [NWF78; FNW78]. Indeed, the greedy algorithm has an intimate relationship with submodularity, a fact that is well studied in some of the seminal works on submodularity [Edm70; Lov83; Sch04]. It is also possible to define constraints consisting of an *intersection of matroids*, meaning that the solution must be simultaneously independent in multiple distinct matroids. Adding on to this, we might wish a set to be independent in multiple matroids and also satisfy a knapsack constraint. Knapsack constraints are not matroid constraints, since there can be multiple maximal cost solutions that are not the same size (as must be the case in a matroid). It is also possible to define discrete constraints using level sets of another completely different submodular function [IB13] — given two submodular functions f and g, this leads to optimization problems of the form $\max_{X \subseteq V : g(X) \leq \alpha} f(X)$ (the submodular cost submodular knapsack, or SCSK, problem) and $\min_{X \subseteq V : g(X) \geq \alpha} f(X)$ (the submodular cost submodular cover, or SCSC, problem). Other examples include covering constraints [IN09], and cut constraints [JB16]. Indeed, the type of constraints on submodular maximization for which good and scalable algorithms exist is quite vast, and still growing.

One last note on submodular maximization. In the above, the function f has been assumed to be a polymatroid function. There are many submodular functions that are not monotone [Buc+12]. One example we saw before, namely the graph cut function. Another example is the log of the determinant (log-determinant) of a submatrix of a positive-definite matrix (which is the Gaussian entropy plus a constant). Suppose that $\mathbf{M}$ is an $n \times n$ symmetric positive-definite (SPD) matrix, and that $\mathbf{M}_X$ is a row-column submatrix (i.e., it is an $|X| \times |X|$ matrix consisting of the rows and columns of $\mathbf{M}$ consisting of the elements in X). Then the function defined as $f(X) = \log\det(\mathbf{M}_X)$ is submodular but not necessarily monotone non-decreasing. In fact, the submodularity of the log-determinant function is one of the reasons that *determinantal point processes* (DPPs), which

instantiate probability distributions over sets in such a way that high probability is given to those subsets that are diverse according to $\mathbf{M}$, are useful for certain tasks where we wish to probabilistically model diversity [KT11]. Diversity of a set X here is measured by the volume of the paralellepiped which is known to be computed as the determinant of the submatrix $\mathbf{M}_X$ and taking the log of this volume makes the function submodular in X. A DPP in fact is an example of a log-submodular probabilistic model (more in Section 6.11.10).

6.11.4.3 Submodular Function Minimization

In the case of a polymatroid function, unconstrained minimization is again trivial. However, even in the unconstrained case, the minimization of an arbitrary (i.e., not necessarily monotone) submodular function $\min_{X \subseteq V} f(X)$ might seem hopelessly intractable. Unconstrained submodular maximization is NP-hard (albeit approximable) and this is not surprising given that there are an exponential number of sets needing to be considered. Remarkably, submodular minimization does not require exponential computation, is not NP-hard, and in fact, there are polynomial time algorithms for doing so, something that is not at all obvious. This is one of the important characteristics that submodular functions share with convex functions, their common amenability to minimization. Starting in the very late 1960s, and spearheaded by individuals such as Jack Edmonds [Edm70], there was a concerted effort in the discrete mathematics community in search of either an algorithm that could minimize a submodular function in polynomial time or a proof that such a problem was NP-hard. The nut was finally cracked in a classic paper [GLS81] on the ellipsoid algorithm that gave a polynomial time algorithm for submodular function minimization (SFM). While the algorithm was polynomial, it was a continuous algorithm, and it was not practical, so the search continued for a purely combinatorial strongly polynomial time algorithm. Queyranne [Que98] then proved that an algorithm [NI92] worked for this problem when the set function also satisfies a symmetry condition (i.e., $\forall X \subseteq V, f(X) = f(V \setminus X)$), which only requires $O(n^3)$ time. The result finally came in around year 2000 using two mostly independent methods [IFF00; Sch00]. These algorithms, however, also were impractical in that while they are polynomial time, they have unrealistically high polynomial degree (i.e., $\tilde{O}(V^7 * \gamma + V^8)$ for [Sch00] and $\tilde{O}(V^7 * \gamma)$ for [IFF00]). This led to additional work on combinatorial algorithms for SFM leading to algorithms that could perform SFM in time $\tilde{O}(V^5\gamma + V^6)$ in [IO09]. Two practical algorithms for SFM include the Fujishige-Wolfe procedure [Fuj05; Wol76][7] as well as the Frank-Wolfe procedure, each of which minimize the 2-norm on a polyhedron B_f associated with the submodular function f and which is defined below (it should also be noted that the Frank-Wolfe algorithm can also be used to minimize the convex extension of the function, something that is relatively easy to compute via the Lovász extension [Lov83]). More recent work on SFM are also based continuous relaxations of the problem in some form or another, leading algorithms with strongly polynomial running time [LSW15] of $O(n^3 \log^2 n)$ for which it was possible to drop the log factors leading to a complexity of $O(n^3)$ in [Jia21], weakly-polynomial running time [LSW15] of $\tilde{O}(n^2 \log M)$ (where $M >= \max_{S \subseteq V} |f(S)|$), pseudo-polynomial running time [ALS20; Cha+17] of $\tilde{O}(nM^2)$, and a ϵ-approximate minimization with a linear running time [ALS20] of $\tilde{O}(n/\epsilon^2)$. There have been other efforts to utilize parallelism to further improve SFM [BS20].

7. This is the same Wolfe as the Wolfe in Frank-Wolfe but not the same algorithm.

6.11.5 Applications of Submodularity in Machine Learning and AI

Submodularity arises naturally in applications in machine learning and artificial intelligence, but its utility has still not yet been as widely recognized and exploited as other techniques. For example, while information theoretic concepts like entropy and mutual information are extremely widely used in machine learning (e.g., the cross entropy loss for classification is ubiquitous), the submodularity property of entropy is not nearly as widely explored.

Still, the last several decades, submodularity has been increasingly studied and utilized in the context of machine learning. The below we begin to provide only a brief survey of some of the major sub-areas within machine learning that have been touched by submodularity. The list is not meant to be exhaustive, or even extensive. It is hoped that the below should, at least, offer a reasonable introduction into how submodularity has been and can continue to be useful in machine learning and artificial intelligence.

6.11.6 Sketching, CoreSets, Distillation, and Data Subset & Feature Selection

A summary is a concise representation of a body of data that can be used as an effective and efficient substitute for that data. There are many types of summary and some are extremely simple. For example, the mean or median of a list of numbers summarizes some property (the central tendency) of that list. A random subset is also a form of summary.

Any given summary, however, is not guaranteed to do a good job serving all purposes. Moreover, a summary usually involves at least some degree of approximation and fidelity loss relative to the original, and different summaries are faithful to the original in different ways and for different tasks. For these and other reasons, the field of summarization is rich and diverse, and summarization procedures are often very specialized.

Several distinct names for summarization have been used over the past few decades, including "sketches", "coresets", (in the field of natural language processing) "summaries", and "distillation."

Sketches [Cor17; CY20; Cor+12], arose in the field of computer science and was based on the acknowledgment that data is often too large to fit in memory and too large for an algorithm to run on a given machine, something enabled by a much smaller but still representative, and provably approximate, representation of the data.

Coresets are similar to sketches and there are some properties that are more often associated with coresets than with sketches, but sometimes the distinction is a bit vague. The notion of a coreset [BHPI02; AHP+05; BC08] comes from the field of computational geometry where one is interested in solving certain geometric problems based on a set of points in $\mathbb{R}^d$. For any geometric problem and a set of points, a coreset problem typically involves finding the smallest weighted subset of points so that when an algorithm is run on the weighted subset, it produces approximately the same answer as when it is run on the original large dataset. For example, given a set of points, one might wish to find the diameter of set, or the radius of the smallest enclosing sphere, or finding the narrowest annulus (ring) containing the points, or a subset of points whose k-center clustering is approximately the same as the k-center clustering of the whole [BHPI02].

Document summarization became one of the most important problems in natural language processing (NLP) in the 1990s although the idea of computing a summary of a text goes back much further to the 1950s [Luh58; Edm69], also and coincidentally around the same time that the CliffsNotes [Wik21] organization began. There are two main forms of document summarization [YWX17]. With *extractive*

summarization [NM12], a set of sentences (or phrases) are extracted from the documents needing to be summarized, and the resulting subset of sentences, perhaps appropriately ordered, comprises the summary.

With abstractive summarization [LN19], on the other hand, the goal is to produce an "abstract" of the documents, where one is not constrained to have any of the sentences in the abstract correspond to any of the sentences in the original documents. With abstractive summarization, therefore, the goal is to synthesize a small set of new pseudo sentences that represent the original documents. CliffsNotes, for example, are abstractive summaries of the literature being represented.

Another form of summarization that has more recently become popular in the machine learning community is *data distillation* [SG06b; Wan+20c; Suc+20; BYH20; NCL20; SS21; Ngu+21] or equivalently *dataset condensation* [ZMB21; ZB21]. With data distillation[8], the goal is to produce a small set of synthetic pseudo-samples that can be used, for example, to train a model. The key here is that in the reduced dataset, the samples are not compelled to be the same as, or a subset of, the original dataset.

All of the above should be contrasted with data *compression*, which in some sense is the most extreme data reduction method. With compression, either lossless or lossy, one is no longer under any obligation that the reduced form of the data need to be used or recognizable by any algorithm or entity other than the decoder, or uncompression, algorithm.

6.11.6.1 Summarization Algorithm Design Choices

It is the author's contention that the notions of summarization, coresets, sketching, and distillation are certainly analogous and quite possibly synonymous, and they are all different from compression. The different names for summarization are simply different nomenclatures for the same language game. What matters is not what you call it but the choices one makes when designing a procedure for summarization. And indeed, there are many choices.

Submodularity offers essentially an infinite number ways to perform data sketching and coresets. When we view the submodular function as an information function (as we discuss in Section 6.11.7), where $f(X)$ is the information contained in set X and $f(V)$ is the maximum available information, finding the small X that maximizes $f(X)$ (i.e., $X^* \in \text{argmax}\{f(X) : |X| \leq k\}$), is a form of coreset computation that is parameterized by the function f which has 2^n parameters since f lives in a 2^n-dimensional cone. Performing this maximization will then minimize the residual information $f(V \setminus X|X)$ about anything not present the summary $V \setminus X$ since $f(V) = f(X \cup V \setminus X) = f(V \setminus X|X) + f(X)$ so maximizing $f(X)$ will minimize $f(V \setminus X|X)$. For every f, moreover, the same algorithm (e.g., the greedy algorithm) can be used to produce the summarization, and in every case there is an approximation guarantee relative to the current f, as mentioned in earlier sections, as long as f stays submodular. Hence, submodularity provides a universal framework for summarization, coresets, and sketches to the extent that the space of submodular functions itself is sufficiently diverse and spans over different coreset problems.

Overall, the corset or sketching problem, when using submodular functions, therefore becomes a problem of "submodular design." That is, how do we construct submodular function that, for a particular problem, acts as a good coreset producer when the function is maximized. There are three general approaches to produce an f that works well as a summarization objective: (1) a pragmatic

8. Data distillation is distinct from the notion of *knowledge distillation* [HVD14; BC14; BCNM06] or *model distillation*, where the "knowledge" contained in a large model is distilled or reduced down into a different smaller model.

approach where the function is constructed by hand and heuristics, (2) a learning approach where all or part of the submodular function is inferred from an optimization procedure, and (3) a mathematical approach where a given submodular function when optimized offers of a coreset property.

When the primary goal is a practical and scalable algorithm that can produce an extractive summary that works well on a variety of different data types, and if one is comfortable with heuristics that work well in practice, a good option is to specify a submodular function by hand. For example, given a similarity matrix, it is easy to instantiate a facility location function and maximize it to produce a summary. If there are multiple similarity matrices, one can construct multiple facility location functions and maximize their convex combination. Such an approach is viable and practical and has been used successfully many times in the past for producing good summaries. One of the earliest examples of this the algorithm presented in [KKT03] that shows how a submodular model can be used to select the most influential nodes in a social network. Perhaps the earliest example of this approach used for data subset selection for machine learning is [LB09] which utilizes a submodular facility location function based on Fisher kernels (gradients w.r.t. parameters of log probabilities) and applies it to unsupervised speech selection to reduce transcription costs. Other examples of this approach includes: [LB10a; LB11] which developed submodular functions for query-focused document summarization; [KB14b] which computes a subset of training data in the context of transductive learning in a statistical machine translation system; [LB10b; Wei+13; Wei+14] which develops submodular functions for speech data subset selection (the former, incidentally, is the first use of a deep submodular function and the latter does this in an unsupervised label-free fashion); [SS18a] which is a form of robust submodularity for producing coresets for training CNNs; [Kau+19] which uses a facility location to facilitate diversity selection in active learning; [Bai+15; CTN17] which develops a mixture of submodular functions for document summarization where the mixture coefficients are also included in the hyperparameter set; [Xu+15] uses a symmetrized submodular function for the purposes of video summarization.

The learnability and identifiability of submodular functions has received a good amount of study from a theoretical perspective. Starting with the strictest learning settings, the problem looks pretty dire. For example, [SF08; Goe+09] shows that if one is restricted to making a polynomial number of queries (i.e., training pairs of the form $(S, f(S))$) of a monotone submodular function, then it is not possible to approximate f with a multiplicative approximation factor better than $\tilde{\Omega}(\sqrt{n})$. In [BH11], goodness is judged multiplicatively, meaning for a set $A \subseteq V$ we wish that $\tilde{f}(A) \leq f(A) \leq g(n)f(A)$ for some function $g(n)$, and this is typically a probabilistic condition (i.e., measured by distribution, or $\tilde{f}(A) \leq f(A) \leq g(n)f(A)$, should happen on a fraction at least $1-\beta$ of the points). Alternatively, goodness may also be measured by an additive approximation error, say by a norm. I.e., defining $\text{err}_p(f, \tilde{f}) = \|f - \tilde{f}\|_p = (E_{A \sim \mathbf{Pr}}[|f(A) - \tilde{f}(A)|^p])^{1/p}$, we may wish $\text{err}_p(f, \tilde{f}) < \epsilon$ for $p = 1$ or $p = 2$. In the PAC (probably approximately correct) model, we probably ($\delta > 0$) approximately ($\epsilon > 0$ or $g(n) > 1$) learn ($\beta = 0$) with a sample or algorithmic complexity that depends on δ and $g(n)$. In the PMAC (probably mostly approximately correct) model [BH11], we also "mostly" $\beta > 0$ learn. In some cases we wish to learn the best submodular approximation to a non-submodular function. In other cases, we are allowed to deviate from submodularity as long as the error is small. Learning special cases includes coverage functions [FK14; FK13a], and low-degree polynomials [FV15], curvature limited functions [IJB13], functions with a limited "goal" [DHK14; Bac+18], functions that are Fourier sparse [Wen+20a], or that are of a family called "juntas" [FV16], or that come from families other than submodular [DFF21], and still others [BRS17; FKV14; FKV17; FKV20; FKV13; YZ19]. Other results include that one can not minimize a submodular function by learning it first from

samples [BS17]. The essential strategy of learning is to attempt to construct a submodular function approximation $\hat{f}$ from an underlying submodular function f querying the latter only a small number of times. The overall gist of these results is that it is hard to learn everywhere and accurately.

In the machine learning community, learning can be performed extremely efficiently in practice, although there are not the types of guarantees as one finds above. For example, given a mixture of submodular components of the form $f(A) = \sum_i \alpha_i f_i(A)$, if each f_i is considered fixed, then the learning occurs only over the mixture coefficients α_i. This can be solved as a linear regression problem where the optimal coefficients can be computed in a linear regression setting. Alternatively, such functions can be learnt in a max-margin setting where the goal is primarily to adjust α_i to ensure that $f(A)$ is large on certain subsets [SSJ12; LB12; Tsc+14]. Even here there are practical challenges, however, since it is in general hard in practice to obtain a training set of pairs $\{(S_i, F(S_i))\}_i$. Alternatively, one also "learn" a submodular function in a reinforcement learning setting [CKK17] by optimizing the implicit function directly from gain vectors queried from an environment. In general, such practical learning algorithms have been used for image summarization [Tsc+14], document summarization [LB12], and video summarization [GGG15; Vas+17a; Gon+14; SGS16; SLG17]. While none of these learning approaches claim to approximate some true underlying submodular function, in practice, they do perform better than the by-hand crafting of a submodular functions mentioned above.

By a submodularity based coreset, we mean one where the direct optimization of a submodular function offers a theoretical guarantee for some specific problem. This is distinct from above where the submodular function is used as a surrogate heuristic objective function and for which, even if the submodular function is learnt, optimizing it is only a heuristic for the original problem. In some limited cases, it can be shown that the function we wish to approximate is already submodular, e.g., in the case of certain naive Bayes and k-NN classifiers [WIB15] where the training accuracy, as a function of the training data subset, can be shown to be submodular. Hence, maximizing this function offers the same guarantee on the training accuracy as it does on the submodular function. Unfortunately, the accuracy function for many models are not submodular, although they do have a difference of submodular [NB05; IB12] decomposition.

In other cases, it can be shown that certain desirable coreset objectives are inherently submodular. For example, in [MBL20], it is shown that the normed difference between the overall gradient (from summing over all samples in the training data) and an approximate gradient (from summing over only samples in a summary) can be upper bounded with a supermodular function that, when converted to a submodular facility location function and maximized, will select a set that reduces this difference, and will lead to similar convergence rates to an approximate optimum solution in the convex case. A similar example of this in a DPP context is shown in [TBA19]. In other cases, subsets of the training data and training occur simultaneously using a continuous-discrete optimization framework, where the goal is to minimize the loss on diverse and challenging samples measured by a submodular objective [ZB18]. In still other cases, bi-level objectives related to but not guaranteed to be submodular can be formed where a set is selected from a training set with the deliberate purpose of doing well on a validation set [Kil+20; BMK20].

The methods above have focused on reducing the number of samples in a training data. Considering the transpose of a design matrix, however, all of the above methods can be used for reducing the features of a machine learning procedure as well. Specifically, any of the extractive summarization, subset selection, or coreset methods can be seen as feature selection while any of the abstract summarization, sketching, or distillation approaches can be seen as dimensionality reduction.

6.11.7 Combinatorial Information Functions

The entropy function over a set of random variables $X_1, X_2, \ldots, X_n$ is defined as $H(X_1, X_2, \ldots, X_n) = -\sum_{x_1,x_2,\ldots,x_n} p(x_1, \ldots, x_n) \log p(x_1, \ldots, x_n)$. From this we can define three set-argument conditional mutual information functions as $I_H(A; B|C) = I(X_A; X_B|X_C)$ where the latter is the mutual information between variables indexed by A and B given variables indexed by C. This mutual information expresses the residual information between X_A and X_B that is not explained by their common information with X_C.

As mentioned above, we may view any polymatroid function as a type of information function over subsets of V. That is, $f(A)$ is the information in set A — to the extent that this is true, this property justifies f's use as a summarization objective as mentioned above. The reason f may be viewed as an information function stems from f being normalized, f's non-negativity, f's monotonicity, and the property that further conditioning reduces valuation (i.e., $f(A|B) \geq f(A|B, C)$ which is identical to the submodularity property). These properties were outlines as being essential to the entropy function in Shannon's original work [Sha48] but are true of any polymatroid function as well. Hence, given any polymatroid function f, is possible to define a combinatorial mutual information function [Iye+21] in a similar way. Specifically, we can define the combinatorial (submodular) conditional mutual information (CCMI) as $I_f(A; B|C) = f(A + C) + f(B + C) - f(C) - f(A + B + C)$, which has been known as the connectivity function [Cun83] amongst other names. If f is the entropy function then this yields the standard entropic mutual information but here the mutual information can be defined for any submodular information measure f. For an arbitrary polymatroid f, therefore, $I_f(A; B|C)$ can be seen as an A, B set-pair similarity score that ignores, neglects, or discounts any common similarity between the A, B pair that is due to C.

Historical use of a special case of CCMI, i.e., $I_f(A; B)$ where $C = \emptyset$, occurred in a number of circumstances. For example, in [GKS05] the function $g(A) = I_f(A; V \setminus A)$ (which, incidentally is both symmetric ($g(A) = g(V \setminus A)$ for all A) and submodular was optimized using the greedy procedure which has a guarantee as long as $g(A)$ is monotone up $2k$ elements whenever one wishes for a summary of size k. This was done for f being the entropy function, but it can be used for any polymatroid function. In similar work where f is Shannon entropy, [KG05] demonstrated that $g_C(A) = I_f(A; C)$ for a fixed set C is not submodular in A but if it is the case that the elements of V are independent given C then submodularity is preserved. This can be seen quickly easily by the consequence of the assumption which states that $I_f(A; C) = f(A) - f(A|C) = f(A) - \sum_{a \in A} f(a|C)$ where the second equality is due to the conditional independence property. In this case, I_f is the difference between a submodular and a modular function which preserves submodularity for any polymatroid f.

On the other hand, it would be useful for $g_{B,C}(A) = I_f(A; B|C)$, where B and C are fixed, to be possible to optimize in terms of A. One can view this function as one that, when it is maximized, chooses A to be similar to B in a way that neglects or discounts any common similarity that A and B have with C. One option to optimize this function to utilize difference of submodular [NB05; IB12] optimization as mentioned earlier. A more recent result shows that in some cases $g_{B,C}(A)$ is still submodular in A. Define the second order partial derivative of a submodular function f as follows $f(i, j|S) \triangleq f(j|S + i) - f(j|S)$. Then if it is the case that $f(i, j|S)$ is monotone non-decreasing in S for $S \subseteq V \setminus \{i, j\}$ then $I_f(A; B|C)$ is submodular in A for fixed B and C. It may be thought that only esoteric functions have this property but in fact [Iye+21] shows that this is true for a number of widely used submodular functions in practice, including the facility location function which results

in the form $I_f(A;B|C) = \sum_{v \in V} \max\Big(\min\big(\sum_{a \in A} \text{sim}(v,a), \max_{b \in B} \text{sim}(v,b)\big) - \max_{c \in C} \text{sim}(v,c), 0\Big)$. This function was used [Kot+22] to produce summaries A that were particularly relevant to a query given by B but that should neglect information in C that can be considered "private" information to avoid.

6.11.8 Clustering, Data Partitioning, and Parallel Machine Learning

There are an almost limited number of clustering algorithms and a plethora of reviews on their variants. Any given submodular function can also instantiate a clustering procedure as well, and there are several ways to do this. Here we offer only a brief outline of the approach. In the last section, we defined $I_f(A; V \setminus A)$ as the CCMI between A and everything but A. When we view this as a function of A, then $g(A) = I_f(A; V \setminus A)$ and $g(A)$ is a symmetric submodular function that can be minimized using Queyranne's algorithm [Que98; NI92]. Once this is done, the resulting A is such that it is least similar to $V \setminus A$, according to $I_f(A; V \setminus A)$ and hence forms a 2-clustering. This process can then be recursively applied where we form two new functions $g_A(B) = I_f(B; A \setminus B)$ for $B \subseteq A$ and $g_{V \setminus A}(B) = I_f(B; (V \setminus A) \setminus B)$ for $B \subseteq V \setminus A$. These are two symmetric submodular functions on different ground sets that also can be minimized using Queyranne's algorithm. This recursive bisection algorithm then repeats until the desired number of clusters is formed. Hence, the CCMI function can be used as a top-down recursive bisection clustering procedure and has been called Q-clustering [NJB05; NB06]. It should be noted that such forms of clustering often generalizes forming a multi-way cut in an undirected graph in which case the objective becomes the graph-cut function that, as we saw above, is also submodular. In some cases, the number of clusters need not be specified in advance [NKI10]. Another submodular approach to clustering can be found in [Wei+15b] where the goal is to minimize the maximum valued block in a partitioning which can lead to submodular load balancing or minimum makespan scheduling [HS88; LST90].

Yet another form of clustering can be seen via the simple cardinality constrained submodular maximization process itself which can be compared to a k-medoids process whenever the objective f is the facility location function. Hence, any such submodular function can be seen as a submodular-function-parameterized form of finding the k "centers" among a set of data items. There have been numerous applications of submodular clustering. For example, using these techniques it is possible to identify parcellations of the human brain [Sal+17a]. Other applications include partitioning data for more effective and accurate and lower variance distributed machine learning training [Wei+15a] and also for more ideal mini-batch construction for training deep neural networks [Wan+19b].

6.11.9 Active and Semi-Supervised Learning

Suppose we are given data set $\{x_i, y_i\}_{i \in V}$ consisting of $|V| = n$ samples of x, y pairs but where the labels are unknown. Samples are labeled one at a time or one mini-batch at a time, and after each labeling step t each remaining unlabeled sample is given a score $s_t(x_i)$ that indicates the potential benefit of acquiring a label for that sample. Examples include the entropy of the model's output distribution on x_i, or a margin based score consisting of the difference between the top and the second-from-the-top posterior probability. This produces a modular function on the unlabeled samples, $m_t(A) = \sum_{a \in A} s(x_a)$ where $A \subseteq V$. It is simple to use this modular function to produce a mini-batch active learning procedure where at each stage we form $A_t \in \text{argmax}_{A \subseteq U_t : |A| = k} m_t(A)$ where U_t is the set of labeled samples at stage t. Then A_t is a set of size k that gets labeled, we form

$U_t = U_t \setminus A_t$, update $s_t(a)$ for $a \in U_t$ and repeat. This is called **active learning** (Section 34.7).

The reason for using active learning with mini-batches of size greater than one is that it is often inefficient to ask for single label at a time. The problem with such a minibatch strategy, however, is that the set A_t can be redundant. The reason is that the uncertainty about every sample in A_t could be owing to the same underlying cause — even though the model is most uncertain about samples in A_t, once one sample in A_t is labeled, it may not be optimal to label the remaining samples in A_t due to this redundancy. Utilizing submodularity, therefore, can help reduce this redundancy. Suppose $f_t(A)$ is a submodular diversity model over samples at step t. At each stage, choosing the set of samples to label becomes $A_t \in \operatorname{argmax}_{A \subseteq U_t : |A|=k} m_t(A) + f_t(A)$ — A_t is selected based on a combination of both uncertainty (via $m_t(A)$) and diversity (via $f_t(A)$). This is precisely the submodular active learning approach taken in [WIB15; Kau+19].

Another quite different approach to a form of submodular "batch" active learning setting where a batch L of labeled samples are selected all at once and then used to label the rest of the unlabeled samples. This also allows the remaining unlabeled samples to be utilized in a semi-supervised framework [GB09; GB11]. In this setting, we start with a graph $G = (V, E)$ where the nodes V need to be given a binary $\{0,1\}$-valued label, $y \in \{0,1\}^V$. For any $A \subseteq V$ let $y_A \in \{0,1\}^A$ be the labels just for node set A. We also define $V(y) \subseteq V$ as $V(y) = \{v \in V : y_v = 1\}$. Hence $V(y)$ are the graph nodes labeled 1 by y and $V \setminus V(y)$ are the nodes labeled 0. Given submodular objective f, we form its symmetric CCMI variant $I_f(A) \triangleq I_f(A; V \setminus A)$ — note that $I_f(A)$ is always submodular in A. This allows $I_f(V(y))$ to determine the "smoothness" of a given candidate labeling y. For example, if I_f is the weighted graph cut function where each weight corresponds to an affinity between the corresponding two nodes, then $I_f(V(y))$ would be small if $V(y)$ (the 1-labeled nodes) do not have strong affinity with $V \setminus V(y)$ (the 0-labeled nodes). In general, however, I_f can be any symmetric submodular function. Let $L \subseteq V$ be any candidate set of nodes to be labeled, and define $\Psi(L) \triangleq \min_{T \subseteq (V \setminus L) : T \neq \emptyset} I_f(T)/|T|$. Then $\Psi(L)$ measures the "strength" of L in that if $\Psi(L)$ is small, an adversary can label nodes other than L without being too unsmooth according to I_f, while if $\Psi(L)$ is large, an adversary can do no such thing. Then [GB11] showed that given a node set L to be queried, and the corresponding correct labels y_L that are completed (in a semi-supervised fashion) according to the following $y' = \operatorname{argmin}_{\hat{y} \in \{0,1\}^V : \hat{y}_L = y_L} I_f(V(\hat{y}))$, then this results in the following bound on the true labeling $\|y - y'\|^2 \leq 2 I_f(V(y))/\Psi(L)$ suggesting that we can find a good set to query by maximizing L in $\Psi(L)$, and this holds for any submodular function. Of course it is necessary to find an underlying submodular function f that fits a given problem, and this is discussed in Section 6.11.6.

6.11.10 Probabilistic Modeling

Graphical models are often used to describe factorization requirements on families of probability distributions. Factorization is not the only way, however, to describe restrictions on such families. In a graphical model, graphs describe only which random variable may directly interact with other random variable. An entirely different strategy for producing families of often-tractable probabilistic models can be produced without requiring any factorization property at all. Considering an energy function $E(x)$ where $p(x) \propto \exp(E(x))$, factorizations correspond to there being cliques in the graph such that the graph's tree-width often is limited. On the other hand, finding $\max_x p(x)$ is the same as finding $\min_x E(x)$, something that can be done if $E(x) = f(V(x))$ is a submodular function (using the earlier used notation $V(x)$ to map from binary vectors to subsets of V). Even a submodular

function as simple as $f(A) = \sqrt{|A|} - m(A)$ where m is modular has tree-width of $n-1$, and this leads to an energy function $E(x)$ that allows $\max_x p(x)$ to be solved in polynomial time using submodular function minimization (see Section 6.11.4.3). Such restrictions to $E(x)$ therefore are not of the form *amongst the random variables, who is allowed to directly interact with whom*, but rather *amongst the random variables, what is the manner that they interact.* Such potential function restrictions can also combine with direct interaction restrictions as well and this has been widely used in computer vision, leading to cases where graph-cut and graph-cut like "move making" algorithms (such as *alpha-beta* swap and *alpha*-expansion algorithms) used in attractive models [BVZ99; BK01; BVZ01; SWW08]. In fact, the culmination of these efforts [KZ02] lead to a rediscovery of the submodularity (or the "regular" property) as being the essential ingredient for when Markov random fields can be solved using graph cut minimization, which is a special case of submodular function minimization.

The above model can be seen as log-supermodular since $\log p(x) = -E(x) + \log 1/Z$ is a supermodular function. These are all distributions that put high probability on configurations that yield small valuation by a submodular function. Therefore, these distributions have high probability when x consists of a homogeneous and for this reason they are useful for computer vision segmentation problems (e.g., in a segment of an image, the nearby pixels should roughly be homogeneous as that is often what defines an object). The DPPs we saw above, however, are an example of a log-submodular probability distribution since $f(X) = \log\det(\mathbf{M}_X)$ is submodular. These models have high probability for diverse sets.

More generally, $E(x)$ being either a submodular or supermodular function can produce log-submodular or log-supermodular distributions, covering both cases above where the partition function takes the form $Z = \sum_{A \subseteq V} \exp(f(A))$ for objective f. Moreover, we often wish to perform tasks much more than just finding the most probable random variable assignments. This includes marginalization, computing the partition function, constrained maximization, and so on. Unfortunately, many of these more general probabilistic inference problems do not have polynomial time solutions even though the objectives are submodular or supermodular. On the other hand, such structure has opened the doors to an assortment of new probabilistic inference procedures that exploit this structure [DK14; DK15a; DTK16; ZDK15; DJK18]. Most of these methods were of the variational sort and offered bounds on the partition function Z, sometimes making use of the fact that submodular functions have easily computable semi-gradients [IB15; Fuj05] which are modular upper and lower bounds on a submodular or supermodular function that are tight at one or more subsets. Given a submodular (or supermodular) function f and a set A, it is possible to easily construct (in linear time) a modular function upperbound $m^A : 2^V \to \mathbb{R}$ and a modular function lower bound $m_A : 2^V \to \mathbb{R}$ having the properties that $m_A(X) \leq f(X) \leq m^A(X)$ for all $X \subseteq V$ and that is tight at $X = A$ meaning $m_A(A) = f(A) = m^A(A)$ [IB15]. For any modular function m, the probability function for a characteristic vector $x = \mathbf{1}_A$ becomes $p(\mathbf{1}_A) = 1/Z \exp(E(\mathbf{1}_A)) = \prod_{a \in A} \sigma(m(a)) \prod_{a \notin A} \sigma(-m(a))$ where σ is the logistic function. Thus, a modular approximation of a submodular function is like a mean-field approximation of the distribution, and makes the assumption that all random variables are independent. Such an approximation can then be used to compute quantities such as upper and lower bounds on the partition function, and much else.

6.11.11 Structured Norms and Loss Functions

Convex norms are used ubiquitously in machine learning, often as complexity penalizing regularizers (e.g., the ubiquitous p-norms for $p \geq 1$) and also sometimes as losses (e.g., squared error). Identifying

new useful structured and possibly learnable sparse norms is an interesting and useful endeavor, and submodularity can help here as well. Firstly, recall the ℓ_0 or counting norm $\|x\|_0$ simply counts the number of nonzero entries in x. When we wish for a sparse solution, we may wish to regularize using $\|x\|_0$ but it both leads to an intractable combinatorial optimization problem and it leads to an object that is not differentiable. The usual approach is to find the closest convex relaxation of this norm and that is the one norm or $\|x\|_1$. This is convex in x and has a sub-gradient structure and hence can be combined with a loss function to produce an optimizable machine learning objective, for example the lasso. On the other hand $\|x\|_1$ has no structure, as each element of x is penalized based on its absolute value irrespective of the state of any of the other elements. There have thus been efforts to develop group norms that penalize groups or subsets of elements of x together, such as group lasso [HTW15].

It turns out that there is a way to utilize a submodular function as the regularizer. Penalizing x via $\|x\|_0$ is identical to penalizing it via $|V(x)|$ and note that $m(A) = |A|$ is a modular function. Instead, we could penalize x via $f(V(x))$ for a submodular function f. Here, any element of x being non-zero would allow for a diminishing penalty of other elements of x being zero all according to the submodular function, and such cooperative penalties can be obtained via a submodular parameterization. Like when using the zero-norm $\|x\|_0$, this leads to the same combinatorial problem due to continuous optimization of x with a penalty term of the form $f(V(x))$. To address this, we can use the Lovász extension $\breve{f}(x)$ on a vector x. This function is convex but it is not a norm, but if we consider the construct defined as $\|x\|_f = \breve{f}(|x|)$, it can be shown that this satisfies all the properties of a norm for all non-trivial submodular functions [PG98; Bac+13] (i.e., those normalized submodular functions for which $f(v) > 0$ for all v). In fact, the group lasso mentioned above is a special case for a particularly simple feature-based submodular function (a sum of min-truncated cardinality functions). But in principle, the same submodular design strategies mentioned in Section 6.11.6 can be used to produce a submodular function to instantiate an appropriate convex structured norm for a given machine learning problem.

6.11.12 Conclusions

We have only barely touched the surface of submodularity and how it applies to and can benefit machine learning. For more details, see [Bil22] and the many references contained therein. Considering once again the innocuous looking submodular inequality, then very much like the definition of convexity, we observe something that belies much of its complexity while opening the gates to wide and worthwhile avenues for machine learning exploration.

Part II

Inference

7 Inference algorithms: an overview

7.1 Introduction

In the probabilistic approach to machine learning, all unknown quantities — be they predictions about the future, hidden states of a system, or parameters of a model — are treated as random variables, and endowed with probability distributions. The process of **inference** corresponds to computing the posterior distribution over these quantities, conditioning on whatever data is available.

Let $\boldsymbol{h}$ represent the unknown variables, and $\mathcal{D}$ represent the known variables. Given a likelihood $p(\mathcal{D}|\boldsymbol{h})$ and a prior $p(\boldsymbol{h})$, we can compute the posterior $p(\boldsymbol{h}|\mathcal{D})$ using Bayes' rule:

$$p(\boldsymbol{h}|\mathcal{D}) = \frac{p(\boldsymbol{h})p(\mathcal{D}|\boldsymbol{h})}{p(\mathcal{D})} \tag{7.1}$$

The main computational bottleneck is computing the normalization constant in the denominator, which requires solving the following high dimensional integral:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{h})p(\boldsymbol{h})d\boldsymbol{h} \tag{7.2}$$

This is needed to convert the unnormalized joint probability of some parameter value, $p(\boldsymbol{h}, \mathcal{D})$, to a normalized probability, $p(\boldsymbol{h}|\mathcal{D})$, which takes into account all the other plausible values that $\boldsymbol{h}$ could have. Similarly, computing posterior marginals also requires computing integrals:

$$p(h_i|\mathcal{D}) = \int p(h_i, \boldsymbol{h}_{-i}|\mathcal{D})d\boldsymbol{h}_{-i} \tag{7.3}$$

Thus integration is at the heart of Bayesian inference, whereas differentiation is at the heart of optimization.

In this chapter, we give a high level summary of algorithmic techniques for computing (approximate) posteriors. We will give more details in the following chapters. Note that most of these methods are independent of the specific model. This allows problem solvers to focus on creating the best model possible for the task, and then relying on some inference engine to do the rest of the work — this latter process is sometimes called "**turning the Bayesian crank**". For more details, see e.g., [Gel+14a; MKL21; MFR20].

7.2 Common inference patterns

There are kinds of posterior we may want to compute, but we can identify 3 main patterns, as we discuss below. These give rise to different types of inference algorithm, as we will see in later chapters.

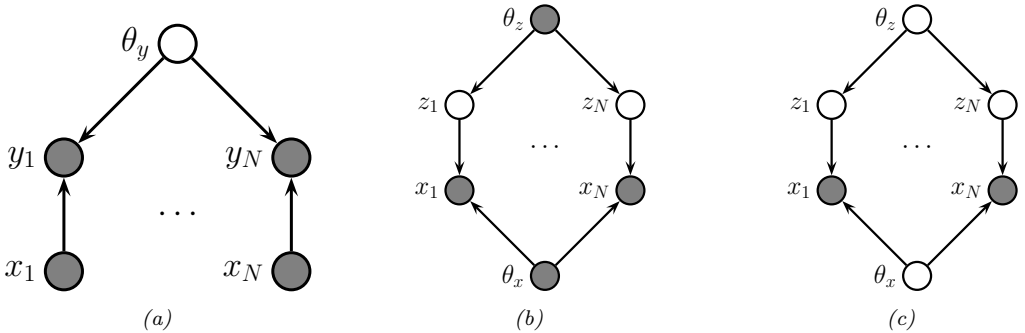


Figure 7.1: Graphical models with (a) Global hidden variables for representing the Bayesian discriminative model $p(\boldsymbol{y}_{1:N}, \boldsymbol{\theta}_y|\boldsymbol{x}_{1:N}) = p(\boldsymbol{\theta}_y)\prod_{n=1}^N p(\boldsymbol{y}_n|\boldsymbol{x}_n;\boldsymbol{\theta}_y)$; (b) Local hidden variables for representing the generative model $p(\boldsymbol{x}_{1:N}, \boldsymbol{z}_{1:N}|\boldsymbol{\theta}) = \prod_{n=1}^N p(\boldsymbol{z}_n|\boldsymbol{\theta}_z)p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\theta}_x)$; (c) Local and global hidden variables for representing the Bayesian generative model $p(\boldsymbol{x}_{1:N}, \boldsymbol{z}_{1:N}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}_z)p(\boldsymbol{\theta}_x)\prod_{n=1}^N p(\boldsymbol{z}_n|\boldsymbol{\theta}_z)p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\theta}_x)$. Shaded nodes are assumed to be known (observed), unshaded nodes are hidden.

7.2.1 Global latents

The first pattern arises when we need to perform inference in models which have **global latent variables**, such as parameters of a model $\boldsymbol{\theta}$, which are shared across all N observed training cases. This is shown in Figure 7.1a, and corresponds to the usual setting for supervised or discriminative learning, where the joint distribution has the form

$$p(\boldsymbol{y}_{1:N}, \boldsymbol{\theta}|\boldsymbol{x}_{1:N}) = p(\boldsymbol{\theta})\left[\prod_{n=1}^N p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta})\right] \tag{7.4}$$

The goal is to compute the posterior $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:N}, \boldsymbol{y}_{1:N})$. Most of the Bayesian supervised learning models discussed in Part III follow this pattern.

7.2.2 Local latents

The second pattern arises when we need to perform inference in models which have **local latent variables**, such as hidden states $\boldsymbol{z}_{1:N}$; we assume the model parameters $\boldsymbol{\theta}$ are known. This is shown in Figure 7.1b. Now the joint distribution has the form

$$p(\boldsymbol{x}_{1:N}, \boldsymbol{z}_{1:N}|\boldsymbol{\theta}) = \left[\prod_{n=1}^N p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\theta}_x)p(\boldsymbol{z}_n|\boldsymbol{\theta}_z)\right] \tag{7.5}$$

The goal is to compute $p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta})$ for each n. This is the setting we consider for most of the PGM inference methods in Chapter 9.

If the parameters are not known (which is the case for most latent variable models, such as mixture models), we may choose to estimate them by some method (e.g., maximum likelihood), and then plugin this point estimate. The advantage of this approach is that, conditional on $\boldsymbol{\theta}$, all the latent variables are conditionally independent, so we can perform inference in parallel across the data. This

lets us use methods such as expectation maximization (Section 6.6.3), in which we infer $p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta}_t)$ in the E step for all n simultaneously, and then update $\boldsymbol{\theta}_t$ in the M step. If the inference of $\boldsymbol{z}_n$ cannot be done exactly, we can use variational inference, a combination known as variational EM (Section 6.6.6.1).

Alternatively, we can use a minibatch approximation to the likelihood, marginalizing out $\boldsymbol{z}_n$ for each example in the minibatch to get

$$\log p(\mathcal{D}_t|\boldsymbol{\theta}_t) = \sum_{n \in \mathcal{D}_t} \log \left[\sum_{\boldsymbol{z}_n} p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\theta}_t) \right] \tag{7.6}$$

where $\mathcal{D}_t$ is the minibatch at step t. If the marginalization cannot be done exactly, we can use variational inference, a combination known as stochastic variational inference or SVI (Section 10.3.1). We can also learn an inference network $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}; \boldsymbol{\theta})$ to perform the inference for us, rather than running an inference engine for each example n in each batch t; the cost of learning $\boldsymbol{\phi}$ can be amortized across the batches. This is called amortized SVI, and is commonly used to train deep latent variable models such as VAEs (Section 21.2).

7.2.3 Global and local latents

The third pattern arises when we need to perform inference in models which have **local and global latent variables**. This is shown in Figure 7.1c, and corresponds to the following joint distribution:

$$p(\boldsymbol{x}_{1:N}, \boldsymbol{z}_{1:N}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}_x)p(\boldsymbol{\theta}_z) \left[\prod_{n=1}^{N} p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\theta}_x)p(\boldsymbol{z}_n|\boldsymbol{\theta}_z) \right] \tag{7.7}$$

This is essentially a Bayesian version of the latent variable model in Figure 7.1b, where now we model uncertainty in both the local variables $\boldsymbol{z}_n$ and the shared global variables $\boldsymbol{\theta}$. This approach is less common in the ML community, since it is often assumed that the uncertainty in the parameters $\boldsymbol{\theta}$ is negligible compared to the uncertainty in the local variables $\boldsymbol{z}_n$. The reason for this is that the parameters are "informed" by all N data cases, whereas each local latent $\boldsymbol{z}_n$ is only informed by a single data point, namely $\boldsymbol{x}_n$. Nevertheless, there are advantages to being "fully Bayesian", and modeling uncertainty in both local and global variables. We will see some examples of this later in the book.

7.3 Exact inference algorithms

In some cases, we can perform example posterior inference in a tractable manner. In particular, if the prior is **conjugate** to the likelihood, the posterior will be analytically tractable. In general, this will be the case when the prior and likelihood are from the same exponential family (Section 2.3). In particular, if the unknown variables are represented by $\boldsymbol{\theta}$, then we assume

$$p(\boldsymbol{\theta}) \propto \exp(\boldsymbol{\lambda}_0^\mathsf{T} \mathcal{T}(\boldsymbol{\theta})) \tag{7.8}$$

$$p(\boldsymbol{y}_i|\boldsymbol{\theta}) \propto \exp(\tilde{\boldsymbol{\lambda}}_i(\boldsymbol{y}_i)^\mathsf{T} \mathcal{T}(\boldsymbol{\theta})) \tag{7.9}$$

where $\mathcal{T}(\boldsymbol{\theta})$ are the sufficient statistics, and $\boldsymbol{\lambda}$ are the natural parameters. We can then compute the posterior by just adding the natural parameters:

$$p(\boldsymbol{\theta}|\boldsymbol{y}_{1:N}) = \exp(\boldsymbol{\lambda}_*^\mathsf{T}\mathcal{T}(\boldsymbol{\theta})) \tag{7.10}$$

$$\boldsymbol{\lambda}_* = \boldsymbol{\lambda}_0 + \sum_{n=1}^{N} \tilde{\boldsymbol{\lambda}}_n(\boldsymbol{y}_n) \tag{7.11}$$

See Section 3.2 for details.

Another setting where we can compute the posterior exactly arises when the D unknown variables are all discrete, each with K states; in this case, the integral for the normalizing constant becomes a sum with K^D terms. In many cases, K^D will be too large to be tractable. However, if the distribution satisfies certain conditional independence properties, as expressed by a probabilistic graphical model (PGM), then we can write the joint as a product of local terms (see Chapter 4 and Chapter 4). This lets us use dynamic programming to make the computation tractable. See Chapter 9 for details.

7.4 Approximate inference algorithms

For most probability models, we will not be able to compute marginals or posteriors exactly, so we must resort to using **approximate inference**. There are many different algorithms, which trade off speed, accuracy, simplicity, and generality. We briefly discuss some of these algorithms below. We give more detail in the following chapters.

7.4.1 MAP estimation

The simplest approximate inference method is to compute the MAP estimate

$$\hat{\boldsymbol{\theta}} = \operatorname{argmax} p(\boldsymbol{\theta}|\mathcal{D}) = \operatorname{argmax} \log p(\boldsymbol{\theta}) + \log p(\mathcal{D}|\boldsymbol{\theta}) \tag{7.12}$$

and then to assume that the posterior puts 100% of its probability on this single value:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \tag{7.13}$$

The advantage of this approach is that we can compute the MAP estimate using a variety of optimization algorithms, which we discuss in Chapter 6. The disavantages of the MAP approximation are discussed in Section 3.1.5.

7.4.2 Grid approximation

If we want to capture uncertainty, we need to allow for the fact that $\boldsymbol{\theta}$ may have a range of possible values, each with non-zero probability. The simplest way to capture this property is to partition the space of possible values into a finite set of regions, call them $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_K$, each representing a region of parameter space of volume Δ centered on $\boldsymbol{\theta}_k$. This is called a **grid approximation**. The probability of being in each region is given by $p(\boldsymbol{\theta} \in \boldsymbol{r}_k|\mathcal{D}) \approx p_k\Delta$, where

$$p_k = \frac{\tilde{p}_k}{\sum_{k'=1}^{K} \tilde{p}_{k'}} \tag{7.14}$$

$$\tilde{p}_k = p(\mathcal{D}|\boldsymbol{\theta}_k)p(\boldsymbol{\theta}_k) \tag{7.15}$$

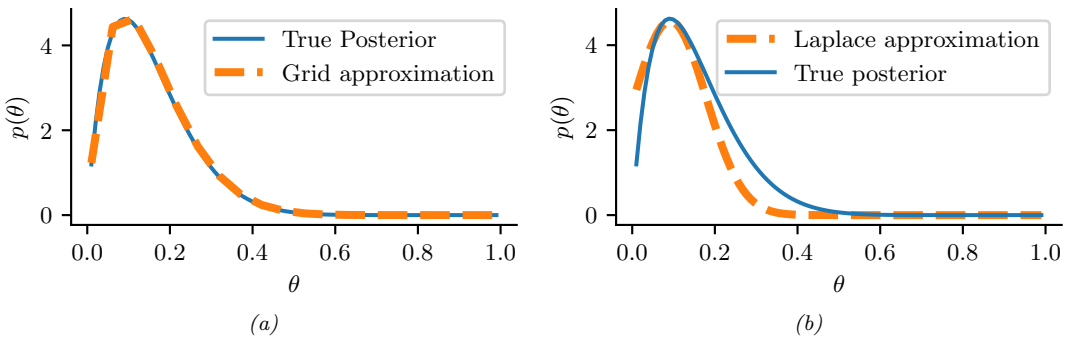


Figure 7.2: Approximating the posterior of a beta-Bernoulli model. (a) Grid approximation using 20 grid points. (b) Laplace approximation. Generated by laplace_approx_beta_binom.ipynb.

As K increases, we decrease the size of each grid cell. Thus the denominator is just a simple numerical approximation of the integral

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \approx \sum_{k=1}^{K} \Delta\tilde{p}_k \tag{7.16}$$

As a simple example, we will use the problem of approximating the posterior of a beta-Bernoulli model. Specifically, the goal is to approximate

$$p(\theta|\mathcal{D}) \propto \left[\prod_{n=1}^{N} \text{Ber}(y_n|\theta)\right] \text{Beta}(1,1) \tag{7.17}$$

where $\mathcal{D}$ consists of 10 heads and 1 tail (so the total number of observations is $N = 11$), with a uniform prior. Although we can compute this posterior exactly using the method discussed in Section 3.2.1, this serves as a useful pedagogical example since we can compare the approximation to the exact answer. Also, since the target distribution is just 1d, it is easy to visualize the results.

In Figure 7.2a, we illustrate the grid approximation applied to our 1d problem. We see that it is easily able to capture the skewed posterior (due to the use of an imbalanced sample of 10 heads and 1 tail). Unfortunately, this approach does not scale to problems in more than 2 or 3 dimensions, because the number of grid points grows exponentially with the number of dimensions.

7.4.3 Laplace (quadratic) approximation

In this section, we discuss a simple way to approximate the posterior using a multivariate Gaussian; this known as a **Laplace approximation** or **quadratic approximation** (see e.g., [TK86; RMC09]).

Suppose we write the posterior as follows:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z} e^{-\mathcal{E}(\boldsymbol{\theta})} \tag{7.18}$$

where $\mathcal{E}(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta}, \mathcal{D})$ is called an energy function, and $Z = p(\mathcal{D})$ is the normalization constant. Performing a Taylor series expansion around the mode $\hat{\boldsymbol{\theta}}$ (i.e., the lowest energy state) we get

$$\mathcal{E}(\boldsymbol{\theta}) \approx \mathcal{E}(\hat{\boldsymbol{\theta}}) + (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^\mathsf{T}\boldsymbol{g} + \frac{1}{2}(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^\mathsf{T}\mathbf{H}(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \tag{7.19}$$

where $\boldsymbol{g}$ is the gradient at the mode, and $\mathbf{H}$ is the Hessian. Since $\hat{\boldsymbol{\theta}}$ is the mode, the gradient term is zero. Hence

$$\hat{p}(\boldsymbol{\theta}, \mathcal{D}) = e^{-\mathcal{E}(\hat{\boldsymbol{\theta}})} \exp\left[-\frac{1}{2}(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^\mathsf{T}\mathbf{H}(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})\right] \tag{7.20}$$

$$\hat{p}(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z}\hat{p}(\boldsymbol{\theta}, \mathcal{D}) = \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}, \mathbf{H}^{-1}) \tag{7.21}$$

$$Z = e^{-\mathcal{E}(\hat{\boldsymbol{\theta}})}(2\pi)^{D/2}|\mathbf{H}|^{-\frac{1}{2}} \tag{7.22}$$

The last line follows from normalization constant of the multivariate Gaussian.

The Laplace approximation is easy to apply, since we can leverage existing optimization algorithms to compute the MAP estimate, and then we just have to compute the Hessian at the mode. (In high dimensional spaces, we can use a diagonal approximation.)

In Figure 7.2b, we illustrate this method applied to our 1d problem. Unfortunately we see that it is not a particularly good approximation. This is because the posterior is skewed, whereas a Gaussian is symmetric. In addition, the parameter of interest lies in the constrained interval $\theta \in [0, 1]$, whereas the Gaussian assumes an unconstrained space, $\boldsymbol{\theta} \in \mathbb{R}^D$. Fortunately, we can solve this latter problem by using a change of variable. For example, in this case we can apply the Laplace approximation to $\alpha = \text{logit}(\theta)$. This is a common trick to simplify the job of inference.

7.4.4 Variational inference

In Section 7.4.3, we discussed the Laplace approximation, which uses an optimization procedure to find the MAP estimate, and then approximates the curvature of the posterior at that point based on the Hessian. In this section, we discuss **variational inference** (**VI**), also called **variational Bayes** (**VB**). This is another optimization-based approach to posterior inference, but which has much more modeling flexibility (and thus can give a much more accurate approximation).

VI attempts to approximate an intractable probability distribution, such as $p(\boldsymbol{\theta}|\mathcal{D})$, with one that is tractable, $q(\boldsymbol{\theta})$, so as to minimize some discrepancy D between the distributions:

$$q^* = \underset{q \in \mathcal{Q}}{\text{argmin}}\, D(q, p) \tag{7.23}$$

where $\mathcal{Q}$ is some tractable family of distributions (e.g., fully factorized distributions). Rather than optimizing over functions q, we typically optimize over the parameters of the function q; we denote these **variational parameters** by $\boldsymbol{\psi}$.

It is common to use the KL divergence (Section 5.1) as the discrepancy measure, which is given by

$$D(q, p) = D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}|\boldsymbol{\psi}) \parallel p(\boldsymbol{\theta}|\mathcal{D})\right) = \int q(\boldsymbol{\theta}|\boldsymbol{\psi}) \log \frac{q(\boldsymbol{\theta}|\boldsymbol{\psi})}{p(\boldsymbol{\theta}|\mathcal{D})} d\boldsymbol{\theta} \tag{7.24}$$

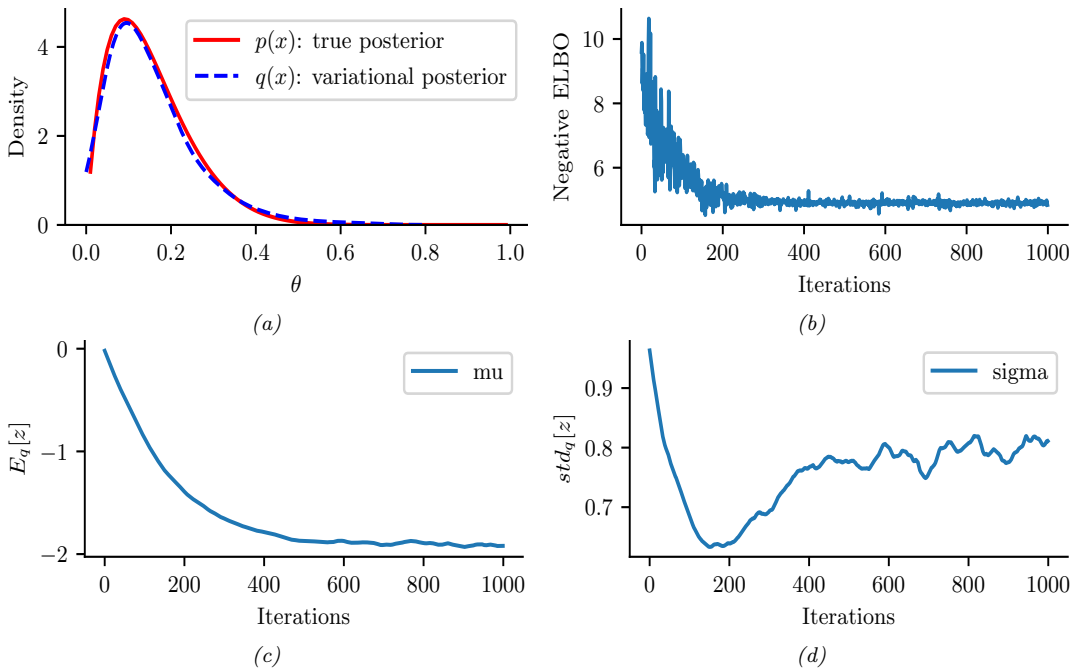


Figure 7.3: ADVI applied to the beta-Bernoulli model. (a) Approximate vs true posterior. (b) Negative ELBO over time. (c) Variational μ parameter over time. (d) Variational σ parameter over time. Generated by advi_beta_binom.ipynb.

where $p(\boldsymbol{\theta}|\mathcal{D}) = p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})/p(\mathcal{D})$. The inference problem then reduces to the following optimization problem:

$$\boldsymbol{\psi}^* = \underset{\boldsymbol{\psi}}{\operatorname{argmin}}\, D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}|\boldsymbol{\psi}) \parallel p(\boldsymbol{\theta}|\mathcal{D})\right) \tag{7.25}$$

$$= \underset{\boldsymbol{\psi}}{\operatorname{argmin}}\, \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\log q(\boldsymbol{\theta}|\boldsymbol{\psi}) - \log\left(\frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})}\right)\right] \tag{7.26}$$

$$= \underset{\boldsymbol{\psi}}{\operatorname{argmin}}\, \underbrace{\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[-\log p(\mathcal{D}|\boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) + \log q(\boldsymbol{\theta}|\boldsymbol{\psi})\right]}_{-Ł(\boldsymbol{\psi})} + \log p(\mathcal{D}) \tag{7.27}$$

Note that $\log p(\mathcal{D})$ is independent of $\boldsymbol{\psi}$, so we can ignore it when fitting the approximate posterior, and just focus on maximizing the term

$$Ł(\boldsymbol{\psi}) \triangleq \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\psi})\right] \tag{7.28}$$

Since we have $D_{\mathbb{KL}}\left(q \parallel p\right) \geq 0$, we have $Ł(\boldsymbol{\psi}) \leq \log p(\mathcal{D})$. The quantity $\log p(\mathcal{D})$, which is the log marginal likelihood, is also called the **evidence**. Hence $Ł(\boldsymbol{\psi})$ is known as the **evidence lower bound** or **ELBO**. By maximizing this bound, we are making the variational posterior closer to the true posterior. (See Section 10.1 for details.)

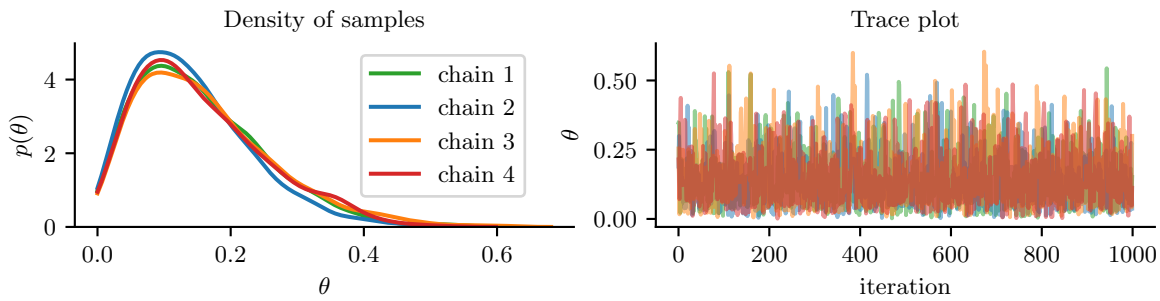


Figure 7.4: Approximating the posterior of a beta-Bernoulli model using MCMC. (a) Kernel density estimate derived from samples from 4 independent chains. (b) Trace plot of the chains as they generate posterior samples. Generated by hmc_beta_binom.ipynb.

We can chose any kind of approximate posterior that we like. For example, we may use a Gaussian, $q(\boldsymbol{\theta}|\boldsymbol{\psi}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. This is different from the Laplace approximation, since in VI, we optimize $\boldsymbol{\Sigma}$, rather than equating it to the Hessian. If $\boldsymbol{\Sigma}$ is diagonal, we are assuming the posterior is fully factorized; this is called a **mean field** approximation.

A Gaussian approximation is not always suitable for all parameters. For example, in our 1d example we have the constraint that $\theta \in [0, 1]$. We could use a variational approximation of the form $q(\theta|\boldsymbol{\psi}) = \text{Beta}(\theta|a, b)$, where $\boldsymbol{\psi} = (a, b)$. However choosing a suitable form of variational distribution requires some level of expertise. To create a more easily applicable, or "turn-key", method, that works on a wide range of models, we can use a method called **automatic differentiation variational inference** or **ADVI** [Kuc+16]. This uses the change of variables method to convert the parameters to an unconstrained form, and then computes a Gaussian variational approximation. The method also uses automatic differentiation to derive the Jacobian term needed to compute the density of the transformed variables. See Section 10.3.5 for details.

We now apply ADVI to our 1d beta-Bernoulli model. Let $\theta = \sigma(z)$, where we replace $p(\theta|\mathcal{D})$ with $q(z|\boldsymbol{\psi}) = \mathcal{N}(z|\mu, \sigma)$, where $\boldsymbol{\psi} = (\mu, \sigma)$. We optimize a stochastic approximation to the ELBO using SGD. The results are shown in Figure 7.3 and seem reasonable.

7.4.5 Markov Chain Monte Carlo (MCMC)

Although VI is fast, it can give a biased approximation to the posterior, since it is restricted to a specific function form $q \in \mathcal{Q}$. A more flexible approach is to use a non-parametric approximation in terms of a set of samples, $q(\boldsymbol{\theta}) \approx \frac{1}{S}\sum_{s=1}^{S} \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s)$. This is called a **Monte Carlo approximation**. The key issue is how to create the posterior samples $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$ efficiently, without having to evaluate the normalization constant $p(\mathcal{D}) = \int p(\boldsymbol{\theta}, \mathcal{D}) d\boldsymbol{\theta}$.

For low dimensional problems, we can use methods such as **importance sampling**, which we discuss in Section 11.5. However, for high dimensional problems, it is more common to use **Markov chain Monte Carlo** or **MCMC**. We give the details in Chapter 12, but give a brief introduction here.

The most common kind of MCMC is known as the **Metropolis Hastings algorithm**. The basic idea behind MH is as follows: we start at a random point in parameter space, and then perform a

random walk, by sampling new states (parameters) from a **proposal distribution** $q(\boldsymbol{\theta}'|\boldsymbol{\theta})$. If q is chosen carefully, the resulting Markov chain distribution will satisfy the property that the fraction of time we visit each point in space is proportional to the posterior probability. The key point is that to decide whether to move to a newly proposed point $\boldsymbol{\theta}'$ or to stay in the curent point $\boldsymbol{\theta}$, we only need to evaluate the unnormalized density ratio

$$\frac{p(\boldsymbol{\theta}|\mathcal{D})}{p(\boldsymbol{\theta}'|\mathcal{D})} = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})/p(\mathcal{D})}{p(\mathcal{D}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')/p(\mathcal{D})} = \frac{p(\mathcal{D},\boldsymbol{\theta})}{p(\mathcal{D},\boldsymbol{\theta}')} \tag{7.29}$$

This avoids the need to compute the normalization constant $p(\mathcal{D})$. (In practice we usually work with log probabilities, instead of joint probabilities, to avoid numerical issues.)

We see that the input to the algorithm is just a function that computes the log joint density, $\log p(\boldsymbol{\theta}, \mathcal{D})$, as well as a proposal distribution $q(\boldsymbol{\theta}'|\boldsymbol{\theta})$ for deciding which states to visit next. It is common to use a Gaussian distribution for the proposal, $q(\boldsymbol{\theta}'|\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}'|\boldsymbol{\theta}, \sigma\mathbf{I})$; this is called the **random walk Metropolis** algorithm. However, this can be very inefficient, since it is blindly walking through the space, in the hopes of finding higher probability regions.

In models that have conditional independence structure, it is often easy to compute the **full conditionals** $p(\boldsymbol{\theta}_d|\boldsymbol{\theta}_{-d}, \mathcal{D})$ for each variable d, one at a time, and then sample from them. This is like a stochastic analog of coordinate ascent, and is called **Gibbs sampling** (see Section 12.3 for details).

For models where all unknown variables are continuous, we can often compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}, \mathcal{D})$. We can use this gradient information to guide the proposals into regions of space with higher probability. This approach is called **Hamiltonian Monte Carlo** or **HMC**, and is one of the most widely used MCMC algorithms due to its speed. For details, see Section 12.5.

We apply HMC to our beta-Bernoulli model in Figure 7.4. (We use a logit transformation for the parameter.) In panel b, we show samples generated by the algorithm from 4 parallel Markov chains. We see that they oscillate around the true posterior, as desired. In panel a, we compute a kernel density estimate from the posterior samples from each chain; we see that the result is a good approximation to the true posterior in Figure 7.2.

7.4.6 Sequential Monte Carlo

MCMC is like a stochastic local search algorithm, in that it makes moves through the state space of the posterior distribution, comparing the current value to proposed neighboring values. An alternative approach is to use perform inference using a sequence of different distributions, from simpler to more complex, with the final distribution being equal to the target posterior. This is called **sequential Monte Carlo** or **SMC**. This approach, which is more similar to tree search than local search, has various advantages over MCMC, which we discuss in Chapter 13.

A common application of SMC is to **sequential Bayesian inference**, in which we recursively compute (i.e., in an online fashion) the posterior $p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t})$, where $\mathcal{D}_{1:t} = \{(\boldsymbol{x}_n, y_n) : n = 1:t\}$ is all the data we have seen so far. This sequence of distributions converges to the full batch posterior $p(\boldsymbol{\theta}|\mathcal{D})$ once all the data has been seen. However, the approach can also be used when the data is arriving in a continual, unending stream, as in state-space models (see Chapter 29). The application of SMC to such dynamical models is known as **particle filtering**. See Section 13.2 for details.

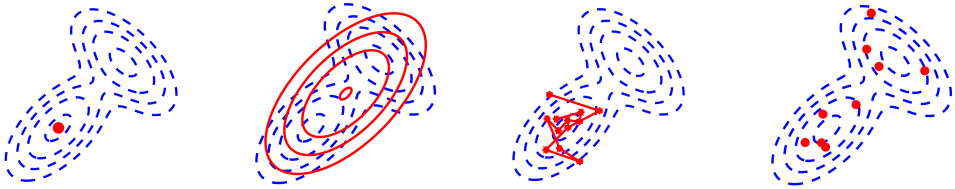


Figure 7.5: Different approximations to a bimodal 2d distribution. (a) Local MAP estimate. (b) Parametric Gaussian approximation. (c) Correlated samples from near one mode. (d) Independent samples from the distribution. Adapted from Figure 2 of [PY14]. Used with kind permission of George Panadreou.

7.4.7 Challenging posteriors

In many applications, the posterior can be high dimensional and multimodal. Approximating such distributions can be quite challenging. In Figure 7.5, we give a simple 2d example. We compare MAP estimation (which does not capture any uncertainty), a Gaussian parametric approximation such as the Laplace approximation or variational inference, and a nonparametric approximation in terms of samples. If the samples are generated from MCMC, they are serially correlated, and may only explore a local model (see panel b). However, ideally we can draw independent samples from the entire support of the distribution, as shown in panel d. We may also be able to fit a local parametric approximation around each such sample (c.f., Section 17.3.9.1), to get a semi-parametric approximation to the posterior.

7.5 Evaluating approximate inference algorithms

There are many different inference algorithms each of which make different tradeoffs between speed, accuracy, generality, simplicity, etc. This makes it hard to compare them on an equal footing. However, a common approach is to evaluate the accuracy of the approximation as a function of compute time.

We give an example of this in Figure 7.6, where we plot performance for several different inference algorithms applied to the following simple 1d model:

$$p(z) = \mathcal{N}(z|0, 100) \tag{7.30}$$

$$p(x_i|z) = 0.5\mathcal{N}(x_i|z, 1) + 0.5\mathcal{N}(x_i|0, 10) \tag{7.31}$$

That is, each measurement x_i is either a noisy copy of the hidden quantity of interest, z, or is an outlier coming from a uniform background noise model, approximated by a Gaussian with a large variance, $\mathcal{N}(0, 10)$. The goal is to compute $p(z|\boldsymbol{x}_{1:N})$. (Tom Minka (who created this example) calls this the **clutter problem**.)

Since this is a 1d problem, we can compute the exact answer using numerical integration. In Figure 7.6, we plot the accuracy of the posterior mean estimate using various approximate inference methods, namely: the Laplace approximation (Section 7.4.3), variational Bayes (Section 10.2.3), expectation propagation (Section 10.7), importance sampling (Section 11.5), and Gibbs sampling (Section 12.3). We see that the error smoothly decreases for the 3 deterministic methods (Laplace,

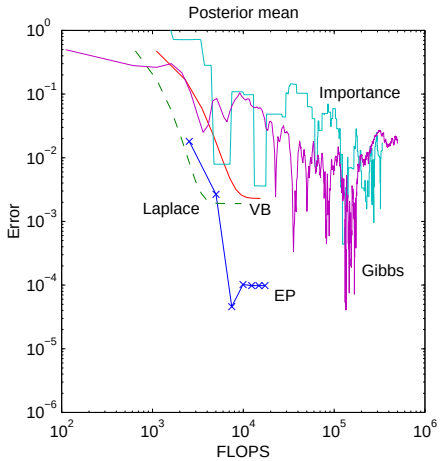


Figure 7.6: Accuracy vs compute time for different inference methods. Code source: https://github.com/tminka/ep-clutter-example. From Figure 1 of [Min01b]. Used with kind permission of Tom Minka.

VB and EP). For the 2 stochastic methods (IS and Gibbs), the error decreases on average, but the performance is quite noisy.

In principle, the Monte Carlo methods will converge to a zero overall error, since they are unbiased estimators, but it is clear that their variance is quite high. The deterministic methods, by comparison, converge to a finite error, so they are biased, but their variance is much lower. We can trade off bias and variance by creating hybrid algorithms, that combine different techniques. We will see examples of this in later chapters.

When we cannot compute the true target posterior distribution to compare to, evaluation is harder, but there are some approaches than can be used in certain cases. For some methods for assessing variational inference, see e.g., [Yao+18b; Hug+20]. For some methods for assessing Monte Carlo methods, see [CGR06; CTM17; GAR16]. For assessing posterior predictive distributions, see Section 14.2.3.

8 Inference for state-space models

8.1 Introduction

In this chapter, we consider the task of posterior inference in **state-space models** (SSM). We discuss SSMs in more detail in Chapter 29, but we can think of them as latent variable sequence models with the conditional independencies shown by the chain-structured graphical model Figure 8.1. The corresponding joint distribution has the form

$$p(\boldsymbol{y}_{1:T}, \boldsymbol{z}_{1:T}|\boldsymbol{u}_{1:T}) = \left[p(\boldsymbol{z}_1|\boldsymbol{u}_1)\prod_{t=2}^{T} p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{u}_t)\right]\left[\prod_{t=1}^{T} p(\boldsymbol{y}_t|\boldsymbol{z}_t, \boldsymbol{u}_t)\right] \tag{8.1}$$

where $\boldsymbol{z}_t$ are the hidden variables at time t, $\boldsymbol{y}_t$ are the observations (outputs), and $\boldsymbol{u}_t$ are the optional inputs.

Given the sequence of observations, and a known model, one of the main tasks with SSMs is to perform posterior inference about the hidden states; this is also called **state estimation**. At each time step t, there are multiple forms of posterior we may be interested in computing, including the **filtering distribution** $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$, the **smoothing distribution** $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T})$ (note that this conditions on future data $T \geq t$), and the **fixed-lag smoothing distribution** $p(\boldsymbol{z}_{t-\ell}|\boldsymbol{y}_{1:t})$ (note that this infers ℓ steps in the past given data up to the present). We may also want to compute the **predictive distribution** h steps into the future:

$$p(\boldsymbol{y}_{t+h}|\boldsymbol{y}_{1:t}) = \sum_{\boldsymbol{z}_{t+h}} p(\boldsymbol{y}_{t+h}|\boldsymbol{z}_{t+h})p(\boldsymbol{z}_{t+h}|\boldsymbol{y}_{1:t}) \tag{8.2}$$

where the hidden state predictive distribution is

$$p(\boldsymbol{z}_{t+h}|\boldsymbol{y}_{1:t}) = \sum_{\boldsymbol{z}_{t:t+h-1}} p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t)p(\boldsymbol{z}_{t+2}|\boldsymbol{z}_{t+1})\cdots p(\boldsymbol{z}_{t+h}|\boldsymbol{z}_{t+h-1}) \tag{8.3}$$

See Figure 8.2 for a summary of these distributions. In addition to computing posterior marginals, we may want to compute the most probable hidden sequence, $\text{argmax}_{\boldsymbol{z}_{1:T}} p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T})$, or sample sequences from the posterior, $\boldsymbol{z}_{1:T} \sim p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T})$. We discuss algorithms for all these tasks later in this chapter. See also [Sar13; Tri21].

8.2 Inference based on the HMM filter

In this section, we consider inference for SSMs where all the hidden variables are discrete. This is known as a **hidden Markov model** or **HMM**. This is discussed in detail in Section 29.2, but

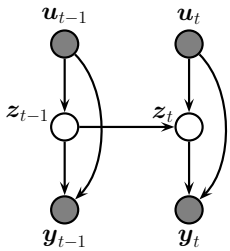


Figure 8.1: A state-space modelrepresented as a graphical model. $\boldsymbol{z}_t$ are the hidden variables at time t, $\boldsymbol{y}_t$ are the observations (outputs), and $\boldsymbol{u}_t$ are the optional inputs.

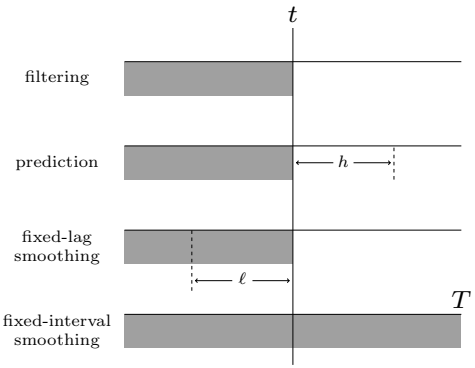


Figure 8.2: The main kinds of inference for state-space models. The shaded region is the interval for which we have data. The arrow represents the time step at which we want to perform inference. t is the current time, T is the sequence length, ℓ is the lag and h is the prediction horizon. Used with kind permission of Peter Chang.

in brief, it corresponds to the model in Equation (8.1) where $p(z_1 = k) = \pi_k$ is the initial state distribution, $p(z_t = k|z_{t-1} = j) = A_{jk}$ is the state transition matrix (assumed stationary), and $p(\boldsymbol{y}_t|z_t = k)$ is some conditional distribution over observations. For notational simplicity, we assume all conditional disributions are stationary (the same over time), and we ignore inputs. Note, however, that the algorithms we discuss can be generalized to the non-stationary case, and also to to 1d undirected graphical models, such as linear-chain CRFs (Section 4.4).

8.2.1 Example: casino HMM

Before explaining the algorithms, we introduce a simple example of an HMM, which we will use to illustrate the methods.

Suppose we are in a casino and observe a series of die rolls, $y_t \in \{1, 2, \ldots, 6\}$. Being a keen-eyed statistician, we notice that the distribution of values is not what we expect from a fair die: it seems that there are occasional "streaks", in which 6s seem to show up more often than other values. (This

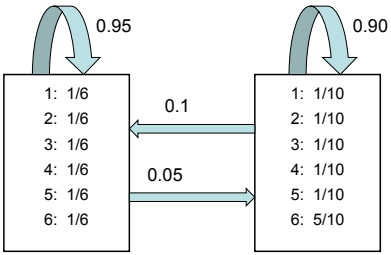


Figure 8.3: The state transition matrix **A** *and observation matrix* **B** *for the casino HMM. Adapted from [Dur+98, p54].*

example is from [Dur+98], who call this setup the **occasionally dishonest casino**.) We would like to **segment** the time series into regimes corresponding to the use of a fair die and a loaded die.

Let $A_{jk} = p(z_t = k|z_{t-1} = j)$ be the state transition matrix, and $B_{kl} = p(y_t = l|z_t = k)$ be the observation matrix corresponding to a categorical distribution over values of the die face. Most of the time the casino uses a fair die, $z = 1$, but occasionally it switches to a loaded die, $z = 2$, for a short period. If $z = 1$ the observation distribution is a uniform categorical distribution over the symbols $\{1, \ldots, 6\}$. If $z = 2$, the observation distribution is skewed towards face 6. That is,

$$p(y_t|z_t = 1) = \text{Cat}(y_t|[1/6, \ldots, 1/6]) \tag{8.4}$$
$$p(y_t|z_t = 2) = \text{Cat}(y_t|[1/10, 1/10, 1/10, 1/10, 1/10, 5/10]) \tag{8.5}$$

See Figure 8.3 for an illustration of the state transition diagram **A** and the emission distributions **B**.

If we sample from this model, we may generate data such as the following:

```
hid:  1111111111222211111111111111111111112222222221222211111111111111111111
obs:  1355534526553366316351551526232112113462221263264265422344645323242361
```

Here obs refers to the observation and hid refers to the hidden state (1 is fair and 2 is loaded). In the full sequence of length 300, we find the empirical fraction of times that we observe a 6 in hidden state 1 to be 0.149, and in state to be 0.472, which is very close to the expected fractions. (See casino_hmm.ipynb for the code.)

Of course, when using an HMM in practice, we just see the observed sequence, and need to infer the hidden sequence, as we discuss below.

8.2.2 Forwards filtering

The **Bayes filter** is an algorithm for recursively computing the **belief state** $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$ given the prior belief from the previous step, $p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1})$, the new observation $\boldsymbol{y}_t$, and the model. This can be done using **sequential Bayesian updating**. For a dynamical model, this reduces to the **predict-update** cycle described below.

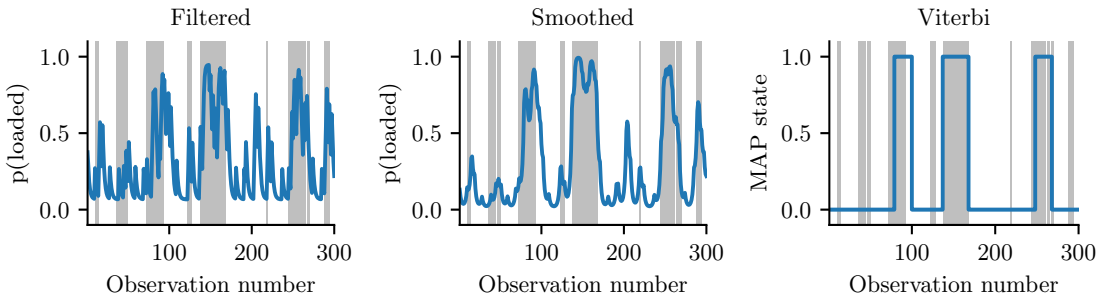


Figure 8.4: Inference in the dishonest casino. Vertical gray bars denote times when the hidden state corresponded to the loaded die. Blue lines represent the posterior probability of being in that state given different subsets of observed data. If we recover the true state exactly, the blue curve will transition at the same time as the gray bars. (a) Filtered estimate. (b) Smoothed estimates. (c) MAP trajectory. Generated by casino_hmm.ipynb.

8.2.2.1 Prediction step

The **prediction step** is just the **Chapman-Kolmogorov equation**:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1})d\boldsymbol{z}_{t-1} \tag{8.6}$$

The prediction step computes the one-step-ahead predictive distribution for the latent state, which updates the posterior from the previous time step into the prior for the current step.[1]

8.2.2.2 Update step

The **update step** is just Bayes rule:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) = \frac{1}{Z_t}p(\boldsymbol{y}_t|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) \tag{8.7}$$

where the normalization constant is

$$Z_t = \int p(\boldsymbol{y}_t|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})d\boldsymbol{z}_t = p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) \tag{8.8}$$

We can use the normalization constants to compute the log likelihood of the sequence as follows:

$$\log p(\boldsymbol{y}_{1:T}) = \sum_{t=1}^{T} \log p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = \sum_{t=1}^{T} \log Z_t \tag{8.9}$$

where we define $p(\boldsymbol{y}_1|\boldsymbol{y}_0) = p(\boldsymbol{y}_1)$. This is useful for computing the MLE of the parameters (see Section 29.4.2).

1. The prediction step is not needed at $t = 1$ if $p(\boldsymbol{z}_1)$ is provided as input to the model. However, if we just provide $p(\boldsymbol{z}_0)$, we need to compute $p(\boldsymbol{z}_1|\boldsymbol{y}_{1:0}) = p(\boldsymbol{z}_1)$ by applying the prediction step.

8.2.2.3 Implementation

When the latent states $\boldsymbol{z}_t$ are discrete, the above integrals become sums which can be implemented as follows. First we define the belief state as $\alpha_t(j) \triangleq p(z_t = j|\boldsymbol{y}_{1:t})$, the local evidence as $\lambda_t(j) \triangleq p(\boldsymbol{y}_t|z_t = j)$, and the transition matrix $A_{i,j} = p(z_t = j|z_{t-1} = i)$. Then the predict step becomes

$$\alpha_{t|t-1}(j) \triangleq p(z_t = j|\boldsymbol{y}_{1:t-1}) = \sum_i \alpha_{t-1}(i)A_{i,j} \tag{8.10}$$

and the update step becomes

$$\alpha_t(j) = \frac{1}{Z_t}\lambda_t(j)\alpha_{t|t-1}(j) = \frac{1}{Z_t}\lambda_t(j)\left[\sum_i \alpha_{t-1}(i)A_{i,j}\right] \tag{8.11}$$

where the normalization constant for each time step is given by

$$Z_t \triangleq p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = \sum_{j=1}^{K} p(\boldsymbol{y}_t|\boldsymbol{z}_t = j)p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t-1}) = \sum_{j=1}^{K} \lambda_t(j)\alpha_{t|t-1}(j) \tag{8.12}$$

We can write the update equation in matrix-vector notation as follows:

$$\boldsymbol{\alpha}_t = \text{normalize}\left(\boldsymbol{\lambda}_t \odot (\mathbf{A}^\mathsf{T}\boldsymbol{\alpha}_{t-1})\right) \tag{8.13}$$

where $\odot$ represents elementwise vector multiplication, and the normalize function just ensures its argument sums to one. (See Section 8.2.5 for more discussion on normalization.)

8.2.2.4 Example

Figure 8.4(a) illustrates filtering for the casino HMM, applied to a random sequence $\boldsymbol{y}_{1:T}$ of length $T = 300$. In blue, we plot the probability that the die is in the loaded (vs fair) state, based on the evidence seen so far. The gray bars indicate time intervals during which the generative process actually switched to the loaded die. We see that the probability generally increases in the right places.

8.2.3 Backwards smoothing

In the offline setting, we want to compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T})$, which is the belief about the hidden state at time t given all the data, both past and future. This is called (fixed interval) **smoothing**. We first perform the forwards or filtering pass, and then compute the smoothed belief states by working backwards, from right (time $t = T$) to left (to $t = 1$), as we explain below. Hence this method is also called **forwards filtering backwards smoothing** or **FFBS**.

8.2.3.1 Backwards recursion

Suppose, by induction, that we have already computed $p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})$. We can convert this into a joint smoothed distribution over two consecutive time steps using

$$p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T})p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T}) \tag{8.14}$$

To derive the first term, note that from the Markov properties of the model, and Bayes rule, we have

$$p(\boldsymbol{z}_t = i|\boldsymbol{z}_{t+1} = j, \boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t = i|\boldsymbol{z}_{t+1} = j, \boldsymbol{y}_{1:t}, \cancel{\boldsymbol{y}_{t+1:T}}) \tag{8.15}$$

$$= \frac{p(\boldsymbol{z}_t = i, \boldsymbol{z}_{t+1} = j|\boldsymbol{y}_{1:t})}{p(\boldsymbol{z}_{t+1} = j|\boldsymbol{y}_{1:t})} \tag{8.16}$$

$$= \frac{p(\boldsymbol{z}_{t+1} = j|\boldsymbol{z}_t = i)p(\boldsymbol{z}_t = i|\boldsymbol{y}_{1:t})}{p(\boldsymbol{z}_{t+1} = j|\boldsymbol{y}_{1:t})} \tag{8.17}$$

Thus the joint distribution over two consecutive time steps is given by

$$p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T}) = \frac{p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})}{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})} \tag{8.18}$$

from which we get the new smoothed marginal distribution:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) \int \left[\frac{p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t)p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})}{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})}\right] d\boldsymbol{z}_{t+1} \tag{8.19}$$

$$= \int p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}) \frac{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})}{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})} d\boldsymbol{z}_{t+1} \tag{8.20}$$

Intuitively we can interpret this as follows: we start with the two-slice filtered distribution, $p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})$, and then we divide out the old $p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})$ and multiply in the new $p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})$, and then marginalize out $\boldsymbol{z}_{t+1}$.

8.2.3.2 Implementation

Let us define the two-slice marginal using the following notation:

$$\xi_{t,t+1}(i,j) \triangleq p(\boldsymbol{z}_t = i, \boldsymbol{z}_{t+1} = j|\boldsymbol{y}_{1:T}) \tag{8.21}$$

We can then rewrite Equation (8.18) as follows:

$$\xi_{t,t+1}(i,j) = \alpha_t(i)A_{i,j}\frac{\gamma_{t+1}(j)}{\alpha_{t+1|t}(j)} \tag{8.22}$$

where

$$\alpha_{t+1|t}(j) = p(\boldsymbol{z}_{t+1} = j|\boldsymbol{y}_{1:t}) = \sum_{i'} A(i',j)\alpha_t(i') \tag{8.23}$$

is the one-step-ahead predictive distribution. We can interpret the ratio in Equation (8.22) as dividing out the old estimate of $\boldsymbol{z}_{t+1}$ given $\boldsymbol{y}_{1:t}$, namely $\alpha_{t+1|t}$, and multiplying in the new estimate given $\boldsymbol{y}_{1:T}$, namely γ_{t+1}.

Finally we can recover the one-slice posterior marginal by marginalizing:

$$\gamma_t(i) \triangleq p(\boldsymbol{z}_t = i|\boldsymbol{y}_{1:T}) = \sum_j \xi_{t,t+1}(i,j) = \alpha_t(i)\sum_j \left[A_{i,j}\frac{\gamma_{t+1}(j)}{\alpha_{t+1|t}(j)}\right] \tag{8.24}$$

We initialize the recursion using $\gamma_T(j) = \alpha_T(j) = p(\boldsymbol{z}_T = j|\boldsymbol{y}_{1:T})$.

8.2.3.3 Example

In Figure 8.4(a-b), we compare filtering and smoothing for the casino HMM. We see that the posterior distributions when conditioned on all the data (past and future) are indeed smoother than when just conditioned on the past (filtering).

To understand this behavior intuitively, consider a detective trying to figure out who committed a crime. As they move through the crime scene, his uncertainty is high until he finds the key clue; then he has an "aha" moment, his uncertainty is reduced, and all the previously confusing observations are, in **hindsight**, easy to explain. Thus we see that, given all the data (including finding the clue), it is much easier to infer the state of the world.

8.2.4 The forwards-backwards algorithm

In this section, we present a more common approach to smoothing in HMMs known as the **forwards-backwards** or **FB** algorithm [Rab89]. In the forwards pass, we compute $\alpha_t(j) = p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t})$ as before. In the backwards pass, we compute the conditional likelihood

$$\beta_t(j) \triangleq p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t = j) \tag{8.25}$$

We then combine these using

$$\gamma_t(j) = p(\boldsymbol{z}_t = j|\boldsymbol{y}_{t+1:T}, \boldsymbol{y}_{1:t}) \propto p(\boldsymbol{z}_t = j, \boldsymbol{y}_{t+1:T}|\boldsymbol{y}_{1:t}) \tag{8.26}$$

$$= p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t})p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t = j, \cancel{\boldsymbol{y}_{1:t}}) = \alpha_t(j)\beta_t(j) \tag{8.27}$$

In matrix notation, this becomes

$$\boldsymbol{\gamma}_t = \text{normalize}(\boldsymbol{\alpha}_t \odot \boldsymbol{\beta}_t) \tag{8.28}$$

Note that the forwards and backwards passes can be computed independently, but both need access to the local evidence $p(\boldsymbol{y}_t|\boldsymbol{z}_t)$. The results are only combined at the end. This is therefore called **two-filter smoothing** [Kit04].

8.2.4.1 Backwards recursion

We can recursively compute the β's in a right-to-left fashion as follows:

$$\beta_{t-1}(i) = p(\boldsymbol{y}_{t:T}|\boldsymbol{z}_{t-1} = i) \tag{8.29}$$

$$= \sum_j p(\boldsymbol{z}_t = j, \boldsymbol{y}_t, \boldsymbol{y}_{t+1:T}|\boldsymbol{z}_{t-1} = i) \tag{8.30}$$

$$= \sum_j p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t = j, \cancel{\boldsymbol{y}_t}, \cancel{\boldsymbol{z}_{t-1} = i})p(\boldsymbol{z}_t = j, \boldsymbol{y}_t|\boldsymbol{z}_{t-1} = i) \tag{8.31}$$

$$= \sum_j p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t = j)p(\boldsymbol{y}_t|\boldsymbol{z}_t = j, \cancel{\boldsymbol{z}_{t-1} = i})p(\boldsymbol{z}_t = j|\boldsymbol{z}_{t-1} = i) \tag{8.32}$$

$$= \sum_j \beta_t(j)\lambda_t(j)A_{i,j} \tag{8.33}$$

We can write the resulting equation in matrix-vector form as

$$\boldsymbol{\beta}_{t-1} = \mathbf{A}(\boldsymbol{\lambda}_t \odot \boldsymbol{\beta}_t) \tag{8.34}$$

The base case is

$$\beta_T(i) = p(\boldsymbol{y}_{T+1:T}|\boldsymbol{z}_T = i) = p(\emptyset|\boldsymbol{z}_T = i) = 1 \tag{8.35}$$

which is the probability of a non-event.

Note that $\boldsymbol{\beta}_t$ is not a probability distribution over states, since it does not need to satisfy $\sum_j \beta_t(j) = 1$. However, we usually normalize it to avoid numerical underflow (see Section 8.2.5).

8.2.4.2 Two-slice smoothed marginals

We can compute the two-slice marginals using the output of the forwards-backwards algorithm as follows:

$$\begin{aligned}
p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T}) &= p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}, \boldsymbol{y}_{t+1:T}) && (8.36)\\
&\propto p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t, \boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:t})p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}) && (8.37)\\
&= p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_{t+1})p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}) && (8.38)\\
&= p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_{t+1})p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) && (8.39)\\
&= p(\boldsymbol{y}_{t+1}, \boldsymbol{y}_{t+2:T}|\boldsymbol{z}_{t+1})p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) && (8.40)\\
&= p(\boldsymbol{y}_{t+1}|\boldsymbol{z}_{t+1})p(\boldsymbol{y}_{t+2:T}|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{t+1})p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) && (8.41)\\
&= p(\boldsymbol{y}_{t+1}|\boldsymbol{z}_{t+1})p(\boldsymbol{y}_{t+2:T}|\boldsymbol{z}_{t+1})p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) && (8.42)
\end{aligned}$$

We can rewrite this in terms of the already computed quantities as follows:

$$\xi_{t,t+1}(i,j) \propto \lambda_{t+1}(j)\beta_{t+1}(j)\alpha_t(i)A_{i,j} \tag{8.43}$$

Or in matrix-vector form:

$$\boldsymbol{\xi}_{t,t+1} \propto \mathbf{A} \odot \left[\boldsymbol{\alpha}_t(\boldsymbol{\lambda}_{t+1} \odot \boldsymbol{\beta}_{t+1})^\mathsf{T}\right] \tag{8.44}$$

Since $\boldsymbol{\alpha}_t \propto \boldsymbol{\lambda}_t \odot \boldsymbol{\alpha}_{t|t-1}$, we can also write the above equation as follows:

$$\boldsymbol{\xi}_{t,t+1} \propto \mathbf{A} \odot (\boldsymbol{\lambda}_t \odot \boldsymbol{\alpha}_{t|t-1}) \odot (\boldsymbol{\lambda}_{t+1} \odot \boldsymbol{\beta}_{t+1})^\mathsf{T} \tag{8.45}$$

This can be interpreted as a product of incoming messages and local factors, as shown in Figure 8.5. In particular, we combine the factors $\boldsymbol{\alpha}_{t|t-1} = p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})$, $\mathbf{A} = p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t)$, $\boldsymbol{\lambda}_t \propto p(\boldsymbol{y}_t|\boldsymbol{z}_t)$, $\boldsymbol{\lambda}_{t+1} \propto p(\boldsymbol{y}_{t+1}|\boldsymbol{z}_{t+1})$, and $\boldsymbol{\beta}_{t+1} \propto p(\boldsymbol{y}_{t+2:T}|\boldsymbol{z}_{t+1})$ to get $p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}, \boldsymbol{y}_t, \boldsymbol{y}_{t+1}, \boldsymbol{y}_{t+2:T}|\boldsymbol{y}_{1:t-1})$, which we can then normalize.

8.2.5 Numerically stable implementation

In most publications on HMMs, such as [Rab89], the forwards message is defined as the following unnormalized *joint* probability:

$$\alpha'_t(j) = p(\boldsymbol{z}_t = j, \boldsymbol{y}_{1:t}) = \lambda_t(j)\left[\sum_i \alpha'_{t-1}(i)A_{i,j}\right] \tag{8.46}$$

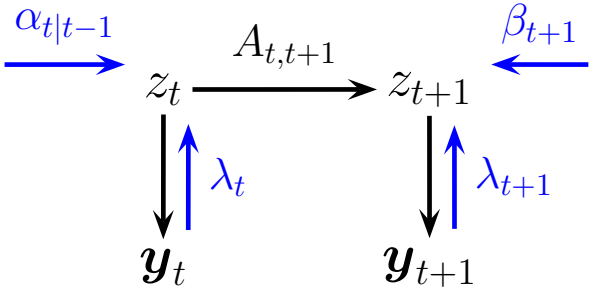


Figure 8.5: Computing the two-slice joint distribution for an HMM from the forwards messages, backwards messages, and local evidence messages.

We instead define the forwards message as the normalized *conditional* probability

$$\alpha_t(j) = p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t}) = \frac{1}{Z_t}\lambda_t(j)\left[\sum_i \alpha_{t-1}(i)A_{i,j}\right] \tag{8.47}$$

The unnormalized (joint) form has several problems. First, it rapidly suffers from numerical underflow, since the probability of the joint event that $(\boldsymbol{z}_t = j, \boldsymbol{y}_{1:t})$ is vanishingly small.[2] Second, it is less interpretable, since it is not a distribution over states. Third, it precludes the use of approximate inference methods that try to approximate posterior distributions (we will see such methods later). We therefore always used the normalized (conditional) form.

Of course, the two definitions only differ by a multiplicative constant, since $p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t}) = p(\boldsymbol{z}_t = j, \boldsymbol{y}_{1:t})/p(\boldsymbol{y}_{1:t})$ [Dev85]. So the *algorithmic* difference is just one line of code (namely the presence or absence of a call to the `normalize` function). Nevertheless, we feel it is better to present the normalized version, since it will encourage readers to implement the method properly (i.e., normalizing after each step to avoid underflow).

In practice it is more numerically stable to compute the log probabilities $\ell_t(j) = \log p(\boldsymbol{y}_t|\boldsymbol{z}_t = j)$ of the evidence, rather than the probabilities $\lambda_t(j) = p(\boldsymbol{y}_t|\boldsymbol{z}_t = j)$. We can combine the state conditional log likelihoods $\lambda_t(j)$ with the state prior $p(\boldsymbol{z}_t = j|\boldsymbol{y}_{1:t-1})$ by using the log-sum-exp trick, as in Equation (28.30).

8.2.6 Time and space complexity

It is clear that a straightforward implementation of the forwards-backwards algorithm takes $O(K^2T)$ time, since we must perform a $K \times K$ matrix multiplication at each step. For some applications, such as speech recognition, K is very large, so the $O(K^2)$ term becomes prohibitive. Fortunately, if the transition matrix is sparse, we can reduce this substantially. For example, in a sparse left-to-right transition matrix (e.g., Figure 8.7(a)), the algorithm takes $O(TK)$ time.

In some cases, we can exploit special properties of the state space, even if the transition matrix is not sparse. In particular, suppose the states represent a discretization of an underlying continuous

2. For example, if the observations are independent of the states, we have $p(\boldsymbol{z}_t = j, \boldsymbol{y}_{1:t}) = p(\boldsymbol{z}_t = j)\prod_{i=1}^t p(\boldsymbol{y}_i)$, which becomes exponentially small with t.

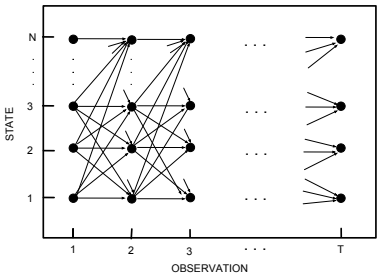


Figure 8.6: The trellis of states vs time for a Markov chain. Adapted from [Rab89].

state-space, and the transition matrix has the form $A_{i,j} \propto \rho(\boldsymbol{z}_j - \boldsymbol{z}_i)$, where $\boldsymbol{z}_i$ is the continuous vector represented by state i and $\rho(u)$ is some scalar cost function, such as Euclidean distance. Then one can implement the forwards-backwards algorithm in $O(TK \log K)$ time. The key is to rewrite Equation (8.10) as a convolution,

$$\alpha_{t|t-1}(j) = p(z_t = j|\boldsymbol{y}_{1:t-1}) = \sum_i \alpha_{t-1}(i) A_{i,j} = \sum_i \alpha_{t-1}(i)\rho(j-i) \tag{8.48}$$

and then to apply the Fast Fourier Transform. (A similar transformation can be applied in the backwards pass.) This is very useful for models with large state spaces. See [FHK03] for details.

We can also reduce inference to $O(\log T)$ time by using a **parallel prefix scan** operator that can be run efficiently on GPUs. For details, see [HSGF21].

In some cases, the bottleneck is memory, not time. In particular, to compute the posteriors $\boldsymbol{\gamma}_t$, we must store the fitered distributions $\boldsymbol{\alpha}_t$ for $t = 1, \ldots, T$ until we do the backwards pass. It is possible to devise a simple divide-and-conquer algorithm that reduces the space complexity from $O(KT)$ to $O(K \log T)$ at the cost of increasing the running time from $O(K^2 T)$ to $O(K^2 T \log T)$. The basic idea is to store $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ vectors at a logarithmic number of intermediate checkpoints, and then recompute the missing messages on demand from these checkpoints. See [BMR97; ZP00] for details.

8.2.7 The Viterbi algorithm

The MAP estimate is (one of) the sequences with maximum posterior probabiltiy:

$$\boldsymbol{z}^*_{1:T} = \operatorname*{argmax}_{\boldsymbol{z}_{1:T}} p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T}) = \operatorname*{argmax}_{\boldsymbol{z}_{1:T}} \log p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T}) \tag{8.49}$$

$$= \operatorname*{argmax}_{\boldsymbol{z}_{1:T}} \log \pi_1(\boldsymbol{z}_1) + \log \lambda_1(\boldsymbol{z}_1) + \sum_{t=2}^{T} [\log A(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t) + \log \lambda_t(\boldsymbol{z}_t)] \tag{8.50}$$

This is equivalent to computing a shortest path through the **trellis diagram** in Figure 8.6, where the nodes are possible states at each time step, and the node and edge weights are log probabilities. This can be computed in $O(TK^2)$ time using the **Viterbi algorithm** [Vit67], as we explain below.

8.2.7.1 Forwards pass

Recall the (unnormalized) forwards equation

$$\alpha'_t(j) = p(\boldsymbol{z}_t = j, \boldsymbol{y}_{1:t}) = \sum_{\boldsymbol{z}_1,\ldots,\boldsymbol{z}_{t-1}} p(\boldsymbol{z}_{1:t-1}, \boldsymbol{z}_t = j, \boldsymbol{y}_{1:t}) \tag{8.51}$$

Now suppose we replace sum with max to get

$$\delta_t(j) \triangleq \max_{\boldsymbol{z}_1,\ldots,\boldsymbol{z}_{t-1}} p(\boldsymbol{z}_{1:t-1}, \boldsymbol{z}_t = j, \boldsymbol{y}_{1:t}) \tag{8.52}$$

This is the maximum probability we can assign to the data so far if we end up in state j. The key insight is that the most probable path to state j at time t must consist of the most probable path to some other state i at time $t-1$, followed by a transition from i to j. Hence

$$\delta_t(j) = \lambda_t(j) \left[\max_i \delta_{t-1}(i) A_{i,j}\right] \tag{8.53}$$

We initialize by setting $\delta_1(j) = \pi_j \lambda_1(j)$.

We often work in the log domain to avoid numerical issues. Let $\delta'_t(j) = -\log \delta_t(j)$, $\lambda_t(j) = -\log p(\boldsymbol{y}_t|\boldsymbol{z}_t = j)$, $A'(i,j) = -\log p(\boldsymbol{z}_t = j|\boldsymbol{z}_{t-1} = i)$. Then we have

$$\delta'_t(j) = \lambda'_t(j) + \left[\min_i \delta'_{t-1}(i) + A'(i,j)\right] \tag{8.54}$$

We also need to keep track of the most likely previous (**ancestor**) state, for each possible state that we end up in:

$$a_t(j) = \operatorname*{argmax}_i \delta_{t-1}(i) A_{i,j} = \operatorname*{argmin}_i \delta'_{t-1}(i) + A'(i,j) \tag{8.55}$$

That is, $a_t(j)$ stores the identity of the previous state on the most probable path to $\boldsymbol{z}_t = j$. We will see why we need this in Section 8.2.7.2.

8.2.7.2 Backwards pass

In the backwards pass, we compute the most probable sequence of states using a **traceback** procedure, as follows: $\boldsymbol{z}^*_t = a_{t+1}(\boldsymbol{z}^*_{t+1})$, where we initialize using $\boldsymbol{z}^*_T = \arg\max_i \delta_T(i)$. This is just following the chain of ancestors along the MAP path.

If there is a unique MAP estimate, the above procedure will give the same result as picking $\hat{z}_t = \operatorname{argmax}_j \gamma_t(j)$, computed by forwards-backwards, as shown in [WF01b]. However, if there are multiple posterior modes, the latter approach may not find any of them, since it chooses each state independently, and hence may break ties in a manner that is inconsistent with its neighbors. The traceback procedure avoids this problem, since once $\boldsymbol{z}_t$ picks its most probable state, the previous nodes condition on this event, and therefore they will break ties consistently.

8.2.7.3 Example

In Figure 8.4(c), we show the Viterbi trace for the casino HMM. We see that, most of the time, the estimated state corresponds to the true state.

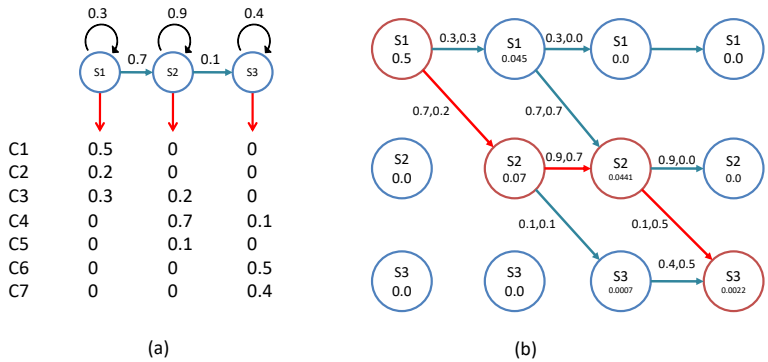


Figure 8.7: Illustration of Viterbi decoding in a simple HMM for speech recognition. (a) A 3-state HMM for a single phone. We are visualizing the state transition diagram. We assume the observations have been vector quantized into 7 possible symbols, $C_1, \ldots, C_7$. Each state s_1, s_2, s_3 has a different distribution over these symbols. Adapted from Figure 15.20 of [RN02]. (b) Illustration of the Viterbi algorithm applied to this model, with data sequence $C1, C3, C4, C6$. The columns represent time, and the rows represent states. The numbers inside the circles represent the $\delta_t(j)$ value for that state. An arrow from state i at $t-1$ to state j at t is annotated with two numbers: the first is the probability of the $i \to j$ transition, and the second is the probability of generating observation $\boldsymbol{y}_t$ from state j. The red lines/ circles represent the most probable sequence of states. Adapted from Figure 24.27 of [RN95].

In Figure 8.7, we give a detailed worked example of the Viterbi algorithm, based on [Rus+95]. Suppose we observe the sequence of discrete observations $\boldsymbol{y}_{1:4} = (C_1, C_3, C_4, C_6)$, representing codebook entries in a vector-quantized version of a speech signal. The model starts in state $\boldsymbol{z}_1 = S_1$. The probability of generating $x_1 = C_1$ in $\boldsymbol{z}_1$ is 0.5, so we have $\delta_1(1) = 0.5$, and $\delta_1(i) = 0$ for all other states. Next we can self-transition to S_1 with probability 0.3, or transition to S_2 with proabability 0.7. If we end up in S_1, the probability of generating $x_2 = C_3$ is 0.3; if we end up in S_2, the probability of generating $x_2 = C_3$ is 0.2. Hence we have

$$\delta_2(1) = \delta_1(1)A(1,1)\lambda_2(1) = 0.5 \cdot 0.3 \cdot 0.3 = 0.045 \tag{8.56}$$

$$\delta_2(2) = \delta_1(1)A(1,2)\lambda_2(2) = 0.5 \cdot 0.7 \cdot 0.2 = 0.07 \tag{8.57}$$

Thus state 2 is more probable at $t = 2$; see the second column of Figure 8.7(b). The algorithm continues in this way until we have reached the end of the sequence. One we have reached the end, we can follow the red arrows back to recover the MAP path (which is 1,2,2,3).

For more details on HMMs for automatic speech recognition (ASR) see e.g., [JM08].

8.2.7.4 Time and space complexity

The time complexity of Viterbi is clearly $O(K^2T)$ in general, and the space complexity is $O(KT)$, both the same as forwards-backwards. If the transition matrix has the form $A_{i,j} \propto \rho(\boldsymbol{z}_j - \boldsymbol{z}_i)$, where $\boldsymbol{z}_i$ is the continuous vector represented by state i and $\rho(u)$ is some scalar cost function, such as Euclidean distance, we can implement Viterbi in $O(TK)$ time, by using the generalized distance transform to implement Equation (8.54). See [FHK03; FH12] for details.

8.2.7.5 N-best list

There are often multiple paths which have the same likelihood. The Viterbi algorithm returns one of them, but can be extended to return the top N paths [SC90; NG01]. This is called the **N-best list**. Computing such a list can provide a better summary of the posterior uncertainty.

In addition, we can perform **discriminative reranking** [CK05] of all the sequences in $\mathcal{L}_N$, based on global features derived from $(\boldsymbol{y}_{1:T}, \boldsymbol{z}_{1:T})$. This technique is widely used in speech recognition. For example, consider the sentence "recognize speech". It is possible that the most probable interpretation by the system of this acoustic signal is "wreck a nice speech", or maybe "wreck a nice beach" (see Figure 34.1). Maybe the correct interpretation is much lower down on the list. However, by using a re-ranking system, we may be able to improve the score of the correct interpretation based on a more global context.

One problem with the N-best list is that often the top N paths are very similar to each other, rather than representing qualitatively different interpretations of the data. Instead we might want to generate a more diverse set of paths to more accurately represent posterior uncertainty. One way to do this is to sample paths from the posterior, as we discuss in Section 8.2.8. Another way is to use a determinantal point process (Section 31.9.5) which encourages points to be diverse [Bat+12; ZA12].

8.2.8 Forwards filtering, backwards sampling

Rather than computing the single most probable path, it is often useful to sample multiple paths from the posterior: $\boldsymbol{z}_{1:T}^s \sim p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T})$. We can do this by modifying the forwards filtering backwards smoothing algorithm from Section 8.2.3, so that we draw samples on the backwards pass, rather than computing marginals. This is called **forwards filtering backwards sampling** (also sometimes unfortunately abbreviated to FFBS). In particular, note that we can write the joint from right to left using

$$p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_T|\boldsymbol{y}_{1:T})p(\boldsymbol{z}_{T-1}|\boldsymbol{z}_T, \boldsymbol{y}_{1:T})p(\boldsymbol{z}_{T-2}|\boldsymbol{z}_{T-1}, \cancel{\boldsymbol{z}_T}, \boldsymbol{y}_{1:T}) \cdots p(\boldsymbol{z}_1|\boldsymbol{z}_2, \cancel{\boldsymbol{z}_{3:T}}, \boldsymbol{y}_{1:T}) \tag{8.58}$$

$$= p(\boldsymbol{z}_T|\boldsymbol{y}_{1:T}) \prod_{t=T-1}^{1} p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T}) \tag{8.59}$$

Thus at step t we sample $\boldsymbol{z}_t^s$ from $p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}^s, \boldsymbol{y}_{1:T})$ given in Equation (8.17).

8.3 Inference based on the Kalman filter

In this section, we discuss inference in SSMs where all the distributions are linear Gaussian. This is called a **linear Gaussian state space model** (**LG-SSM**) or a **linear dynamical system** (**LDS**). We discuss such models in detail in Section 29.6, but in brief they have the following form:

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{u}_t) = \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{z}_{t-1} + \mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t, \mathbf{Q}_t) \tag{8.60}$$

$$p(\boldsymbol{y}_t|\boldsymbol{z}_t, \boldsymbol{u}_t) = \mathcal{N}(\boldsymbol{y}_t|\mathbf{H}_t\boldsymbol{z}_t + \mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t, \mathbf{R}_t) \tag{8.61}$$

where $\boldsymbol{z}_t \in \mathbb{R}^{N_z}$ is the hidden state, $\boldsymbol{y}_t \in \mathbb{R}^{N_y}$ is the observation, and $\boldsymbol{u}_t \in \mathbb{R}^{N_u}$ is the input. (We have allowed the parameters to be time-varying, for later extensions that we will consider.) We often assume the means of the process noise and observation noise (i.e., the bias or offset terms) are zero,

so $\boldsymbol{b}_t = \mathbf{0}$ and $\boldsymbol{d}_t = \mathbf{0}$. In addition, we often have no inputs, so $\mathbf{B}_t = \mathbf{D}_t = \mathbf{0}$. In this case, the model simplifies to the following:[3]

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) = \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{z}_{t-1}, \mathbf{Q}_t) \tag{8.62}$$

$$p(\boldsymbol{y}_t|\boldsymbol{z}_t) = \mathcal{N}(\boldsymbol{y}_t|\mathbf{H}_t\boldsymbol{z}_t, \mathbf{R}_t) \tag{8.63}$$

See Figure 8.1 for the graphical model.

Note that an LG-SSM is just a special case of a Gaussian Bayes net (Section 4.2.3), so the entire joint distribution $p(\boldsymbol{y}_{1:T}, \boldsymbol{z}_{1:T}|\boldsymbol{u}_{1:T})$ is a large multivariate Gaussian with $N_y N_z T$ dimensions. However, it has a special structure that makes it computationally tractable to use, as we show below. In particular, we will discuss the **Kalman filter** and **Kalman smoother**, that can perform exact filtering and smoothing in $O(TN_z^3)$ time.

8.3.1 Examples

Before diving into the theory, we give some motivating examples.

8.3.1.1 Tracking and state estimation

A common application of LG-SSMs is for **tracking** objects, such as airplanes or animals, from noisy measurements, such as radar or cameras. For example, suppose we want to track an object moving in 2d. (We discuss this example in more detail in Section 29.7.1.) The hidden state $\boldsymbol{z}_t$ encodes the location, (x_{t1}, x_{t2}), and the velocity, $(\dot{x}_{t1}, \dot{x}_{t1})$, of the moving object. The observation $\boldsymbol{y}_t$ is a noisy version of the location. (The velocity is not observed but can be inferred from the change in location.) We assume that we obtain measurements with a sampling period of Δ. The new location is the old location plus Δ times the velocity, plus noise added to all terms:

$$\boldsymbol{z}_t = \underbrace{\begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{F}} \boldsymbol{z}_{t-1} + \boldsymbol{q}_t \tag{8.64}$$

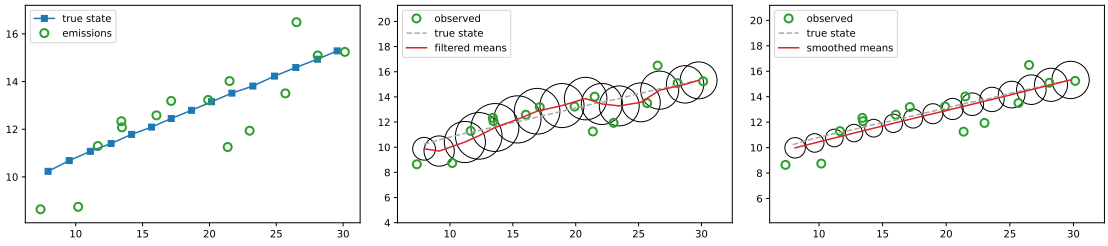
where $\boldsymbol{q}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t)$. The observation extracts the location and adds noise:

$$\boldsymbol{y}_t = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{\mathbf{H}} \boldsymbol{z}_t + \boldsymbol{r}_t \tag{8.65}$$

where $\boldsymbol{r}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t)$.

Our goal is to use this model to estimate the unknown location (and velocity) of the object given the noisy observations. In particular, in the filtering problem, we want to compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$ in a recursive fashion. Figure 8.8(b) illustrates filtering for the linear Gaussian SSM applied to the noisy tracking data in Figure 8.8(a) (shown by the green dots). The filtered estimates are computed

3. Our notation is similiar to [Sar13], except he writes $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) = \mathcal{N}(\boldsymbol{x}_k|\mathbf{A}_{k-1}\boldsymbol{x}_{k-1}, \mathbf{Q}_{k-1})$ instead of $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) = \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{z}_{t-1}, \mathbf{Q}_t)$, and $p(\boldsymbol{y}_k|\boldsymbol{x}_k) = \mathcal{N}(\boldsymbol{y}_k|\mathbf{H}_k\boldsymbol{x}_k, \mathbf{R}_k)$ instead of $p(\boldsymbol{y}_t|\boldsymbol{z}_t) = \mathcal{N}(\boldsymbol{y}_t|\mathbf{H}_t\boldsymbol{z}_t, \mathbf{R}_t)$.



(a) (b) (c)

Figure 8.8: Illustration of Kalman filtering and smoothing for a linear dynamical system. (a) Observations (green cirles) are generated by an object moving to the right (true location denoted by blue squares). (b) Results of online Kalman filtering. Red cross is the posterior mean, circles are 95% confidence ellipses derived from the posterior covariance. (c) Same as (b), but using offline Kalman smoothing. The MSE in the trajectory for filtering is 3.13, and for smoothing is 1.71. Generated by kf_ tracking.ipynb.

using the Kalman filter algorithm described in Section 8.3.2. The red line shows the posterior mean estimate of the location, and the black circles show the posterior covariance. We see that the estimated trajectory is less noisy than the raw data, since it incorporates prior knowledge about how the data was generated.

Another task of interest is the smoothing problem where we want to compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T})$ using an offline dataset. Figure 8.8(c) illustrates smoothing for the LG-SSM, implemented using the Kalman smoothing algorithm described in Section 8.3.3. We see that the resulting estimate is smoother, and that the posterior uncertainty is reduced (as visualized by the smaller confidence ellipses).

The disadvantage of the above smoothing method is that we have to wait until all the data has been observed before we start performing inference. **Fixed lag smoothing** is a useful compromise between online and offline estimation; it involves computing $p(\boldsymbol{z}_{t-\ell}|\boldsymbol{y}_{1:t})$, where $\ell > 0$ is called the lag. This gives better performance than filtering, but incurs a slight delay. By changing the size of the lag, we can trade off accuracy vs delay.

8.3.1.2 Online Bayesian linear regression (recursive least squares)

In Section 29.7.2 we discuss how to use the Kalman filter to recursively compute the exact posterior $p(\boldsymbol{w}|\mathcal{D}_{1:t})$ for a linear regression model in an online fashion. This is known as the recursive least squares algorithm. The basic idea is to treat the latent state to be the parameter values, $\boldsymbol{z}_t = \boldsymbol{w}$, and to define the non-stationary observation model as $p(\boldsymbol{y}_t|\boldsymbol{z}_t) = \mathcal{N}(y_t|\boldsymbol{x}_t^\mathsf{T}\boldsymbol{z}_t, \sigma^2)$, and the dynamics model as $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, 0\mathbf{I})$.

8.3.1.3 Time series forecasting

In Section 29.12, we discuss how to use Kalman filtering to perform time series forecasting.

8.3.2 The Kalman filter

The **Kalman Filter** (**KF**) is an algorithm for exact Bayesian filtering for linear Gaussian state space models. The resulting algorithm is the Gaussian analog of the HMM filter in Section 8.2.2. The belief state at time t is now given by $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$, where we use the notation $\boldsymbol{\mu}_{t|t'}$ and $\boldsymbol{\Sigma}_{t|t'}$ to represent the posterior mean and covariance given $\boldsymbol{y}_{1:t'}$.[4] Since everything is Gaussian, we can perform the prediction and update steps in closed form, as we explain below (see Section 8.3.2.4 for the derivation).

8.3.2.1 Predict step

The one-step-ahead prediction for the hidden state, also called the **time update step**, is given by the following:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.66}$$

$$\boldsymbol{\mu}_{t|t-1} = \mathbf{F}_t\boldsymbol{\mu}_{t-1|t-1} + \mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t \tag{8.67}$$

$$\boldsymbol{\Sigma}_{t|t-1} = \mathbf{F}_t\boldsymbol{\Sigma}_{t-1|t-1}\mathbf{F}_t^\mathsf{T} + \mathbf{Q}_t \tag{8.68}$$

8.3.2.2 Update step

The update step (also called the **measurement step**) can be computed using Bayes rule, as follows:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}, \boldsymbol{u}_{1:t}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \tag{8.69}$$

$$\boldsymbol{m}_t = \mathbf{H}_t\boldsymbol{\mu}_{t|t-1} + \mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t \tag{8.70}$$

$$\mathbf{S}_t = \mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t \tag{8.71}$$

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}\mathbf{S}_t^{-1} \tag{8.72}$$

$$\boldsymbol{e}_t = \boldsymbol{y}_t - \boldsymbol{m}_t \tag{8.73}$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t\boldsymbol{e}_t \tag{8.74}$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1} \tag{8.75}$$

$$= \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\mathsf{T} \tag{8.76}$$

where $\boldsymbol{m}_t$ is the expected observation, $\boldsymbol{e}_t$ is the **residual error** or **innovation term**, and $\mathbf{K}_t$ is the **Kalman gain matrix**.

Note that, by using the matrix inversion lemma, the Kalman gain matrix can also be written as

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}(\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t)^{-1} = (\boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{H}_t)^{-1}\mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1} \tag{8.77}$$

This is useful if $\mathbf{R}_t^{-1}$ is precomputed (e.g., if it is constant over time) and $N_y \gg N_z$.

4. We represent the mean and covariance of the filtered belief state by $\boldsymbol{\mu}_{t|t}$ and $\boldsymbol{\Sigma}_{t|t}$, but some authors use the notation $\boldsymbol{m}_t$ and $\mathbf{P}_t$ instead. We represent the mean and covariance of the smoothed belief state by $\boldsymbol{\mu}_{t|T}$ and $\boldsymbol{\Sigma}_{t|T}$, but some authors use the notation $\boldsymbol{m}_t^s$ and $\mathbf{P}_t^s$ instead. Finally, we represent the mean and covariance of the one-step-ahead posterior predictive distribution, $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})$, by $\boldsymbol{\mu}_{t|t-1}$ and $\boldsymbol{\Sigma}_{t|t-1}$, whereas some authors use $\boldsymbol{\mu}_t^-$ and $\boldsymbol{\Sigma}_t^-$ instead.

8.3.2.3 Posterior predictive

The one-step-ahead posterior predictive density for the observations can be computed as follows. (We ignore inputs and bias terms, for notational brevity.) First we compute the one-step-ahead predictive density for latent states:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1})d\boldsymbol{z}_{t-1} = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.78}$$

$$= \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{\mu}_{t-1|t-1}, \mathbf{F}_t\boldsymbol{\Sigma}_{t-1|t-1}\mathbf{F}_t^{\mathsf{T}} + \mathbf{Q}_t) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.79}$$

Then we convert this to a prediction about observations by marginalizing out $\boldsymbol{z}_t$:

$$p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{y}_t, \boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})d\boldsymbol{z}_t = \int p(\boldsymbol{y}_t|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})d\boldsymbol{z}_t = \mathcal{N}(\boldsymbol{y}_t|\boldsymbol{m}_t, \mathbf{S}_t) \tag{8.80}$$

This can also be used to compute the log-likelihood of the observations: The normalization constant of the new posterior can be computed as follows:

$$\log p(\boldsymbol{y}_{1:T}) = \sum_{t=1}^{T} \log p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = \sum_{t=1}^{T} \log Z_t \tag{8.81}$$

where we define $p(\boldsymbol{y}_1|\boldsymbol{y}_0) = p(\boldsymbol{y}_1)$. This is just a sum of the log probabilities of the one-step-ahead measurement predictions, and is a measure of how "surprised" the model is at each step.

We can generalize the prediction step to predict observations K steps into the future by first forecasting K steps in latent space, and then "grounding" the final state into predicted observations. (This is in contrast to an RNN (Section 16.3.4), which requires generating observations at each step, in order to update future hidden states.)

8.3.2.4 Derivation

In this section we derive the Kalman filter equations, following [Sar13, p57]. The results are a straightforward application of the rules for manipulating linear Gaussian systems, discussed in Section 2.2.6.

First we derive the prediction step. From Equation (2.81), the joint predictive distribution for states is given by

$$p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1}) \tag{8.82}$$

$$= \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{z}_{t-1}, \mathbf{Q}_t)\mathcal{N}(\boldsymbol{z}_{t-1}|\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}) \tag{8.83}$$

$$= \mathcal{N}\left(\begin{pmatrix}\boldsymbol{z}_{t-1}\\ \boldsymbol{z}_t\end{pmatrix}|\boldsymbol{\mu}', \boldsymbol{\Sigma}'\right) \tag{8.84}$$

where

$$\boldsymbol{\mu}' = \begin{pmatrix}\boldsymbol{\mu}_{t-1|t-1}\\ \mathbf{F}_t\boldsymbol{\mu}_{t-1|t-1}\end{pmatrix}, \ \boldsymbol{\Sigma}' = \begin{pmatrix}\boldsymbol{\Sigma}_{t-1|t-1} & \boldsymbol{\Sigma}_{t-1|t-1}\mathbf{F}_t^{\mathsf{T}}\\ \mathbf{F}_t\boldsymbol{\Sigma}_{t-1|t-1} & \mathbf{F}_t\boldsymbol{\Sigma}_{t-1|t-1}\mathbf{F}_t^{\mathsf{T}} + \mathbf{Q}_t\end{pmatrix} \tag{8.85}$$

Hence the marginal predictive distribution for states is given by

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}_t\boldsymbol{\mu}_{t-1|t-1}, \mathbf{F}_t\boldsymbol{\Sigma}_{t-1|t-1}\mathbf{F}_t^{\mathsf{T}} + \mathbf{Q}_t) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.86}$$

Now we derive the the measurement update step. The joint distribution for state and observation is given by

$$p(\boldsymbol{z}_t, \boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = p(\boldsymbol{y}_t|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) \tag{8.87}$$

$$= \mathcal{N}(\boldsymbol{y}_t|\mathbf{H}_t\boldsymbol{z}_t, \mathbf{R}_t)\mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.88}$$

$$= \mathcal{N}\left(\begin{pmatrix}\boldsymbol{z}_t\\ \boldsymbol{y}_t\end{pmatrix}|\boldsymbol{\mu}'', \boldsymbol{\Sigma}''\right) \tag{8.89}$$

where

$$\boldsymbol{\mu}'' = \begin{pmatrix}\boldsymbol{\mu}_{t|t-1}\\ \mathbf{H}_t\boldsymbol{\mu}_{t|t-1}\end{pmatrix}, \ \boldsymbol{\Sigma}'' = \begin{pmatrix}\boldsymbol{\Sigma}_{t|t-1} & \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}\\ \mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1} & \mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}^{-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t\end{pmatrix} \tag{8.90}$$

Finally, we convert this joint into a conditional using Equation (2.39) as follows:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_t, \boldsymbol{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \tag{8.91}$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}(\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t)^{-1}[\boldsymbol{y}_t - \mathbf{H}_t\boldsymbol{\mu}_{t|t-1}] \tag{8.92}$$

$$= \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t[\boldsymbol{y}_t - \mathbf{H}_t\boldsymbol{\mu}_{t|t-1}] \tag{8.93}$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}(\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t)^{-1}\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1} \tag{8.94}$$

$$= \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t\mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1} \tag{8.95}$$

where

$$\mathbf{S}_t = \mathbf{H}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T} + \mathbf{R}_t \tag{8.96}$$

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}_t^\mathsf{T}\mathbf{S}_t^{-1} \tag{8.97}$$

8.3.2.5 Abstract formulation

We can represent the Kalman filter equations much more compactly by defining various functions that create and manipulate jointly Gaussian systems, as in Section 2.2.6. In particular, suppose $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}})$, and $p(\boldsymbol{y}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{y}|\mathbf{A}\boldsymbol{z}+\boldsymbol{b}, \boldsymbol{\Omega})$. Then the joint is given by $p(\boldsymbol{z}, \boldsymbol{y}) = \mathcal{N}(\boldsymbol{z}, \boldsymbol{y}|\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}})$, where $\tilde{\boldsymbol{\mu}} = (\breve{\boldsymbol{\mu}}, \boldsymbol{m})$ and $\tilde{\boldsymbol{\Sigma}} = (\breve{\boldsymbol{\Sigma}}, \mathbf{C}; \mathbf{C}^\mathsf{T}, \mathbf{S})$, where the terms $\boldsymbol{m}$, $\mathbf{S}$ and $\mathbf{C}$ are given by **LinGaussPredict** in Algorithm 7. From this, the posterior is given by $p(\boldsymbol{z}|\boldsymbol{y}) = \mathcal{N}(\boldsymbol{z}|\ \widehat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}})$, where the posterior parameters are given by **GaussCondition** in Algorithm 7.

We can now apply these functions to derive Kalman filtering as follows. In the prediction step, we compute

$$p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = \mathcal{N}\left(\begin{pmatrix}\boldsymbol{\mu}_{t-1|t-1}\\ \boldsymbol{\mu}_{t|t-1}\end{pmatrix}, \begin{pmatrix}\boldsymbol{\Sigma}_{t-1|t-1} & \boldsymbol{\Sigma}_{t-1,t|t-1}\\ \boldsymbol{\Sigma}_{t,t-1|t-1} & \boldsymbol{\Sigma}_{t|t-1}\end{pmatrix}\right) \tag{8.98}$$

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{\Sigma}_{t-1,t|t}) = \mathbf{LinGaussPredict}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{F}_t, \mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t, \mathbf{Q}_t) \tag{8.99}$$

from which we get the marginal distribution

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \tag{8.100}$$

Algorithm 7: Functions for a linear Gaussian system.

def **LinGaussPredict**$(\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}}, \mathbf{A}, \boldsymbol{b}, \boldsymbol{\Omega})$:
  $\boldsymbol{m} = \mathbf{A}\,\breve{\boldsymbol{\mu}} + \boldsymbol{b}$
  $\mathbf{S} = \boldsymbol{\Omega} + \mathbf{A}\,\breve{\boldsymbol{\Sigma}}\,\mathbf{A}^{\mathsf{T}}$
  $\mathbf{C} = \breve{\boldsymbol{\Sigma}}\,\mathbf{A}^{\mathsf{T}}$
  Return $(\boldsymbol{m}, \mathbf{S}, \mathbf{C})$
def **GaussCondition**$(\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}}, \boldsymbol{m}, \mathbf{S}, \mathbf{C}, \boldsymbol{y})$:
  $\mathbf{K} = \mathbf{C}\mathbf{S}^{-1}$
  $\widehat{\boldsymbol{\mu}} = \breve{\boldsymbol{\mu}} + \mathbf{K}(\boldsymbol{y} - \boldsymbol{m})$
  $\widehat{\boldsymbol{\Sigma}} = \breve{\boldsymbol{\Sigma}} - \mathbf{K}\mathbf{S}\mathbf{K}^{\mathsf{T}}$
  $\ell = \log \mathcal{N}(\boldsymbol{y} | \boldsymbol{m}, \mathbf{S})$
  Return $(\widehat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}}, \ell)$
def **LinGaussUpdate**$(\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}}, \mathbf{A}, \boldsymbol{b}, \boldsymbol{\Omega}, \boldsymbol{y})$:
  $(\boldsymbol{m}, \mathbf{S}, \mathbf{C}) =$ **LinGaussPredict**$(\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}}, \mathbf{A}, \boldsymbol{b}, \boldsymbol{\Omega})$
  $(\widehat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}}, \ell) =$ **GaussCondition**$(\breve{\boldsymbol{\mu}}, \breve{\boldsymbol{\Sigma}}, \boldsymbol{m}, \mathbf{S}, \mathbf{C}, \boldsymbol{y})$
  Return $(\widehat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}}, \ell)$

In the update step, we compute the joint distribution

$$p(\boldsymbol{z}_t, \boldsymbol{y}_t | \boldsymbol{y}_{1:t-1}) = \mathcal{N}\left(\begin{pmatrix}\boldsymbol{\mu}_{t|t-1} \\ \boldsymbol{m}_t\end{pmatrix}, \begin{pmatrix}\boldsymbol{\Sigma}_{t|t-1} & \mathbf{C}_t \\ \mathbf{C}_t^{\mathsf{T}} & \mathbf{S}_t\end{pmatrix}\right) \tag{8.101}$$

$$(\boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t) = \textbf{LinGaussPredict}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{H}_t, \mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t, \mathbf{R}_t) \tag{8.102}$$

We then condition this on the observations to get the posterior distribution

$$p(\boldsymbol{z}_t | \boldsymbol{y}_t, \boldsymbol{y}_{1:t-1}) = p(\boldsymbol{z}_t | \boldsymbol{y}_{1:t}) = \mathcal{N}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \tag{8.103}$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \textbf{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t, \boldsymbol{y}_t) \tag{8.104}$$

We can combine the two parts of the update operation by defining the helper function **LinGaussUpdate** in Algorithm 7. The overall KF algorithm is shown in Algorithm 8.

Algorithm 8: Kalman filter.

def KF$(\mathbf{F}_{1:T}, \mathbf{B}_{1:T}, \boldsymbol{b}_{1:T}, \mathbf{Q}_{1:T}, \mathbf{H}_{1:T}, \mathbf{D}_{1:T}, \boldsymbol{d}_{1:T}, \mathbf{R}_{1:T}, \boldsymbol{u}_{1:T}, \boldsymbol{y}_{1:T}, \boldsymbol{\mu}_{0|0}, \boldsymbol{\Sigma}_{0|0})$:
  foreach $t = 1 : T$ **do**
    $(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) =$ **LinGaussPredict**$(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{F}_t, \mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t, \mathbf{Q}_t)$
    $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) =$ **LinGaussUpdate**$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{H}_t, \mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t, \mathbf{R}_t)$
  Return $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})_{t=1}^T$, $\sum_{t=1}^T \ell_t$

8.3.2.6 Numerical issues

In practice, the Kalman filter can encounter numerical issues. One solution is to use the **information filter**, which recursively updates the natural parameters of the Gaussian, $\boldsymbol{\Lambda}_{t|t} = \boldsymbol{\Sigma}_{t|t}^{-1}$ and $\boldsymbol{\eta}_{t|t} = \boldsymbol{\Lambda}_t \boldsymbol{\mu}_{t|t}$, instead of the mean and covariance (see Section 8.3.4). Another solution is the **square root filter**, which works with the Cholesky or QR decomposition of $\boldsymbol{\Sigma}_{t|t}$, which is much more numerically stable than directly updating $\boldsymbol{\Sigma}_{t|t}$. These techniques can be combined to create the **square root information filter** (SRIF) [May79]. (According to [Bie06], the SRIF was developed in 1969 for use in JPL's Mariner 10 mission to Venus.) In [Tol22] they present an approach which uses QR decompositions instead of matrix inversions, which can also be more stable.

8.3.3 The Kalman (RTS) smoother

In Section 8.3.2, we described the Kalman filter, which sequentially computes $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$ for each t. This is useful for online inference problems, such as tracking. However, in an offline setting, we can wait until all the data has arrived, and then compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T})$. By conditioning on past and future data, our uncertainty will be significantly reduced. This is illustrated in Figure 8.8(c), where we see that the posterior covariance ellipsoids are smaller for the smoothed trajectory than for the filtered trajectory.

We now explain how to compute the smoothed estimates, using an algorithm called the **RTS Smoother** or **RTSS**, named after its inventors, Rauch, Tung and Striebel [RTS65]. It is also known as the **Kalman smoothing** algorithm. The algorithm is the linear-Gaussian analog to the forwards-filtering backwards-smoothing algorithm for HMMs in Section 8.2.3.

8.3.3.1 Algorithm

One can show (see Section 8.3.3.2) that one step of the backwards smoothing pass can be implemented as follows:

$$(\boldsymbol{\mu}_{t+1|t}, \boldsymbol{\Sigma}_{t+1|t}, \boldsymbol{\Sigma}_{t,t+1|t}) = \mathbf{LinGaussPredict}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \mathbf{F}_{t+1}, \mathbf{B}_{t+1}\boldsymbol{u}_{t+1} + \boldsymbol{b}_{t+1}, \mathbf{Q}_{t+1}) \tag{8.105}$$

$$(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) = \mathbf{GaussSmoothUpdate}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{\Sigma}_{t,t+1|t}, \boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T}) \tag{8.106}$$

where the **GaussSmoothUpdate** function is defined in Algorithm 9. This uses the **backwards Kalman gain matrix**:

$$\mathbf{G}_t = \boldsymbol{\Sigma}_{t,t+1|t}\boldsymbol{\Sigma}_{t+1|t}^{-1} = \boldsymbol{\Sigma}_{t|t}\mathbf{F}_{t+1}^{\mathsf{T}}\boldsymbol{\Sigma}_{t+1|t}^{-1} \tag{8.107}$$

The algorithm can be initialized from $\boldsymbol{\mu}_{T|T}$ and $\boldsymbol{\Sigma}_{T|T}$ from the Kalman filter.

8.3.3.2 Derivation

In this section, we derive the RTS smoother, following [Sar13, p136]. As in the derivation of the Kalman filter in Section 8.3.2.4, we make heavy use of the rules for manipulating linear Gaussian systems, discussed in Section 2.2.6.

Algorithm 9: Updating a Gaussian distribution given updated beliefs on its child.

1 def **GaussSmoothUpdate**$(\boldsymbol{\mu}_{t+1|t}, \boldsymbol{\Sigma}_{t+1|t}, \boldsymbol{\Sigma}_{t,t+1|t}, \boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T})$
2 $\mathbf{G}_t = \boldsymbol{\Sigma}_{t,t+1|t}\boldsymbol{\Sigma}_{t+1|t}^{-1}$
3 $\boldsymbol{\mu}_{t|T} = \boldsymbol{\mu}_{t|t} + \mathbf{G}_t(\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t})$
4 $\boldsymbol{\Sigma}_{t|T} = \boldsymbol{\Sigma}_{t|t} + \mathbf{G}_t(\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{t+1|t})\mathbf{G}_t^\mathsf{T}$
5 Return $(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T})$

The joint filtered distribution for two consecutive time slices is

$$p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}) = p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) = \mathcal{N}(\boldsymbol{z}_{t+1}|\mathbf{F}_t\boldsymbol{z}_t, \mathbf{Q}_t)\mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t}) \tag{8.108}$$

$$= \mathcal{N}\left(\begin{pmatrix}\boldsymbol{z}_t \\ \boldsymbol{z}_{t+1}\end{pmatrix}|\boldsymbol{m}_1, \mathbf{V}_1\right) \tag{8.109}$$

where

$$\boldsymbol{m}_1 = \begin{pmatrix}\boldsymbol{\mu}_{t|t-1} \\ \mathbf{F}_t\boldsymbol{\mu}_{t|t-1}\end{pmatrix}, \ \mathbf{V}_1 = \begin{pmatrix}\boldsymbol{\Sigma}_{t|t} & \boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T} \\ \mathbf{F}_t\boldsymbol{\Sigma}_{t|t} & \mathbf{F}_t\boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T} + \mathbf{Q}_t\end{pmatrix} \tag{8.110}$$

By the Markov property for the hidden states we have

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:t}) \tag{8.111}$$

and hence by conditioning the joint distribution $p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})$ on the future state we get

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{m}_2(\boldsymbol{z}_{t+1}), \mathbf{V}_2) \tag{8.112}$$

$$\boldsymbol{\Sigma}_{t+1|t} = \mathbf{F}_t\boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T} + \mathbf{Q}_{t+1} \tag{8.113}$$

$$\mathbf{G}_t = \boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T}\boldsymbol{\Sigma}_{t+1|t}^{-1} \tag{8.114}$$

$$\boldsymbol{m}_2(\boldsymbol{z}_{t+1}) = \boldsymbol{\mu}_{t|t-1} + \mathbf{G}_t(\boldsymbol{z}_{t+1} - \mathbf{F}_t\boldsymbol{\mu}_{t|t-1}) \tag{8.115}$$

$$\mathbf{V}_2 = \boldsymbol{\Sigma}_{t|t} - \mathbf{G}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{G}_t^\mathsf{T} \tag{8.116}$$

The joint smoothed distribution of two consecutive time slices is

$$p(\boldsymbol{z}_{t+1}, \boldsymbol{z}_t|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T}) \tag{8.117}$$

$$= \mathcal{N}(\boldsymbol{z}_{t+1}|\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T})\mathcal{N}(\boldsymbol{z}_t|\boldsymbol{m}_2(\boldsymbol{z}_{t+1}), \mathbf{V}_2) \tag{8.118}$$

$$= \mathcal{N}\left(\begin{pmatrix}\boldsymbol{z}_{t+1} \\ \boldsymbol{z}_t\end{pmatrix}|\boldsymbol{m}_3, \mathbf{V}_3\right) \tag{8.119}$$

where

$$\boldsymbol{m}_3 = \begin{pmatrix}\boldsymbol{\mu}_{t+1|T} \\ \boldsymbol{\mu}_{t|t-1} + \mathbf{G}_t(\boldsymbol{\mu}_{t+1} - \mathbf{F}_t\boldsymbol{\mu}_{t|t-1})\end{pmatrix} \ \mathbf{V}_3 = \begin{pmatrix}\boldsymbol{\Sigma}_{t+1|T} & \boldsymbol{\Sigma}_{t+1|T}\mathbf{G}_T^\mathsf{T} \\ \mathbf{G}_t\boldsymbol{\Sigma}_{t+1|T} & \mathbf{G}_t\boldsymbol{\Sigma}_{t+1|T}\mathbf{G}_t^\mathsf{T} + \mathbf{V}_2\end{pmatrix} \tag{8.120}$$

From this, we can extract the smoothed marginal

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) \tag{8.121}$$

$$\boldsymbol{\mu}_{t|T} = \boldsymbol{\mu}_{t|t-1} + \mathbf{G}_t(\boldsymbol{\mu}_{t+1} - \mathbf{F}_t\boldsymbol{\mu}_{t|t-1}) \tag{8.122}$$

$$\boldsymbol{\Sigma}_{t|T} = \mathbf{G}_t\boldsymbol{\Sigma}_{t+1|T}\mathbf{G}_t^\mathsf{T} + \mathbf{V}_2 = \boldsymbol{\Sigma}_{t|t} + \mathbf{G}_t(\boldsymbol{\Sigma}_{t+1|T} - \mathbf{F}_t\boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T} - \mathbf{Q}_{t+1})\mathbf{G}_t^\mathsf{T} \tag{8.123}$$

8.3.3.3 Two-filter smoothing

Note that the backwards pass of the Kalman smoother does not need access to the observations, $\boldsymbol{y}_{1:T}$, but does need access to the filtered belief states from the forwards pass, $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$. There is an alternative version of the algorithm, known as **two-filter smoothing** [FP69; Kit04], in which we compute the forwards pass as usual, and then separately compute backwards messages $p(\boldsymbol{y}_{t+1:T}|\boldsymbol{z}_t) \propto \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}^b_{t|t-1}, \boldsymbol{\Sigma}^b_{t|t})$, similar to the backwards filtering algorithm in HMMs (Section 8.2.4).

However, these backwards messages are are conditional likelihoods, not posteriors, which can cause numerical problems. For example, consider $t = T$; in this case, we need to set the initial covariance matrix to be $\boldsymbol{\Sigma}^b_T = \infty\mathbf{I}$, so that the backwards message has no effect on the filtered posterior (since there is no evidence beyond step T). This problem can be resolved by working in information form. An alternative approach is to generalize the two-filter smoothing equations to ensure the likelihoods are normalizable by multiplying them by artificial distributions [BDM10].

In general, the RTS smoother is preferred to the two-filter smoother, since it is more numerically stable, and it is easier to generalize it to the nonlinear case.

8.3.3.4 Time and space complexity

In general, the Kalman smoothing algorithm takes $O(N_y^3 + N_z^2 + N_yN_z)$ per step, where there are T steps. This can be slow when applied to long sequences. In [SGF21], they describe how to reduce this to $O(\log T)$ steps using a **parallel prefix scan** operator that can be run efficiently on GPUs. In addition, we can reduce the space from $O(T)$, to $O(\log T)$ using the same algorithm as in Section 8.2.6.

8.3.3.5 Forwards filtering backwards sampling

To draw posterior samples from the LG-SSM, we can leverage the following result (derived in [Sar13, p137]):

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t+1}, \boldsymbol{y}_{1:T}) = \mathcal{N}(\boldsymbol{z}_t|\tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t) \tag{8.124}$$

$$\tilde{\boldsymbol{\mu}}_t = \boldsymbol{\mu}_{t|t} + \mathbf{G}_t(\boldsymbol{z}_{t+1} - \mathbf{F}_t\boldsymbol{\mu}_{t|t}) \tag{8.125}$$

$$\tilde{\boldsymbol{\Sigma}}_t = \boldsymbol{\Sigma}_{t|t} - \mathbf{G}_t\boldsymbol{\Sigma}_{t+1|t}\mathbf{G}_t^\mathsf{T} = \boldsymbol{\Sigma}_{t|t} - \boldsymbol{\Sigma}_{t|t}\mathbf{F}_t^\mathsf{T}\boldsymbol{\Sigma}^{-1}_{t+1|t}\boldsymbol{\Sigma}_{t+1|t}\mathbf{G}_t^\mathsf{T} \tag{8.126}$$

$$= \boldsymbol{\Sigma}_{t|t}(\mathbf{I} - \mathbf{F}_t^\mathsf{T}\mathbf{G}_t^\mathsf{T}) \tag{8.127}$$

where $\mathbf{G}_t$ is the backwards Kalman gain defined in Equation (8.107).

8.3.4 Information form filtering and smoothing

This section was written by Giles Harper-Donnelly.

In this section, we derive the Kalman filter and smoother algorithms in information form. We will see that this is the "dual" of Kalman filtering / smoothing in moment form. In particular, while computing marginals in moment form is easy, computing conditionals is hard (requires a matrix inverse). Conversely, for information form, computing marginals is hard, but computing conditionals is easy.

8.3.4.1 Filtering: algorithm

The predict step has a similar structure to the update step in moment form. We start with the prior $p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t-1}) = \mathcal{N}_c(\boldsymbol{z}_{t-1}|\boldsymbol{\eta}_{t-1|t-1}, \boldsymbol{\Lambda}_{t-1|t-1})$ and then compute

$$
\begin{aligned}
p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) &= \mathcal{N}_c(\boldsymbol{z}_t|\boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}) && (8.128)\\
\mathbf{M}_t &= \boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{F}_t && (8.129)\\
\mathbf{J}_t &= \mathbf{Q}_t^{-1}\mathbf{F}_t\mathbf{M}_t^{-1} && (8.130)\\
\boldsymbol{\Lambda}_{t|t-1} &= \mathbf{Q}_t^{-1} - \mathbf{Q}_t^{-1}\mathbf{F}_t(\boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{F}_t)^{-1}\ \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1} && (8.131)\\
&= \mathbf{Q}_t^{-1} - \mathbf{J}_t\mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1} && (8.132)\\
&= \mathbf{Q}_t^{-1} - \mathbf{J}_t\mathbf{M}_t\mathbf{J}_t^\mathsf{T} && (8.133)\\
\boldsymbol{\eta}_{t|t-1} &= \mathbf{J}_t\boldsymbol{\eta}_{t-1|t-1} + \boldsymbol{\Lambda}_{t|t-1}(\mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t), && (8.134)
\end{aligned}
$$

where $\mathbf{J}_t$ is analogous to the Kalman gain matrix in moment form Equation (8.72). From the matrix inversion lemma Equation (2.54), we see that Equation (8.131) is the inverse of the predicted covariance $\boldsymbol{\Sigma}_{t|t-1}$ given in Equation (8.68).

The update step in information form is as follows:

$$
\begin{aligned}
p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}, \boldsymbol{u}_{1:t}) &= \mathcal{N}_c(\boldsymbol{z}_t|\boldsymbol{\eta}_{t|t}, \boldsymbol{\Lambda}_{t|t}) && (8.135)\\
\boldsymbol{\Lambda}_{t|t} &= \boldsymbol{\Lambda}_{t|t-1} + \mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{H}_t && (8.136)\\
\boldsymbol{\eta}_{t|t} &= \boldsymbol{\eta}_{t|t-1} + \mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1}(\boldsymbol{y}_t - \mathbf{D}_t\boldsymbol{u}_t - \boldsymbol{d}_t). && (8.137)
\end{aligned}
$$

8.3.4.2 Filtering: derivation

For the predict step, we first derive the joint distribution over hidden states at $t, t-1$:

$$
\begin{aligned}
p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) &= p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{u}_t)p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t-1}) && (8.138)\\
&= \mathcal{N}_c(\boldsymbol{z}_t, |\mathbf{Q}_t^{-1}(\mathbf{F}_t\boldsymbol{z}_{t-1} + \mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t), \mathbf{Q}_t^{-1}) && (8.139)\\
&\times \mathcal{N}_c(\boldsymbol{z}_{t-1}, |\boldsymbol{\eta}_{t-1|t-1}, \boldsymbol{\Lambda}_{t-1|t-1}) && (8.140)\\
&= \mathcal{N}_c(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{\eta}_{t-1,t|t}, \boldsymbol{\Lambda}_{t-1,t|t}) && (8.141)
\end{aligned}
$$

where

$$\boldsymbol{\eta}_{t-1,t|t-1} = \begin{pmatrix} \boldsymbol{\eta}_{t-1|t-1} - \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}(\mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t) \\ \mathbf{Q}_t^{-1}(\mathbf{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t) \end{pmatrix} \tag{8.142}$$

$$\boldsymbol{\Lambda}_{t-1,t|t-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{F}_t & -\mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1} \\ -\mathbf{Q}_t^{-1}\mathbf{F}_t & \mathbf{Q}_t^{-1} \end{pmatrix} \tag{8.143}$$

The information form predicted parameters $\boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}$ can then be derived using the marginalisation formulae in Section 2.2.5.4.

For the update step, we start with the joint distribution over the hidden state and the observation at t:

$$p(\boldsymbol{z}_t, \boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{y}_t|\boldsymbol{z}_t, \boldsymbol{u}_t)p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t-1}) \tag{8.144}$$

$$= \mathcal{N}_c(\boldsymbol{y}_t, |\mathbf{R}_t^{-1}(\mathbf{H}_t\boldsymbol{z}_t + \mathbf{D}\boldsymbol{u}_t + \boldsymbol{d}_t), \mathbf{R}_t^{-1})\mathcal{N}_c(\boldsymbol{z}_t|\boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}) \tag{8.145}$$

$$= \mathcal{N}_c(\boldsymbol{z}_t, \boldsymbol{y}|\boldsymbol{\eta}_{\mathrm{z,y}|t}, \boldsymbol{\Lambda}_{\mathrm{z,y}|t}) \tag{8.146}$$

where

$$\boldsymbol{\eta}_{\mathrm{z,y}|t} = \begin{pmatrix} \boldsymbol{\eta}_{t|t-1} - \mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1}(\mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t) \\ \mathbf{R}_t^{-1}(\mathbf{D}_t\boldsymbol{u}_t + \boldsymbol{d}_t) \end{pmatrix} \tag{8.147}$$

$$\boldsymbol{\Lambda}_{\mathrm{z,y}|t} = \begin{pmatrix} \boldsymbol{\Lambda}_{t|t-1} + \mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{H}_t & -\mathbf{H}_t^\mathsf{T}\mathbf{R}_t^{-1} \\ -\mathbf{R}_t^{-1}\mathbf{H}_t & \mathbf{R}_t^{-1} \end{pmatrix} \tag{8.148}$$

The information form filtered parameters $\boldsymbol{\eta}_{t|t}, \boldsymbol{\Lambda}_{t|t}$ are then derived using the conditional formulae in 2.2.5.4.

8.3.4.3 Smoothing: algorithm

The smoothing equations are as follows:

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T}) = \mathcal{N}_c(\boldsymbol{z}_t|\boldsymbol{\eta}_{t|T}, \boldsymbol{\Lambda}_{t|T}) \tag{8.149}$$

$$\mathbf{U}_t = \mathbf{Q}_t^{-1} + \boldsymbol{\Lambda}_{t+1|T} - \boldsymbol{\Lambda}_{t+1|t} \tag{8.150}$$

$$\mathbf{L}_t = \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{U}_t^{-1} \tag{8.151}$$

$$\boldsymbol{\Lambda}_{t|T} = \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{F}_t - \mathbf{L}_t\mathbf{Q}_t^{-1}\mathbf{F} \tag{8.152}$$

$$= \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_t^\mathsf{T}\mathbf{Q}_t^{-1}\mathbf{F}_t - \mathbf{L}_t\mathbf{U}_t\mathbf{L}_t^\mathsf{T} \tag{8.153}$$

$$\boldsymbol{\eta}_{t|T} = \boldsymbol{\eta}_{t|t} + \mathbf{L}_t(\boldsymbol{\eta}_{t+1|T} - \boldsymbol{\eta}_{t+1|t}). \tag{8.154}$$

The parameters $\boldsymbol{\eta}_{t|t}$ and $\boldsymbol{\Lambda}_{t|t}$ are the filtered values from Equations (8.137) and (8.136) respectively. Similarly, $\boldsymbol{\eta}_{t+1|t}$ and $\boldsymbol{\Lambda}_{t+1|t}$ are the predicted parameters from Equations (8.134) and (8.131). The matrix $\mathbf{L}_t$ is the information form analog to the backward Kalman gain matrix in Equation (8.107).

8.3.4.4 Smoothing: derivation

From the generic forwards-filtering backwards-smoothing equation, Equation (8.20), we have

$$p(\boldsymbol{z}_t|\boldsymbol{y}_{1:T}) = p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) \int \left[\frac{p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})}{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})} \right] d\boldsymbol{z}_{t+1} \tag{8.155}$$

$$= \int p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t}) \frac{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:T})}{p(\boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})} d\boldsymbol{z}_{t+1} \tag{8.156}$$

$$= \int \mathcal{N}_c(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{\eta}_{t,t+1|t}, \boldsymbol{\Lambda}_{t,t+1|t}) \frac{\mathcal{N}_c(\boldsymbol{z}_{t+1}|\boldsymbol{\eta}_{t+1|T}, \boldsymbol{\Lambda}_{t+1|T})}{\mathcal{N}_c(\boldsymbol{z}_{t+1}|\boldsymbol{\eta}_{t+1|t}, \boldsymbol{\Lambda}_{t+1|t})} d\boldsymbol{z}_{t+1} \tag{8.157}$$

$$= \int \mathcal{N}_c(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{\eta}_{t,t+1|T}, \boldsymbol{\Lambda}_{t,t+1|T}) d\boldsymbol{z}_{t+1}. \tag{8.158}$$

The parameters of the joint filtering predictive distribution, $p(\boldsymbol{z}_t, \boldsymbol{z}_{t+1}|\boldsymbol{y}_{1:t})$, take precisely the same form as those in the filtering derivation described in Section 8.3.4.2:

$$\boldsymbol{\eta}_{t,t+1|t} = \begin{pmatrix} \boldsymbol{\eta}_{t|t} \\ \mathbf{0} \end{pmatrix}, \quad \boldsymbol{\Lambda}_{t,t+1|t} = \begin{pmatrix} \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_{t+1}^\mathsf{T} \mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & -\mathbf{F}_{t+1}^\mathsf{T} \mathbf{Q}_{t+1}^{-1} \\ -\mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & \mathbf{Q}_{t+1}^{-1} \end{pmatrix}, \tag{8.159}$$

We can now update this potential function by subtracting out the filtered information and adding in the smoothing information, using the rules for manipulating Gaussian potentials described in Section 2.2.7:

$$\boldsymbol{\eta}_{t,t+1|T} = \boldsymbol{\eta}_{t,t+1|t} + \begin{pmatrix} \mathbf{0} \\ \boldsymbol{\eta}_{t+1|T} \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \boldsymbol{\eta}_{t+1|t} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\eta}_{t|t} \\ \boldsymbol{\eta}_{t+1|T} - \boldsymbol{\eta}_{t+1|t} \end{pmatrix}, \tag{8.160}$$

and

$$\boldsymbol{\Lambda}_{t,t+1|T} = \boldsymbol{\Lambda}_{t,t+1|t} + \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Lambda}_{t+1|T} \end{pmatrix} - \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Lambda}_{t+1|t} \end{pmatrix} \tag{8.161}$$

$$= \begin{pmatrix} \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_{t+1}^\mathsf{T} \mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & -\mathbf{F}_{t+1}^\mathsf{T} \mathbf{Q}_{t+1}^{-1} \\ -\mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & \mathbf{Q}_{t+1}^{-1} + \boldsymbol{\Lambda}_{t+1|T} - \boldsymbol{\Lambda}_{t+1|t} \end{pmatrix} \tag{8.162}$$

Applying the information form marginalisation formula Equation (2.46) leads to Equation (8.154) and Equation (8.152).

8.4 Inference for non-linear and/or non-Gaussian SSMs

In general state space models (Section 29.1), the transition and/or emission probability distributions can be arbitrary conditional distributions, and are not restricted to being discrete or Gaussian distributions. This makes inference much harder, because the likelihood is not conjugate to the prior, and the posterior may not have any convenient analytical form.

8.4.1 Inference based on discretization

To illustrate the problem, consider a simple 1d SSM with linear dynamics corrupted by additive Student noise:

$$z_t = z_{t-1} + \mathcal{T}_2(0, 1) \tag{8.163}$$

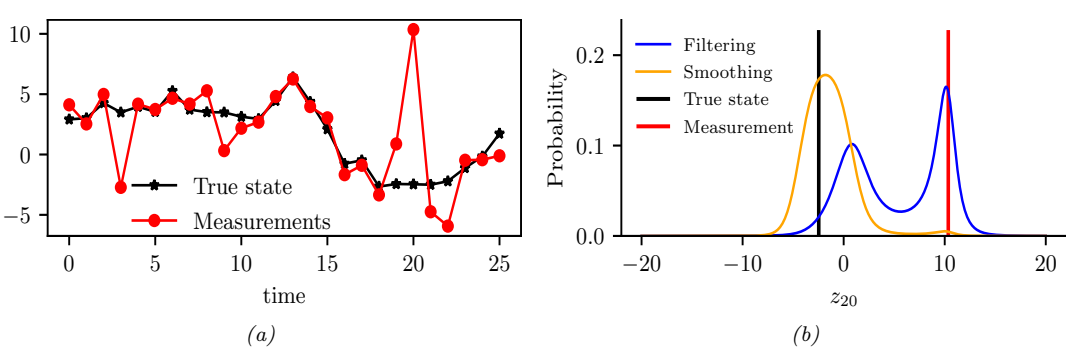


Figure 8.9: (a) Observations and true and estimated state. (b) Marginal distributions for time step $t = 20$. Generated by discretized_ssm_student.ipynb.

The observations are also linear, and are also corrupted by additive Student noise:

$$y_t = z_t + \mathcal{T}_2(0, 1) \tag{8.164}$$

This robust observation model is useful when there are potential outliers in the observed data, such as at time $t = 20$ in Figure 8.9a. (See also Section 8.6.2 for discussion of robust Kalman filters.)

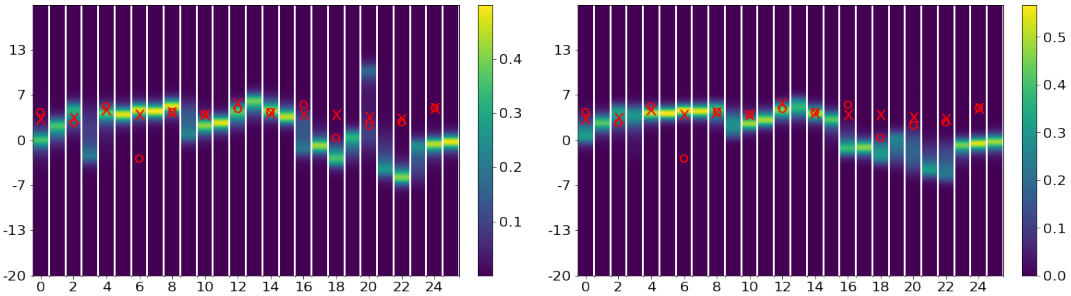
Unfortunately the use of a non-Gaussian likelihood means that the resulting posterior can become multimodal. In models with low-dimensional latent state spaces (e.g., 1-2d), we can always discretize the state space and apply the exact HMM filter and smoother from Section 8.2.4, as proposed in [RG17]. We show the results for filtering and smoothing in Figure 8.10a and in Figure 8.10b. We see that at $t = 20$, the filtering distribution, $p(z_t|\boldsymbol{y}_{1:20})$, is bimodal, with a mean that is quite far from the true state (see Figure 8.9b for a detailed plot). Such a multi-modal distribution can be approximated by a suitably fine discretization. Unfortunately, tackling the inference problem by discretizing the state space will not scale to higher dimensional problems, due to the curse of dimensionality. In particular, we know that the HMM filter takes $O(K^2)$ operations per time step, if there are K states. If we have N_z dimensions, each discretized into B bins, then we have $K = B^{N_z}$, so the approach becomes intractable beyond 1d.

8.4.2 Inference based on Gaussian approximations

In the sections below, we discuss various deterministic approximate inference schemes which make a Gaussian approximation to the posterior. Most of these algorithms assume the SSM has the following form:

$$\begin{aligned} \boldsymbol{z}_t &= \boldsymbol{f}(\boldsymbol{z}_{t-1}, \boldsymbol{u}_t) + \mathcal{N}(\boldsymbol{0}, \mathbf{Q}_t) \\ \boldsymbol{y}_t &= \boldsymbol{h}(\boldsymbol{z}_t, \boldsymbol{u}_t) + \mathcal{N}(\boldsymbol{0}, \mathbf{R}_t) \end{aligned} \tag{8.165}$$

where $\boldsymbol{z}_t \in \mathbb{R}^{N_z}$ is the hidden state, $\boldsymbol{y}_t \in \mathbb{R}^{N_y}$ is the observation, $\boldsymbol{u}_t \in \mathbb{R}^{N_u}$ are the optional inputs, $\boldsymbol{f} : \mathbb{R}^{N_z+N_u} \to \mathbb{R}^{N_z}$ is the dynamics model, and $\boldsymbol{h} : \mathbb{R}^{N_z+N_u} \to \mathbb{R}^{N_y}$ is the observation model. The dynamics and/or observations models can be nonlinear, but we assume the noise terms are Gaussian and additive. Our presentation is based on [Sar13, Ch. 5]; for more details, see e.g., [Sar13; Fan+17; Li+17e; Koy+10].)



(a) (b)

Figure 8.10: Discretized posterior of the latent state at each time step. Red cross is the true latent state. Red circle is observation. (a) Filtering. (b) Smoothing. Generated by discretized_ssm_student.ipynb.

8.5 Inference based on local linearization

In this section, we extend the Kalman filter and smoother to the case where the system dynamics and/or the observation model are nonlinear. (We continue to assume that the noise is additive Gaussian, as in Equation (8.165).) The basic idea is to linearize the dynamics and observation models about the previous state estimate using a first order Taylor series expansion, and then to apply the standard Kalman filter equations from Section 8.3.2. Intuitively we can think of this as approximating a stationary non-linear dynamical system with a non-stationary linear dynamical system. This approach is called the **extended Kalman filter** or **EKF**.

8.5.1 Taylor series expansion

Suppose $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x})$, where $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^m$ is a differentiable and invertible function. The pdf for $\boldsymbol{y}$ is given by

$$p(\boldsymbol{y}) = |\det \text{Jac}(\boldsymbol{g}^{-1})(\boldsymbol{y})|\, \mathcal{N}(\boldsymbol{g}^{-1}(\boldsymbol{y})|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{8.166}$$

In general this is intractable to compute, so we seek an approximation.

Suppose $\boldsymbol{x} = \boldsymbol{\mu} + \boldsymbol{\delta}$, where $\boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. Then we can form a first order Taylor series expansion of the function $\boldsymbol{g}$ as follows:

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{\mu} + \boldsymbol{\delta}) \approx \boldsymbol{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} \tag{8.167}$$

where $\mathbf{G}(\boldsymbol{\mu})$ is the Jacobian of $\boldsymbol{g}$ at $\boldsymbol{\mu}$:

$$[\mathbf{G}(\boldsymbol{\mu})]_{jj'} = \frac{\partial g_j(\boldsymbol{x})}{\partial x_{j'}}|_{\boldsymbol{x}=\boldsymbol{\mu}} \tag{8.168}$$

We now derive the induced Gaussian approximation to $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x})$. The mean is given by

$$\mathbb{E}[\boldsymbol{y}] \approx \mathbb{E}[\boldsymbol{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta}] = \boldsymbol{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\mathbb{E}[\boldsymbol{\delta}] = \boldsymbol{g}(\boldsymbol{\mu}) \tag{8.169}$$

The covariance is given by

$$\begin{aligned}
\text{Cov}\left[\boldsymbol{y}\right] &= \mathbb{E}\left[(\boldsymbol{g}(\boldsymbol{x}) - \mathbb{E}\left[\boldsymbol{g}(\boldsymbol{x})\right])(\boldsymbol{g}(\boldsymbol{x}) - \mathbb{E}\left[\boldsymbol{g}(\boldsymbol{x})\right])^{\mathsf{T}}\right] && (8.170)\\
&\approx \mathbb{E}\left[(\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{g}(\boldsymbol{\mu}))(\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{g}(\boldsymbol{\mu}))^{\mathsf{T}}\right] && (8.171)\\
&\approx \mathbb{E}\left[(\boldsymbol{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} - \boldsymbol{g}(\boldsymbol{\mu}))(\boldsymbol{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} - \boldsymbol{g}(\boldsymbol{\mu}))^{\mathsf{T}}\right] && (8.172)\\
&= \mathbb{E}\left[(\mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta})(\mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta})^{\mathsf{T}}\right] && (8.173)\\
&= \mathbf{G}(\boldsymbol{\mu})\mathbb{E}\left[\boldsymbol{\delta}\boldsymbol{\delta}^{\mathsf{T}}\right]\mathbf{G}(\boldsymbol{\mu})^{\mathsf{T}} && (8.174)\\
&= \mathbf{G}(\boldsymbol{\mu})\,\boldsymbol{\Sigma}\,\mathbf{G}(\boldsymbol{\mu})^{\mathsf{T}} && (8.175)
\end{aligned}$$

Algorithm 10: Linearized approximation to a joint Gaussian distribution.

```
def LinearizedPredict(μ, Σ, g, Ω) :
  m = g(μ)
  G = Jac(g)(μ)
  S = GΣG^T + Ω
  C = ΣG^T
  Return (μ, S, C)
```

When deriving the EKF, we need to compute the joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$ where

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}),\ \boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{q},\ \boldsymbol{q} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Omega}) \tag{8.176}$$

where $\boldsymbol{q}$ is independent of $\boldsymbol{x}$. We can compute this by defining the augmented function $\tilde{\boldsymbol{g}}(\boldsymbol{x}) = [\boldsymbol{x}, \boldsymbol{g}(\boldsymbol{x})]$ and following the procedure above. The resulting linear approximation to the joint is

$$\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}\boldsymbol{\mu}\\ \boldsymbol{m}\end{pmatrix}, \begin{pmatrix}\boldsymbol{\Sigma} & \mathbf{C}\\ \mathbf{C}^{\mathsf{T}} & \mathbf{S}\end{pmatrix}\right) \tag{8.177}$$

where the parameters are computed using Algorithm 10. We can then condition this joint Gaussian on the observed value $\boldsymbol{y}$ to get the posterior.

It is also possible to derive an approximation for the case of non-additive Gaussian noise, where $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{q})$. See [Sar13, Sec 5.1] for details.

8.5.2 The extended Kalman filter (EKF)

We now derive the extended Kalman filter for performing approximate inference in the model given by Equation (8.165). We first linearize the dynamics model around $\boldsymbol{\mu}_{t-1|t-1}$ to get an approximation to the one-step-ahead predictive distribution $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1})$. We then linearize the observation model around $\boldsymbol{\mu}_{t|t-1}$, and then performing a Gaussian update. (In Section 8.5.3, we consider linearizing around a different point that gives better accuracy.)

In more detail, we can write one step of the EKF algorithm as follows, where we use the notation from Section 8.3.2.5:

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \textbf{LinearizedPredict}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \boldsymbol{f}(\cdot, \boldsymbol{u}_t), \mathbf{Q}_t) \tag{8.178}$$

$$(\boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t) = \textbf{LinearizedPredict}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{h}(\cdot, \boldsymbol{u}_t), \mathbf{R}_t) \tag{8.179}$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \textbf{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t, \boldsymbol{y}_t) \tag{8.180}$$

The EKF is widely used because it is simple and relatively efficient. However, there are two cases when the EKF works poorly. The first is when the prior covariance is large. In this case, the prior distribution is broad, so we end up sending a lot of probability mass through different parts of the function that are far from $\boldsymbol{\mu}_{t-1|t-1}$, where the function has been linearized. The other setting where the EKF works poorly is when the function is highly nonlinear near the current mean (see Figure 8.12a).

A more accurate approach is to use a second-order Taylor series approximation, known as the **second order EKF**. The resulting updates can still be computed in closed form (see [Sar13, Sec 5.2] for details). Other more accurate extensions are discussed in Section 8.5.3 and Section 8.7.2.

8.5.3 Iterated EKF

Algorithm 11: Iterated extended Kalman filter.

```
def IEKF(f, Q, h, R, y_{1:T}, J):
foreach t = 1 : T do
    Predict step:
    (μ_{t|t-1}, Σ_{t|t-1}, −) = LinearizedPredict(μ_{t-1|t-1}, Σ_{t-1|t-1}, f(·, u_t), Q_t)
    Update step:
    μ_{t|t} = μ_{t|t-1}, Σ_{t|t} = Σ_{t|t-1}
    foreach j = 1 : J do
        (m_t, S_t, C_t) = LinearizedPredict(μ_{t|t}, Σ_{t|t}, h(·, u_t), R_t)
        (μ_{t|t}, Σ_{t|t}, ℓ_t) = GaussCondition(μ_{t|t-1}, Σ_{t|t-1}, m_t, S_t, C_t, y_t)
Return (μ_{t|t}, Σ_{t|t})_{t=1}^T
```

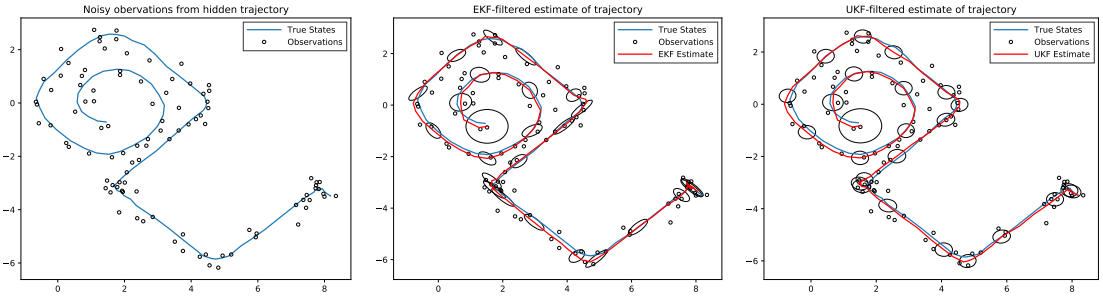
We can improve performance of the EKF by repeatedly re-linearizing the measurement model around the current posterior, $\boldsymbol{\mu}_{t|t}$, instead of $\boldsymbol{\mu}_{t|t-1}$; this is called the **iterated EKF** [BC93]. See Algorithm 11 for the pseudocode. Unfortunately the IEKF can diverge. A robust IEKF method, that uses line search to perform damped (partial) updates of $\boldsymbol{\mu}_{t|t}^j$, is presented in [SHA15].

8.5.4 The extended Kalman smoother (EKS)

We can extend the EKF to the offline smoothing case, resulting in the **extended Kalman smoother**, also called the **extended RTS smoother**. We just need to linearize around the filtered mean, and then apply the standard RTS update equations:

$$(\boldsymbol{\mu}_{t+1|t}, \boldsymbol{\Sigma}_{t+1|t}, \boldsymbol{\Sigma}_{t,t+1|t}) = \textbf{LinearizedPredict}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{f}(\cdot, \boldsymbol{u}_{t+1}), \mathbf{Q}_{t+1}) \tag{8.181}$$

$$(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) = \textbf{GaussSmoothUpdate}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{\Sigma}_{t,t+1|t}, \boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T}) \tag{8.182}$$



(a) (b) (c)

Figure 8.11: Illustration of filtering applied to a 2d nonlinear dynamical system. (a) True underlying state and observed data. (b) Extended Kalman filter estimate. Generated by ekf_spiral.ipynb. (c) Unscented Kalman filter estimate. Generated by ukf_spiral.ipynb.

See [Sar13, Sec 9.1] for more details.

For improved accuracy, you can use the **iterated EKS** which relinearizes the model at the previous MAP estimate (see Section 8.9.4).

8.5.5 Examples

In this section, we give some examples of EKF/EKS.

8.5.5.1 Tracking a point spiraling in 2d

In Section 8.3.1.1, we considered an example of state estimation and tracking of an object moving in 2d under a linear dynamics model with a linear observation model. However, motion and observation models are often nonlinear. For example, consider an object that is moving along a curved trajectory, such as this:

$$\boldsymbol{f}(\boldsymbol{z}) = (z_1 + \Delta \sin(z_2), z_2 + \Delta \cos(z_1)) \tag{8.183}$$

where Δ is the step size. For simplicity, we assume full visibility of the state vector (modulo observation noise), so $\boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{z}$.

Despite the simplicity of this model, exact inference is intractable. However, we can easily apply the EKF. The results are shown in Figure 8.11b.

8.5.5.2 Neural network training

In Section 17.5.2, we show how to use the EKF to perform online parameter inference for an MLP regression model.

8.6 Other variants of the Kalman filter

In this section, we briefly mention some other variants of Kalman filtering. For a more extensive review, see [Li+17e].

8.6.1 Ensemble Kalman filter

The **ensemble Kalman filter** (**EnKF**) is a technique developed in the geoscience (meteorology) community to perform approximate online inference in large nonlinear systems. In particular, it is mostly used for problems where the hidden state represents an unknown physical quantity (e.g., temperature and pressure) at each point on a spatial grid, and the measurements are sparse and spatially localized. Combining this information over space and time is called **data assimilation**.

The canonical reference is [Eve09], but a more accessible tutorial (using the same Bayesian signal processing approach we adopt in this chapter) is in [Rot+17].

The key idea is to represent the belief state $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$ by a finite number of samples $\mathbf{Z}_{t|t} = \{\boldsymbol{z}^s_{t|t} : s = 1 : N_s\}$, where each $\boldsymbol{z}^s_{t|t} \in \mathbb{R}^{N_z}$. In contrast to particle filtering (Section 13.2), the samples are updated in a manner that closely resembles the Kalman filter, so there is no importance sampling or resampling step. The downside is that the posterior does not converge to the true Bayesian posterior even as $N_s \to \infty$ [LGMT11], except in the linear-Gaussian case. However, sometimes the performance of EnKF can be better for small number of samples (although this depends of course on the PF proposal distribution).

The posterior mean and covariance can be derived from the ensemble of samples as follows:

$$\tilde{\boldsymbol{\mu}}_{t|t} = \frac{1}{N_s}\sum_{s=1}^{N_s} \boldsymbol{z}^s_{t|t} = \frac{1}{N_s}\mathbf{Z}_{t|t}\mathbf{1} \tag{8.184}$$

$$\tilde{\boldsymbol{\Sigma}}_{t|t} = \frac{1}{N_s - 1}\sum_{s=1}^{N_s}(\boldsymbol{z}^s_t - \tilde{\boldsymbol{\mu}}_{t|t})(\boldsymbol{z}^s_t - \tilde{\boldsymbol{\mu}}_{t|t})^\mathsf{T} = \frac{1}{N_s - 1}\tilde{\mathbf{Z}}_{t|t}\tilde{\mathbf{Z}}^\mathsf{T}_{t|t} \tag{8.185}$$

where $\tilde{\mathbf{Z}}_{t|t} = \mathbf{Z}_{t|t} - \tilde{\boldsymbol{\mu}}_{t|t}\mathbf{1}^\mathsf{T}$.

We update the samples as follows. For the time update, we first draw N_s system noise variables $\boldsymbol{q}^s_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t)$, and then we pass these, and the previous state estimate, through the dynamics model to get the one-step-ahead state predictions, $\boldsymbol{z}^s_{t|t-1} = \boldsymbol{f}(\boldsymbol{z}^s_{t-1|t-1}, \boldsymbol{q}^s_t)$, from which we get $\mathbf{Z}_{t|t-1} = \{\boldsymbol{z}^s_{t|t-1}\}$, which has size $N_z \times N_s$. Next we draw N_s observation noise variables $\boldsymbol{r}^s_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t)$, and use them to compute the one-step-ahead observation predictions, $\boldsymbol{y}^s_{t|t-1} = \boldsymbol{h}(\boldsymbol{z}^s_{t|t-1}, \boldsymbol{r}^s_t)$ and $\mathbf{Y}_{t|t-1} = \{\boldsymbol{y}^s_{t|t-1}\}$, which has size $N_y \times N_s$. Finally we compute the measurement update using

$$\mathbf{Z}_{t|t} = \mathbf{Z}_{t|t-1} + \tilde{\mathbf{K}}_t(\boldsymbol{y}_t\mathbf{1}^\mathsf{T} - \mathbf{Y}_{t|t-1}) \tag{8.186}$$

which is the analog of Equation (8.74).

We now discuss how to compute $\tilde{\mathbf{K}}_t$, which is the analog of the Kalman gain matrix in Equation (8.72). First note that we can write the exact Kalman gain matrix (in the linear-Gaussian case) as $\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{H}^\mathsf{T}\mathbf{S}_t^{-1} = \mathbf{C}_t\mathbf{S}_t^{-1}$, where $\mathbf{S}_t$ is the covariance of the measurements, and $\mathbf{C}_t$ is the cross-covariance between the state and output predictions. In the EnKF, we approximate $\mathbf{S}_t$ and $\mathbf{C}_t$ empirically as follows. First we compute the deviations from predictions:

$$\tilde{\mathbf{Z}}_{t|t-1} = \mathbf{Z}_{t|t-1}(\mathbf{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^\mathsf{T}), \quad \tilde{\mathbf{Y}}_{t|t-1} = \mathbf{Y}_{t|t-1}(\mathbf{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^\mathsf{T}) \tag{8.187}$$

Then we compute the sample covariance matrices

$$\tilde{\mathbf{S}}_t = \frac{1}{N_s - 1}\tilde{\mathbf{Y}}_{t|t-1}\tilde{\mathbf{Y}}^\mathsf{T}_{t|t-1}, \quad \tilde{\mathbf{C}}_t = \frac{1}{N_s - 1}\tilde{\mathbf{Z}}_{t|t-1}\tilde{\mathbf{Y}}^\mathsf{T}_{t|t-1} \tag{8.188}$$

Finally we compute

$$\tilde{\mathbf{K}}_t = \tilde{\mathbf{C}}_t \tilde{\mathbf{S}}_t^{-1} \tag{8.189}$$

We now compare the computational complexity to the KF algorithm. Recall that N_z fis the number of latent dimensions N_y is the number of observed dimensions, and N_s is the number of samples. We will assume $N_z > N_s > N_y$, as occurs in most geospatial problems. The EnKF time update takes $O(N_z^2 N_s)$ operations, and the measurement update takes $O(N_z N_y N_s)$, By contrast, in the KF, the time update takes $O(N_z^3)$ operations, and the measurement update takes $O(N_z^2 N_y)$. So we see that the EnKF is faster for high dimensional state spaces, because it uses a low-rank approximation to the posterior covariance.

Unfortunately, if N_s is too small, the EnKF can be become overconfident, and the filter can diverge. Various heuristics (e.g., covariance inflation) have been proposed to fix this. However, most of these methods are ad-hoc. A variety of more well-principled solutions have also been proposed, see e.g., [FK13b; Rei13].

8.6.2 Robust Kalman filters

In practice we often have noise that is non-Gaussian. A common example is when we have clutter, or outliers, in the observation model, or sudden changes in the process model. In this case, we might use the Laplace distribution [Ara+09] or the Student-t distribution [Ara10; RÖG13; Ara+17] as noise models.

[Hua+17b] proposes a variational Bayes (Section 10.2.3) approach, that allows the dynamical prior and the observation model to both be (linear) Student distributions, but where the posterior is approximated at each step using a Gaussian, conditional on the noise scale matrix, which is modeled using an inverse Wishart distribution. An extension of this, to handle mixture distributions, can be found in [Hua+19].

8.6.3 Dual EKF

In this section, we briefly discuss one approach to estimating the parameters of an SSM. In an offline setting, we can use EM, SGD or Bayesian inference to compute an approximation to $p(\boldsymbol{\theta}|\boldsymbol{y}_{1:T})$ (see Section 29.8). In the online setting, we want to compute $p(\boldsymbol{\theta}_t|\boldsymbol{y}_{1:t})$. We can do this by adding the parameters to the state space, possibly with an **artificial dynamics**, $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \epsilon\mathbf{I})$, and then performing joint inference of states and parameters. The latent variables at each step now contain the latent states, $\boldsymbol{z}_t$, and the latent parameters, $\boldsymbol{\theta}_t$. One approach to performing approximating inference in such a model is to use the **dual EKF**, in which one EKF performs state estimation and the other EKF performs parameter estimation [WN01].

8.7 Inference based on the unscented transform

In this section, we replace the local linearization of the model with a different approximation known as the **unscented transform**. When applied to Bayesian filtering, we get the **unscented Kalman filter** (**UKF**), since it is a version of the EKF that "doesn't stink" [JU97; JUDW00]. The key idea is this: instead of computing a linear approximation to the function and then passing a Gaussian through it, we instead pass a deterministically chosen set of points, known as **sigma points**, through

the function, and fit a Gaussian to the resulting transformed points; this is known as the unscented transform (see Section 8.7). Using the unscented transform for the transition and observation models gives the the overall method, which is also called a **sigma point filter** [VDMW03].

The main advantage of the UKF over the EKF is that it can be more accurate, and more stable. (The EKF can sometimes lead to large errors and divergence of the filter [IX00; VDMW03].) In addition, the UKF does not need to compute Jacobians of the observation and dynamics model, so it can be applied to non-differentiable models, or ones with hard constraints. However, the UKF can be slower, since it requires N_z evaluations of the dynamics and observation models. In addition, it has 3 hyper-parameters that need to be set.

8.7.1 The unscented transform

Algorithm 12: Computing sigma points using unscented transform.

1 def **SigmaPoints**$(\boldsymbol{\mu}, \boldsymbol{\Sigma}; \alpha, \beta, \kappa)$:
2 $n =$ dimensionality of $\boldsymbol{\mu}$
3 $\lambda = \alpha^2(n + \kappa) - n$
4 Compute a set of $2n + 1$ sigma points:
$\mathcal{X}_0 = \boldsymbol{\mu},\ \mathcal{X}_i = \boldsymbol{\mu} + \sqrt{n + \lambda}\,[\sqrt{\boldsymbol{\Sigma}}]_{:i},\ \mathcal{X}_{i+n} = \boldsymbol{\mu} - \sqrt{n + \lambda}\,[\sqrt{\boldsymbol{\Sigma}}]_{:i}$
5 Compute a set of $2n + 1$ weights for the mean and covariance:
$w_0^m = \frac{\lambda}{n+\lambda},\ w_0^c = \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta),\ w_i^m = w_i^c = \frac{1}{2(n+\lambda)}$
6 Return $(\mathcal{X}_{0:2n}, w^m_{0:2n}, w^c_{0:2n})$

Algorithm 13: Unscented approximation to a joint Gaussian distribution.

1 def **UnscentedPredict**$(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{g}, \boldsymbol{\Omega}; \alpha, \beta, \kappa)$:
2 $(\mathcal{X}_{0:2n}, w^m_{0:2n}, w^c_{0:2n}) =$ **SigmaPoints**$(\boldsymbol{\mu}, \boldsymbol{\Sigma}; \alpha, \beta, \kappa)$
3 $\mathcal{Y}_i = \boldsymbol{g}(\mathcal{X}_i),\ i = 0 : 2n$
4 $\boldsymbol{m} = \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i$
5 $\mathbf{S} = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \boldsymbol{\mu}_U)(\mathcal{Y}_i - \boldsymbol{\mu}_U)^\mathsf{T} + \boldsymbol{\Omega}$
6 $\mathbf{C} = \sum_{i=0}^{2n} w_i^c (\mathcal{X}_i - \boldsymbol{\mu})(\mathcal{Y}_i - \boldsymbol{\mu}_U)^\mathsf{T}$
7 Return $(\boldsymbol{m}, \mathbf{S}, \mathbf{C})$

Suppose we have two random variables $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x})$, where $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^m$. The unscented transform forms a Gaussian approximation to $p(\boldsymbol{y})$ using the following process. First we compute a set of $2n+1$ sigma points, $\mathcal{X}_i$, and corresponding weights, w_i^m and w_i^c, using Algorithm 12, for $i = 0 : 2n$. (The notation $\mathbf{M}_{:i}$ means the i'th column of matrix $\mathbf{M}$, $\sqrt{\boldsymbol{\Sigma}}$ is the matrix square root, so $\sqrt{\boldsymbol{\Sigma}}\sqrt{\boldsymbol{\Sigma}}^\mathsf{T} = \boldsymbol{\Sigma}$.) Next we propagate the sigma points through the nonlinear function to get the following $2n + 1$ outputs:

$$\mathcal{Y}_i = \boldsymbol{g}(\mathcal{X}_i),\ i = 0 : 2n \tag{8.190}$$

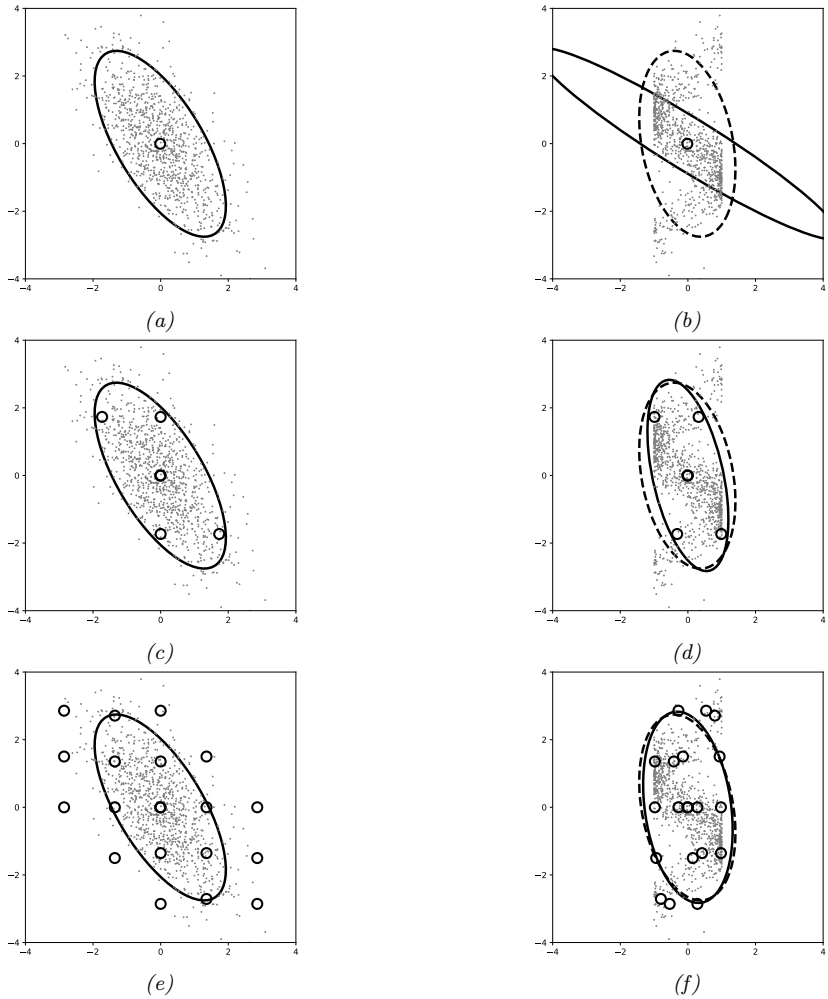


Figure 8.12: Illustration of different ways to approximate the distribution induced by a nonlinear transformation $f : \mathbb{R}^2 \to \mathbb{R}^2$. (a) Data from the source distribution, $\mathcal{D} = \{\boldsymbol{x}_i \sim p(\boldsymbol{x})\}$, with Gaussian approximation superimposed. (b) The dots show a Monte Carlo approximation to $p(f(\boldsymbol{x}))$ derived from $\mathcal{D}' = \{f(\boldsymbol{x}_i)\}$. The dotted ellipse is a Gaussian approximation to this target distribution, computed from the empirical moments. The solid ellipse is Taylor Transform. (c) Unscented sigma points. (d) Unscented transform. (e) Gauss-Hermite points (order 5). (f) GH transform. Adapted from Figures 5.3–5.4 of [Sar13]. Generated by gaussian_transforms.ipynb.

Finally we estimate the mean and covariance of the resulting set of points:

$$\mathbb{E}\left[\boldsymbol{g}(\boldsymbol{x})\right] \approx \boldsymbol{m} = \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i \tag{8.191}$$

$$\text{Cov}\left[\boldsymbol{g}(\boldsymbol{x})\right] \approx \mathbf{S} = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \boldsymbol{\mu}_U)(\mathcal{Y}_i - \boldsymbol{\mu}_U)^\mathsf{T} \tag{8.192}$$

Now suppose we want to approximate the joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{e}$, and $\boldsymbol{e} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Omega})$. By defining the augmented function $\tilde{\boldsymbol{g}}(\boldsymbol{x}) = (\boldsymbol{x}, \boldsymbol{g}(\boldsymbol{x}))$, and applying the above procedure (and adding extra noise), we get

$$\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{m} \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma} & \mathbf{C} \\ \mathbf{C}^\mathsf{T} & \mathbf{S} \end{pmatrix} \right) \tag{8.193}$$

where the parameters are computed using Algorithm 13.

The sigma points and their weights depend on three hyper-parameters, α, β and κ, which determine the spread of the sigma points around the mean. A typical recommended setting for these is $\alpha = 10^{-3}$, $\kappa = 1$, $\beta = 2$ [Bit16].

In Figure 8.12(a-b), we show the linearized Taylor transform discussed in Section 8.5.1 applued to a nonlinear function. In Figure 8.12(c-d), we show the corresponding unscented transform, which we can see is more accurate. In fact, the unscented transform is a third-order method in the sense that the mean of $\boldsymbol{y}$ is exact for polynomials up to order 3. However the covariance is only exact for linear functions (first order polynomials). We can compute even more accurate approximations using other forms of numerical integration. For example, we can use **Gauss-Hermite integration** of order p, which will be exact for polynomials of order up to $2p - 1$. See [Sar13, Sec 6.3] for details, and Figure 8.12(e-f) for an illustration. However, this comes at a price: the number of sigma points is now p^n, whereas it is only $2n + 1$ for the unscented transform.

8.7.2 The unscented Kalman filter (UKF)

The UKF applies the unscented transform twice to in Equation (8.165), once to approximate passing through the system model $\boldsymbol{f}$, and once to approximate passing through the measurement model $\boldsymbol{h}$. By analogy to Section 8.3.2.5, we can derive the UKF algorithm as follows:

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \textbf{UnscentedPredict}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \boldsymbol{f}(\cdot, \boldsymbol{u}_t), \mathbf{Q}_t) \tag{8.194}$$

$$(\boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t) = \textbf{UnscentedPredict}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{h}(\cdot, \boldsymbol{u}_t), \mathbf{R}_t) \tag{8.195}$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \textbf{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{m}_t, \mathbf{S}_t, \mathbf{C}_t, \boldsymbol{y}_t) \tag{8.196}$$

See [Sar13, p86] for more details.

8.7.3 The unscented Kalman smoother (UKS)

The **unscented Kalman smoother**, also called the **unscented RTS smoother** [Sar08], is a simple modification of the usual Kalman smoothing method, where we approximate the nonlinearity

by the unscented transform. By analogy to Section 8.3.3, the update is as follows:

$$(\boldsymbol{\mu}_{t+1|t}, \boldsymbol{\Sigma}_{t+1|t}, \boldsymbol{\Sigma}_{t,t+1|t}) = \mathbf{UnscentedPredict}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{f}(\cdot, \boldsymbol{u}_{t+1}), \mathbf{Q}_{t+1}) \tag{8.197}$$

$$(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) = \mathbf{GaussSmoothUpdate}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{\Sigma}_{t,t+1|t}, \boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T}) \tag{8.198}$$

See [Sar13, p148] for more details.

8.7.4 Examples

In this section we give some examples of the UKF.

8.7.4.1 Tracking a point spiraling in 2d

Let us revisit the 2d nonlinear tracking problem from Section 8.5.5.1. In Figure 8.11c, we see that the UKF algorithm (with $\alpha = 1$, $\beta = 0$, $\kappa = 2$) works well on this problem.

8.7.4.2 COVID-19 risk score estimation

During the COVID-19 pandemic in 2020, many governments created contact tracing apps, in which the distance between people was estimated (anonymously) by measuring bluetooth signal strength between mobile phones. This can be done using the UKF, as explained in [Lov+20]. Indeed, this approach formed the basis of the UK COVID-19 risk score estimation app [BCH20].[5]

8.8 Inference based on moment matching

In this section, we discuss a simple unified framework, known as the **general Gaussian filter** or **GGF** [IX00; Wu+06], which includes EKF, UKF, and various other algorithms. Our presentation is based on [Sar13, Ch. 6].

8.8.1 Gaussian moment matching

To explain the approach, we temporarily drop the time indices, and the conditioning on past information, and consider a single time step of inference. Furthermore, we will use the shorthand

$$\int_{\boldsymbol{x}} g(\boldsymbol{x}) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} g(x_1, \ldots, x_N) dx_1 \cdots dx_N \tag{8.199}$$

Let $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{\Omega})$ for some function $\boldsymbol{g}$. Let $p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})$ be the exact joint distribution. The best Gaussian approximation to the joint can be obtained by **moment matching**, i.e.,

$$q(\boldsymbol{x}, \boldsymbol{y}) = \mathcal{N}\left(\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} | \begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_M \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma} & \mathbf{C}_M \\ \mathbf{C}_M^{\mathsf{T}} & \mathbf{S}_M \end{pmatrix} \right) \tag{8.200}$$

5. Once the distance has been estimated, it needs to be combined with other signals, such as contact duration and infectiousness level of the index case, to estimate the risk of transmission. For an ML based approach to this problem, see [MKS21].

where

$$\boldsymbol{\mu}_M = \int_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x}) \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{8.201}$$

$$\mathbf{S}_M = \int_{\boldsymbol{x}} (\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\mu}_M)(\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\mu}_M)^{\mathsf{T}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) + \boldsymbol{\Omega} \tag{8.202}$$

$$\mathbf{C}_M = \int_{\boldsymbol{x}} (\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\mu}_M)^{\mathsf{T}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{8.203}$$

We can either compute these integrals by linearizing $\boldsymbol{g}$ and using closed-form expressions, or by using numerical integration, as we discuss in Section 8.8.3.

Once we have computed the joint $q(\boldsymbol{x}, \boldsymbol{y})$, we can compute the marginal $q(\boldsymbol{y})$ and the posterior $q(\boldsymbol{x}|\boldsymbol{y})$ using the usual equations for Gaussian distributions.

8.8.2 The general Gaussian filter

Let us now apply this method in the filtering context. We assume the prior has the form $p(\boldsymbol{z}_{t-1}|\boldsymbol{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1})$. In the prediction step, we compute a Gaussian approximation to $p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{y}_{1:t-1})$, from which we can compute the marginal $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t-1}) \approx \mathcal{N}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1})$. In the update step, we compute a Gaussian approximation to $p(\boldsymbol{z}_t, \boldsymbol{y}_t|\boldsymbol{y}_{1:t-1})$, from which we can compute the posterior $p(\boldsymbol{z}_t|\boldsymbol{y}_t, \boldsymbol{y}_{1:t-1}) \approx \mathcal{N}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$.

8.8.3 Implementation using numerical integration

To implement the above integrals in practice, we usually need some approximations. One approach is to linearize $\boldsymbol{g}$ around the predicted mean, as in Section 8.5.1. This gives rise to the EKF. However, this approach can lead to large errors and sometimes divergence of the filter [IX00; VDMW03].

A more stable method is to use **numerical integration**[6], which can be writen as

$$\int_{\boldsymbol{z}} f(\boldsymbol{z}) \approx \sum_{k=1}^{K} w^k f(\boldsymbol{z}^k) \tag{8.204}$$

for a suitable set of evaluation points $\boldsymbol{z}^k$ and weights w^k.

One approach is to use the unscented transform, as in Algorithm 13. We can denote this by

$$(\boldsymbol{\mu}_M, \mathbf{S}_M, \mathbf{C}_M) = \textbf{UnscentedPredict}(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{g}, \boldsymbol{\Omega}) \tag{8.205}$$

Using this inside the GGF is equivalent to the UKF in Section 8.7.2. Alternatively, we can use **spherical cubature integration**, which gives rise to the **cubature Kalman filter** or **CKF** [AH09]. This turns out (see [Sar13, p110]) to be a special case of the UKF, with $2n + 1$ sigma points, and hyper-parameters values of $\alpha = 1$ and $\beta = 0$ (with κ left free).

A more accurate approximation uses **Gauss-Hermite integration**, which allows the user to select more sigma points, as we discussed in Line 7. This gives rise to the **Gauss-Hermite Kalman filter** or **GHKF** [IX00], also known as the **quadrature Kalman filter** or **QKF** [AHE07].

6. One-dimensional integrals are called **quadratures**, and multi-dimensional integrals are called **cubatures**.

We can also approximate the integrals with Monte Carlo. Note, however, that this is not the same as particle filtering (Section 13.2), which approximates the conditional $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$ rather than the joint $p(\boldsymbol{z}_t, \boldsymbol{y}_t|\boldsymbol{y}_{1:t-1})$ (see Section 8.10.1 for discussion of this difference).

8.9 Inference based on statistical linearization

In this section, we discuss an approach to approximate inference which is based on generalized Gaussian filtering (Section 8.8), but which chooses the parameters of the linear approximation to minimize the mean squared error. This approach is therefore called **statistical linear regression** or **SLR** [LBS01; AHE07]. SLR is more accurate than Taylor linearization, which is only optimal at a single point, since it takes the entire distribution over the hidden quantities into account. We can show that the UKF is a special case of SLR. However, SLR is more general, and can be extended to handle non-Gaussian noise (see Section 8.9.5).

8.9.1 Statistical linear regression

In this section, we describe the SLR method, following [GFSS17]. We drop the time index for brevity. Suppose we have the prior $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and likelihood $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{\Omega})$. We want to approximate the joint, $p(\boldsymbol{x}, \boldsymbol{y})$, by a Gaussian, from which we can compute the marginal $p(\boldsymbol{y})$ and the posterior $p(\boldsymbol{x}|\boldsymbol{y})$ in the usual way.

We use the following enabling approximation:

$$\boldsymbol{g}(\boldsymbol{x}) \approx \mathbf{A}\boldsymbol{x} + \boldsymbol{b} + \boldsymbol{e} \tag{8.206}$$

where $\boldsymbol{e} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Omega})$ is the additional noise term introduced by the approximation. This will induce the joint distribution $q(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{\theta})$, where $\boldsymbol{\theta} = (\mathbf{A}, \boldsymbol{b}, \boldsymbol{\Omega})$. We choose the parameters of the approximate model so that the induced joint is as close as possible to the true joint:

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmin}_{\boldsymbol{\theta}} D_{\mathbb{KL}}\left(p(\boldsymbol{x}, \boldsymbol{y}) \parallel q(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{\theta})\right) = \operatorname*{argmax}_{\boldsymbol{\theta}} \int_{\boldsymbol{x},\boldsymbol{y}} p(\boldsymbol{x}, \boldsymbol{y}) \log q(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{\theta}) \tag{8.207}$$

This can be solved by making the moments of q match the moments of p (see Section 5.1.3.4). One can also show that this objective is equivalent to minimizing the mean squared error of the approximation

$$(\hat{\mathbf{A}}, \hat{\boldsymbol{b}}) = \operatorname*{argmin}_{\mathbf{A},\boldsymbol{b}} \mathbb{E}\left[(\boldsymbol{g}(\boldsymbol{x}) - \mathbf{A}\boldsymbol{x} - \boldsymbol{b})^{\mathsf{T}}(\boldsymbol{g}(\boldsymbol{x}) - \mathbf{A}\boldsymbol{x} - \boldsymbol{b})\right] \tag{8.208}$$

where $\hat{\boldsymbol{\Omega}}$ is the covariance of the resulting errors:

$$\hat{\boldsymbol{\Omega}} = \mathbb{E}\left[(\boldsymbol{g}(\boldsymbol{x}) - \hat{\mathbf{A}}\boldsymbol{x} - \hat{\boldsymbol{b}})(\boldsymbol{g}(\boldsymbol{x}) - \hat{\mathbf{A}}\boldsymbol{x} - \hat{\boldsymbol{b}})^{\mathsf{T}}\right] \tag{8.209}$$

One can show that the resulting solution is given by

$$\hat{\mathbf{A}} = \mathbf{C}^{\mathsf{T}}\boldsymbol{\Sigma}^{-1} \tag{8.210a}$$

$$\hat{\boldsymbol{b}} = \boldsymbol{m} - \hat{\mathbf{A}}\boldsymbol{\mu} \tag{8.210b}$$

$$\hat{\boldsymbol{\Omega}} = \mathbf{S} - \hat{\mathbf{A}}\boldsymbol{\Sigma}\hat{\mathbf{A}}^{\mathsf{T}} \tag{8.210c}$$