CHAPTER 9

Mean Field Inference

Bayesian inference is an important and useful tool, but it comes with a serious practical problem. It will help to have some notation. Write X for a set of observed values, H for the unknown (hidden) values of interest, and recall Bayes' rule has

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Normalizing constant}}.$$

The problem is that it is usually very difficult to form posterior distributions, because the normalizing constant is hard to evaluate for almost every model. This point is easily dodged in first courses. For MAP inference, we can ignore the normalizing constant. A careful choice of problem and of conjugate prior can make things look easy (or, at least, hide the real difficulty). But most of the time we cannot compute

$$P(X) = \int P(X|H)P(H)dX.$$

Either the integral is too hard, or – in the case of discrete models – the marginalization requires an unmanageable sum. In such cases, we must approximate.

Warning: The topics of this chapter allow a great deal of room for mathematical finicking, which I shall try to avoid. Generally, when I define something I'm going to leave out the information that it's only meaningful under some circumstances, etc. None of the background detail I'm eliding is difficult or significant for anything we do. Those who enjoy this sort of thing can supply the ifs ands and buts without trouble; those who don't won't miss them. I will usually just write an integral sign for marginalization, and I'll assume that, when the variables are discrete, everyone's willing to replace with a sum.

9.1 USEFUL BUT INTRACTABLE EXAMPLES

9.1.1 Boltzmann Machines

Here is a formal model we can use. A **Boltzmann machine** is a distribution model for a set of binary random variables. Assume we have N binary random variables U_i, which take the values 1 or -1. The values of these random variables are not observed (the true values of the pixels). These binary random variables are not independent. Instead, we will assume that some (but not all) pairs are coupled. We could draw this situation as a graph (Figure 9.1), where each node represents a U_i and each edge represents a coupling. The edges are weighted, so the coupling strengths vary from edge to edge.

Write $\mathcal{N}(i)$ for the set of random variables whose values are coupled to that

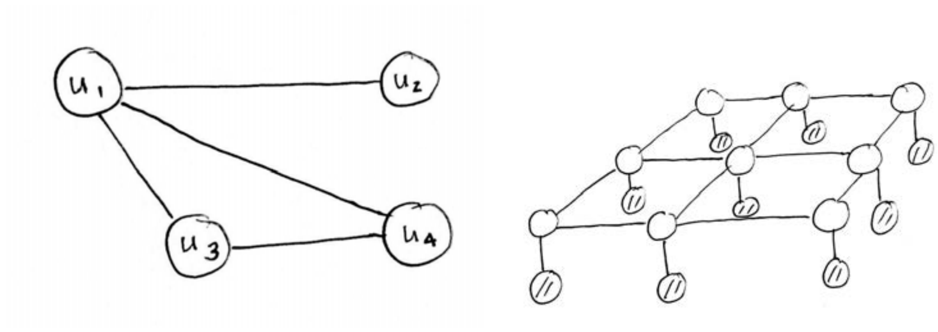


FIGURE 9.1: *On the* **left***, a simple Boltzmann machine. Each U_i has two possible states, so the whole thing has 16 states. Different choices of the constants coupling the U's along each edge lead to different probability distributions. On the* **right***, this Boltzmann machine adapted to denoising binary images. The shaded nodes represent the known pixel values (X_i in the text) and the open nodes represent the (unknown, and to be inferred) true pixel values H_i. Notice that pixels depend on their neighbors in the grid.*

of i – these are the neighbors of i in the graph. The joint probability model is

$$\log P(U|\theta) = \left[\sum_i \sum_{j \in \mathcal{N}(i)} \theta_{ij} U_i U_j \right] - \log Z(\theta) = -E(U|\theta) - \log Z(\theta).$$

Now $U_i U_j$ is 1 when U_i and U_j agree, and -1 otherwise (this is why we chose U_i to take values 1 or -1). The θ_{ij} are the edge weights; notice if $\theta_{ij} > 0$, the model generally prefers U_i and U_j to agree (as in, it will assign higher probability to states where they agree, unless other variables intervene), and if $\theta_{ij} < 0$, the model prefers they disagree.

Here $E(U|\theta)$ is sometimes referred to as the **energy** (notice the sign - higher energy corresponds to lower probability) and $Z(\theta)$ ensures that the model normalizes to 1, so that

$$Z(\theta) = \sum_{\text{all values of U}} \left[\exp\left(-E(U|\theta)\right) \right].$$

9.1.2 Denoising Binary Images with Boltzmann Machines

Here is a simple model for a binary image that has been corrupted by noise. At each pixel, we observe the corrupted value, which is binary. Hidden from us are the true values of each pixel. The observed value at each pixel is random, but depends only on the true value. This means that, for example, the value at a pixel can change, but the noise doesn't cause blocks of pixels to, say, shift left. This is a fairly good model for many kinds of transmission noise, scanning noise, and so on. The true value at each pixel is affected by the true value at each of its neighbors – a reasonable model, as image pixels tend to agree with their neighbors.

We can apply a Boltzmann machine. We split the U into two groups. One group represents the observed value at each pixel (I will use X_i, and the convention that i chooses the pixel), and the other represents the hidden value at each pixel (I will use H_i). Each observation is either 1 or -1. We arrange the graph so that the edges between the H_i form a grid, and there is a link between each X_i and its corresponding H_i (but no other - see Figure 9.1).

Assume we know good values for θ. We have

$$P(H|X,\theta) = \frac{\exp(-E(H,X|\theta))/Z(\theta)}{\Sigma_H \left[\exp(-E(H,X|\theta))/Z(\theta)\right]} = \frac{\exp\left(-E(H,X|\theta)\right)}{\Sigma_H \exp\left(-E(H,X|\theta)\right)}$$

so posterior inference doesn't require evaluating the normalizing constant. This isn't really good news. Posterior inference still requires a sum over an exponential number of values. Unless the underlying graph is special (a tree or a forest) or very small, posterior inference is intractable.

9.1.3 MAP Inference for Boltzmann Machines is Hard

You might think that focusing on MAP inference will solve this problem. Recall that MAP inference seeks the values of H to maximize $P(H|X,\theta)$ or equivalently, maximizing the log of this function. We seek

$$\underset{H}{\text{argmax}} \quad \log P(H|X,\theta) = (-E(H,X|\theta)) - \log\left[\Sigma_H \exp\left(-E(H,X|\theta)\right)\right]$$

but the second term is not a function of H, so we could avoid the intractable sum. This doesn't mean the problem is tractable. Some pencil and paper work will establish that there is some set of constants a_{ij} and b_j so that the solution is obtained by solving

$$\begin{array}{c}\underset{H}{\text{argmax}} \quad \left(\sum_{ij} a_{ij} h_i h_j\right) + \sum_j b_j h_j \\ \text{subject to } h_i \in \{-1,1\}\end{array} .$$

This is a combinatorial optimization problem with considerable potential for unpleasantness. How nasty it is depends on some details of the a_{ij}, but with the right choice of weights a_{ij}, the problem is **max-cut**, which is NP-complete.

9.1.4 A Discrete Markov Random Field

Boltzmann machines are a simple version of a much more complex device widely used in computer vision and other applications. In a Boltzmann machine, we took a graph and associated a binary random variable with each node and a coupling weight with each edge. This produced a probability distribution. We obtain a **Markov random field** by placing a random variable (doesn't have to be binary, or even discrete) at each node, and a coupling function (almost anything works) at each edge. Write U_i for the random variable at the i'th node, and $\theta(U_i, U_j)$ for the coupling function associated with the edge from i to j (the arguments tell you which function; you can have different functions on different edges).

We will ignore the possibility that the random variables are continuous. A **discrete Markov random field** has all U_i discrete random variables with a finite set of possible values. Write U_i for the random variable at each node, and $\theta(U_i, U_j)$ for the coupling function associated with the edge from i to j (the arguments tell you which function; you can have different functions on different edges). For a discrete Markov random field, we have

$$\log P(U|\theta) = \left[\sum_i \sum_{j\in\mathcal{N}(i)} \theta(U_i, U_j)\right] - \log Z(\theta).$$

It is usual – and a good idea – to think about the random variables as indicator functions, rather than values. So, for example, if there were three possible values at node i, we represent U_i with a 3D vector containing one indicator function for each value. One of the components must be one, and the other two must be zero. Vectors like this are sometimes know as **one-hot vectors**. The advantage of this representation is that it helps keep track of the fact that the *values* that each random variable can take are not really to the point; it's the *interaction* between assignments that matters. Another advantage is that we can easily keep track of the parameters that matter. I will adopt this convention in what follows.

I will write $\mathbf{u}_i$ for the random variable at location i represented as a vector. All but one of the components of this vector are zero, and the remaining component is 1. If there are $\#(U_i)$ possible values for U_i and $\#(U_j)$ possible values for U_j, we can represent $\theta(U_i, U_j)$ as a $\#(U_i) \times \#(U_j)$ table of values. I will write $\Theta^{(ij)}$ for the table representing $\theta(U_i, U_j)$, and $\theta^{(ij)}_{mn}$ for the m, n'th entry of that table. This entry is the value of $\theta(U_i, U_j)$ when U_i takes its m'th value and U_j takes its n'th value. I write $\Theta^{(ij)}$ for a matrix whose m, n'th component is $\theta^{(ij)}_{mn}$. In this notation, I write

$$\theta(U_i, U_j) = \mathbf{u}_i^T \Theta^{(ij)} \mathbf{u}_j.$$

All this does not simplify computation of the normalizing constant. We have

$$Z(\theta) = \sum_{\text{all values of } \mathbf{u}} \left[\exp\left(\sum_i \sum_{j\in\mathcal{N}(i)} \mathbf{u}_i^T \Theta^{(ij)} \mathbf{u}_j\right)\right].$$

Note that the collection of all values of $\mathbf{u}$ has rather nasty structure, and is very big – it consists of all possible one-hot vectors representing each U.

9.1.5 Denoising and Segmenting with Discrete MRF's

A simple denoising model for images that aren't binary is just like the binary denoising model. We now use a discrete MRF. We split the U into two groups, H and X. We observe a noisy image (the X values) and we wish to reconstruct the true pixel values (the H). For example, if we are dealing with grey level images with 256 different possible grey values at each pixel, then each H has 256 possible values. The graph is a grid for the H and one link from an X to the corresponding H (like Figure 9.1). Now we think about $P(H|X, \theta)$. As you would expect, the model is intractable – the normalizing constant can't be computed.

Worked example 9.1 *A simple discrete MRF for image denoising.*

Set up an MRF for grey level image denoising.

Solution: Construct a graph that is a grid. The grid represents the true value of each pixel, which we expect to be unknown. Now add an extra node for each grid element, and connect that node to the grid element. These nodes represent the observed value at each pixel. As before, we will separate the variables U into two sets, X for observed values and H for hidden values (Figure 9.1). In most grey level images, pixels take one of 256 ($= 2^8$) values. For the moment, we work with a grey level image, so each variable takes one of 256 values. There is no reason to believe that any one pixel behaves differently from any other pixel, so we expect the $\theta(H_i, H_j)$ not to depend on the pixel location; there'll be one copy of the same function at each grid edge. By far the most usual case has

$$\theta(H_i, H_j) = \left[\begin{array}{ll} 0 & \text{if } H_i = H_j \\ c & \text{otherwise} \end{array} \right.$$

where $c > 0$. Representing this function using one-hot vectors is straightforward. There is no reason to believe that the relationship between observed and hidden values depends on the pixel location. However, large differences between observed and hidden values should be more expensive than small differences. Write X_j for the observed value at node j, where j is the observed value node corresponding to H_i. We usually have

$$\theta(H_i, X_j) = (H_i - X_j)^2.$$

If we think of H_i as an indicator function, then this function can be represented as a vector of values; one of these values is picked out by the indicator. Notice there is a different vector at each H_i node (because there may be a different X_i at each).

Now write $\mathbf{h}_i$ for the hidden variable at location i represented as a vector, etc. Remember, all but one of the components of this vector are zero, and the remaining component is 1. The one-hot vector representing an observed value at location i is $\mathbf{x}_i$. I write $\Theta^{(ij)}$ for a matrix who's m, n'th component is $\theta^{(ij)}_{mn}$. In this notation, I write

$$\theta(H_i, H_j) = \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j$$

and

$$\theta(H_i, X_j) = \mathbf{h}_i^T \Theta^{(ij)} \mathbf{x}_j = \mathbf{h}_i^T \beta_i.$$

In turn, we have

$$\log p(H|X) = \left[\left(\sum_{ij} \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j \right) + \sum_i \mathbf{h}_i^T \beta_i \right] + \log Z.$$

Worked example 9.2 *Denoising MRF - II*

Write out $\Theta^{(ij)}$ for the $\theta(H_i, H_j)$ with the form given in example 9.1 using the one-hot vector notation.

Solution: This is more a check you have the notation. $c\mathcal{I}$ is the answer.

Worked example 9.3 *Denoising MRF - III*

Assume that we have $X_1 = 128$ and $\theta(H_i, X_j) = (H_i - X_j)^2$. What is β_1 using the one-hot vector notation? Assume pixels take values in the range $[0, 255]$.

Solution: Again, a check you have the notation. We have

$$\beta_1 = \begin{pmatrix} 128^2 & \text{first component} \\ \dots & \\ (i-128)^2 & i\text{'th component} \\ \dots & \\ 127^2 & \end{pmatrix}$$

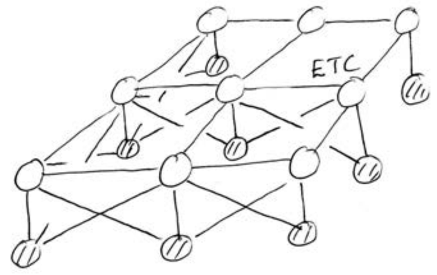


FIGURE 9.2: *The graph of an MRF adapted to image segmentation. The shaded nodes represent the known pixel values (X_i in the text) and the open nodes represent the (unknown, and to be inferred) labels H_i. A particular hidden node may depend on many pixels, because we will use all these pixel values to compute the cost of labelling that node in a particular way.*

Segmentation is another application that fits this recipe. We now want to break the image into a set of regions. Each region will have a label (eg "grass", "sky", "tree", etc.). The X_i are the observed values of each pixel value, and the H_i are the labels. In this case, the graph may have quite complex structure (eg figure 9.2). We must come up with a process that computes the cost of labelling a given pixel location in the image with a given label. Notice this process could look at many other pixel values in the image to come up with the label, but not at other labels. There are many possibilities. For example, we could build a logistic regression classifier that predicts the label at a pixel from image features around that pixel (if you don't know any image feature constructions, assume we use the pixel color; if you do, you can use anything that pleases you). We then model the cost of a having a particular label at a particular point as the negative log probability of the label under that model. We obtain the $\theta(H_i, H_j)$ by assuming that labels on neighboring pixels should agree with one another, as in the case of denoising.

9.1.6 MAP Inference in Discrete MRF's can be Hard

As you should suspect, focusing on MAP inference doesn't make the difficulty go away for discrete Markov random fields.

Worked example 9.4 *Useful facts about MRF's.*

Show that, using the notation of the text, we have: (a) for any i, $\mathbf{1}^T\mathbf{h}_i = 1$; (b) the MAP inference problem can be expressed as a quadratic program, with linear constraints, on discrete variables.

Solution: For (a) the equation is true because exactly one entry in $\mathbf{h}_i$ is 1, the others are zero. But (b) is more interesting. MAP inference is equivalent to maximizing $\log p(H|X)$. Recall $\log Z$ does not depend on the $\mathbf{h}$. We seek

$$\max_{\mathbf{h}_1,\ldots,\mathbf{h}_N} \left[\left(\sum_{ij} \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j \right) + \sum_i \mathbf{h}_i^T \beta_i \right] + \log Z$$

subject to very important constraints. We must have $\mathbf{1}^T\mathbf{h}_i = 1$ for all i. Furthermore, any component of any $\mathbf{h}_i$ must be either 0 or 1. So we have a quadratic program (because the cost function is quadratic in the variables), with linear constraints, on discrete variables.

Example 9.4 is a bit alarming, because it implies (correctly) that MAP inference in MRF's can be very hard. You should remember this. Gradient descent is no use here because the idea is meaningless. You can't take a gradient with respect to discrete variables. If you have the background, it's quite easy to prove by producing (eg from example 9.4) an MRF where inference is equivalent to max-cut, which is NP hard.

Worked example 9.5 *MAP inference for MRF's is a linear program*

Show that, using the notation of the text, the MAP inference for an MRF problem can be expressed as a linear program, with linear constraints, on discrete variables.

Solution: If you have two binary variables z_i and z_j both in $\{0,1\}$, then write $q_{ij} = z_i z_j$. We have that $q_{ij} \leq z_i$, $q_{ij} \leq z_j$, $q_{ij} \in \{0,1\}$, and $q_{ij} \geq z_i + z_j - 1$. You should check (a) these inequalities and (b) that q_{ij} is uniquely identified by these inequalities. Now notice that each $\mathbf{h}_i$ is just a bunch of binary variables, and the quadratic term $\mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j$ is linear in q_{ij}.

Example 9.5 is the start of an extremely rich vein of approximation mathematics, which we shall not mine. If you are of a deep mathematical bent, you can phrase everything in what follows in terms of approximate solutions of linear programs. For example, this makes it possible to identify MRF's for which MAP inference can be done in polynomial time; the family is more than just trees. We won't go there.

9.2 VARIATIONAL INFERENCE

We could just ignore intractable models, and stick to tractable models. This isn't a good idea, because intractable models are often quite natural. The discrete Markov random field model of an image is a fairly natural model. Image labels *should* depend on pixel values, and on neighboring labels. It is better to try and deal with the intractable model. One really successful strategy for doing so is to choose a tractable parametric family of probability models $Q(H;\theta)$, then adjust θ to find an element that is "close" in the right sense to $P(H|X)$. This process is known as **variational inference**.

9.2.1 The KL Divergence: Measuring the Closeness of Probability Distributions

Assume we have two probability distributions $P(X)$ and $Q(X)$. A measure of their similarity is the **KL-divergence** (or sometimes **Kullback-Leibler divergence**) written

$$\mathbb{D}(P \,\|\, Q) = \int P(X) \log \frac{P(X)}{Q(X)} dX$$

(you've clearly got to be careful about zeros in P and Q here). This likely strikes you as an odd measure of similarity, because it isn't symmetric. It is not the case that $\mathbb{D}(P \,\|\, Q)$ is the same as $\mathbb{D}(Q \,\|\, P)$, which means you have to watch your P's and Q's. Furthermore, some work will demonstrate that it does not satisfy the triangle inequality, so KL divergence lacks two of the three important properties of a metric.

KL divergence has some nice properties, however. First, we have

$$\mathbb{D}(P \,\|\, Q) \geq 0$$

with equality only if P and Q are equal almost everywhere (i.e. except on a set of measure zero).

Second, there is a suggestive relationship between KL divergence and maximum likelihood. Assume that X_i are IID samples from some *unknown* $P(X)$, and we wish to fit a parametric model $Q(X|\theta)$ to these samples. This is the usual situation we deal with when we fit a model. Now write $H(P)$ for the entropy of $P(X)$, defined by

$$H(P) = -\int P(X)\log P(X)dx = -\mathbb{E}_P[\log P].$$

The distribution P is unknown, and so is its entropy, but it is a constant. Now we can write

$$\mathbb{D}(P\,\|\,Q) = \mathbb{E}_P[\log P] - \mathbb{E}_P[\log Q]$$

Then

$$\begin{aligned}\mathcal{L}(\theta) = \sum_i \log Q(X_i|\theta) \approx \int P(X)\log Q(X|\theta)dX &= \mathbb{E}_{P(X)}[\log Q(X|\theta)] \\ &= -H(P) - \mathbb{D}(P\,\|\,Q)(\theta).\end{aligned}$$

Equivalently, we can write

$$\mathcal{L}(\theta) + \mathbb{D}(P\,\|\,Q)(\theta) = -H(P).$$

Recall P doesn't change (though it's unknown), so $H(P)$ is also constant (though unknown). This means that when $\mathcal{L}(\theta)$ goes up, $\mathbb{D}(P\,\|\,Q)(\theta)$ must go down. When $\mathcal{L}(\theta)$ is at a maximum, $\mathbb{D}(P\,\|\,Q)(\theta)$ must be at a minimum. All this means that, when you choose θ to maximize the likelihood of some dataset given θ for a parametric family of models, you are choosing the model in that family with smallest KL divergence from the (unknown) $P(X)$.

9.2.2 The Variational Free Energy

We have a $P(H|X)$ that is hard to work with (usually because we can't evaluate $P(X)$) and we want to obtain a $Q(H)$ that is "close to" $P(H|X)$. A good choice of "close to" is to require that

$$\mathbb{D}(Q(H)\,\|\,P(H|X))$$

is small. Expand the expression for KL divergence, to get

$$\begin{aligned}\mathbb{D}(Q(H)\,\|\,P(H|X)) &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H|X)] \\ &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)] + \mathbb{E}_Q[\log P(X)] \\ &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)] + \log P(X)\end{aligned}$$

which at first glance may look unpromising, because we can't evaluate $P(X)$. But $\log P(X)$ is fixed (although unknown). Now rearrange to get

$$\begin{aligned}\log P(X) &= \mathbb{D}(Q(H)\,\|\,P(H|X)) - (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)]) \\ &= \mathbb{D}(Q(H)\,\|\,P(H|X)) - \mathsf{E}_Q.\end{aligned}$$

Here

$$\mathsf{E}_Q = (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H, X)])$$

is referred to as the **variational free energy**. We can't evaluate $\mathbb{D}(Q(H) \,\|\, P(H|X))$. But, because $\log P(X)$ is fixed, when E_Q goes down, $\mathbb{D}(Q(H) \,\|\, P(H|X))$ must go down too. Furthermore, a minimum of E_Q will correspond to a minimum of $\mathbb{D}(Q(H) \,\|\, P(H|X))$. And we can evaluate E_Q.

We now have a strategy for building approximate $Q(H)$. We choose a family of approximating distributions. From that family, we obtain the $Q(H)$ that minimises E_Q (which will take some work). The result is the $Q(H)$ in the family that minimizes $\mathbb{D}(Q(H) \,\|\, P(H|X))$. We use that $Q(H)$ as our approximation to $P(H|X)$, and extract whatever information we want from $Q(H)$.

9.3 EXAMPLE: VARIATIONAL INFERENCE FOR BOLTZMANN MACHINES

We want to construct a $Q(H)$ that approximates the posterior for a Boltzmann machine. We will choose $Q(H)$ to have one factor for each hidden variable, so $Q(H) = q_1(H_1)q_2(H_2)\ldots q_N(H_N)$. We will then assume that all but one of the terms in Q are known, and adjust the remaining term. We will sweep through the terms doing this until nothing changes.

The i'th factor in Q is a probability distribution over the two possible values of H_i, which are 1 and -1. There is only one possible choice of distribution. Each q_i has one parameter $\pi_i = P(\{H_i = 1\})$. We have

$$q_i(H_i) = (\pi_i)^{\frac{(1+H_i)}{2}} (1 - \pi_i)^{\frac{(1-H_i)}{2}} .$$

Notice the trick; the power each term is raised to is either 1 or 0, and I have used this trick as a switch to turn on or off each term, depending on whether H_i is 1 or -1. So $q_i(1) = \pi_i$ and $q_i(-1) = (1 - \pi_i)$. This is a standard, and quite useful, trick. We wish to minimize the variational free energy, which is

$$\mathsf{E}_Q = (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H, X)]).$$

We look at the $\mathbb{E}_Q[\log Q]$ term first. We have

$$\begin{aligned}\mathbb{E}_Q[\log Q] &= \mathbb{E}_{q_1(H_1)\ldots q_N(H_N)}[\log q_1(H_1) + \ldots \log q_N(H_N)] \\ &= \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)] + \ldots \mathbb{E}_{q_N(H_N)}[\log q_N(H_N)]\end{aligned}$$

where we get the second step by noticing that

$$\mathbb{E}_{q_1(H_1)\ldots q_N(H_N)}[\log q_1(H_1)] = \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)]$$

(write out the expectations and check this if you're uncertain).

Now we need to deal with $\mathbb{E}_Q[\log P(H|X)]$. We have

$$\begin{aligned}\log p(H, X) &= -E(H, X) - \log Z \\ &= \sum_{i \in H} \sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij} H_i H_j + \sum_{i \in H} \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij} H_i X_j + K\end{aligned}$$

(where K doesn't depend on any H and is so of no interest). Assume all the q's are known except the i'th term. Write $Q_{\hat{i}}$ for the distribution obtained by omitting q_i from the product, so $Q_{\hat{1}} = q_2(H_2)q_3(H_3)\ldots q_N(H_N)$, etc. Notice that

$$\mathbb{E}_Q[\log P(H|X)] = \left(\begin{array}{c} q_i(-1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)] + \\ q_i(1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)] \end{array} \right).$$

This means that if we fix all the q terms *except* $q_i(H_i)$, we must choose q_i to minimize

$$\begin{array}{rc} q_i(-1)\log q_i(-1) + q_i(1)\log q_i(1) & - \\ q_i(-1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)] & + \\ q_i(1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)] & \end{array}$$

subject to the constraint that $q_i(1) + q_i(-1) = 1$. Introduce a Lagrange multiplier to deal with the constraint, differentiate and set to zero, and get

$$\begin{array}{rcl} q_i(1) & = & \frac{1}{c}\exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)]\right) \\ q_i(-1) & = & \frac{1}{c}\exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)]\right) \\ \text{where } c & = & \exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)]\right) + \\ & & \exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)]\right). \end{array}$$

In turn, this means we need to know $\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)]$, etc. only up to a constant. Equivalently, we need to compute only $\log q_i(H_i) + K$ for K some unknown constant (because $q_i(1) + q_i(-1) = 1$). Now we compute

$$\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N|X)].$$

This is equal to

$$\mathbb{E}_{Q_{\hat{i}}}\left[\sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij}(-1)H_j + \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij}(-1)X_j + \text{terms not containing } H_i\right]$$

which is the same as

$$\sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij}(-1)\mathbb{E}_{Q_{\hat{i}}}[H_j] + \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij}(-1)X_j + K$$

and this is the same as

$$\sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij}(-1)((\pi_j)(1) + (1 - \pi_j)(-1)) + \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij}(-1)X_j + K$$

and this is

$$\sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij}(-1)(2\pi_j - 1) + \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij}(-1)X_j + K.$$

If you thrash through the case for

$$\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)]$$

(which works the same) you will get

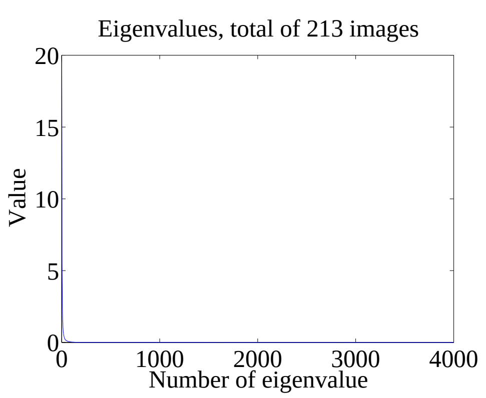
$$\begin{array}{rcl}\log q_i(1) & = & \mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N, X)] + K \\ & = & \sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j] + K \\ \text{and} & & \\ \log q_i(-1) & = & \mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N, X)] + K \\ & = & \sum_{j \in \mathcal{N}(i) \cap H} [-\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [-\theta_{ij} X_j] + K\end{array}$$

All this means that

$$\pi_i = \frac{e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j]\right)}}{e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j]\right)} + e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [-\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [-\theta_{ij} X_j]\right)}}.$$

After this blizzard of calculation, our inference algorithm is straightforward. We visit each hidden node in turn, set the associated π_i to the value of the expression above *assuming all the other π_j are fixed at their current values*, and repeat until convergence. We can test convergence by evaluating the variational free energy; an alternative is to check the size of the change in each π_j.

We can now do anything to $Q(H)$ that we would have done to $P(H|X)$. For example, we might compute the values of H that maximize $Q(H)$ for MAP inference. It is wise to limit ones ambition here, because $Q(H)$ is an approximation. It's straightforward to set up and describe, but it isn't particularly good. The main problem is that the variational distribution is unimodal. Furthermore, we chose a variational distribution by assuming that each H_i was independent of all others. This means that computing, say, covariances will likely lead to the wrong numbers (although it's easy — almost all are zero, and the remainder are easy). Obtaining an approximation by assuming that H_i is independent of all others is often called a **mean field method**.



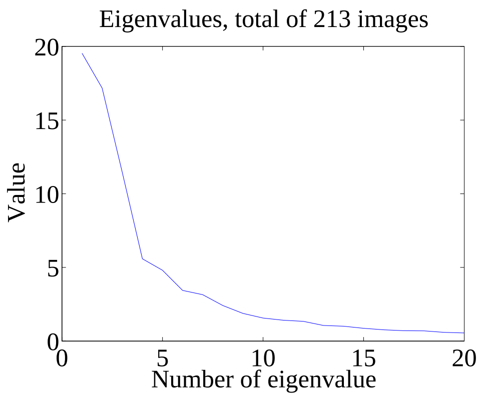


FIGURE 10.13: *On the* **left***,the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right***, a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

remember one adjusts the mean towards a data point by adding (or subtracting) some scale times the component. So the first few principal components have to do with the shape of the haircut; by the fourth, we are dealing with taller/shorter faces; then several components have to do with the height of the eyebrows, the shape of the chin, and the position of the mouth; and so on. These are all images of women who are not wearing spectacles. In face pictures taken from a wider set of models, moustaches, beards and spectacles all typically appear in the first couple of dozen principal components.

10.4 MULTI-DIMENSIONAL SCALING

One way to get insight into a dataset is to plot it. But choosing what to plot for a high dimensional dataset could be difficult. Assume we must plot the dataset in two dimensions (by far the most common choice). We wish to build a scatter plot in two dimensions — but where should we plot each data point? One natural requirement is that the points be laid out in two dimensions in a way that reflects how they sit in many dimensions. In particular, we would like points that are far apart in the high dimensional space to be far apart in the plot, and points that are close in the high dimensional space to be close in the plot.

10.4.1 Choosing Low D Points using High D Distances

We will plot the high dimensional point $\mathbf{x}_i$ at $\mathbf{v}_i$, which is a two-dimensional vector. Now the squared distance between points i and j in the high dimensional space is

$$D_{ij}^{(2)}(\mathbf{x}) = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)$$

(where the superscript is to remind you that this is a squared distance). We could build an $N \times N$ matrix of squared distances, which we write $\mathcal{D}^{(2)}(\mathbf{x})$. The i, j'th

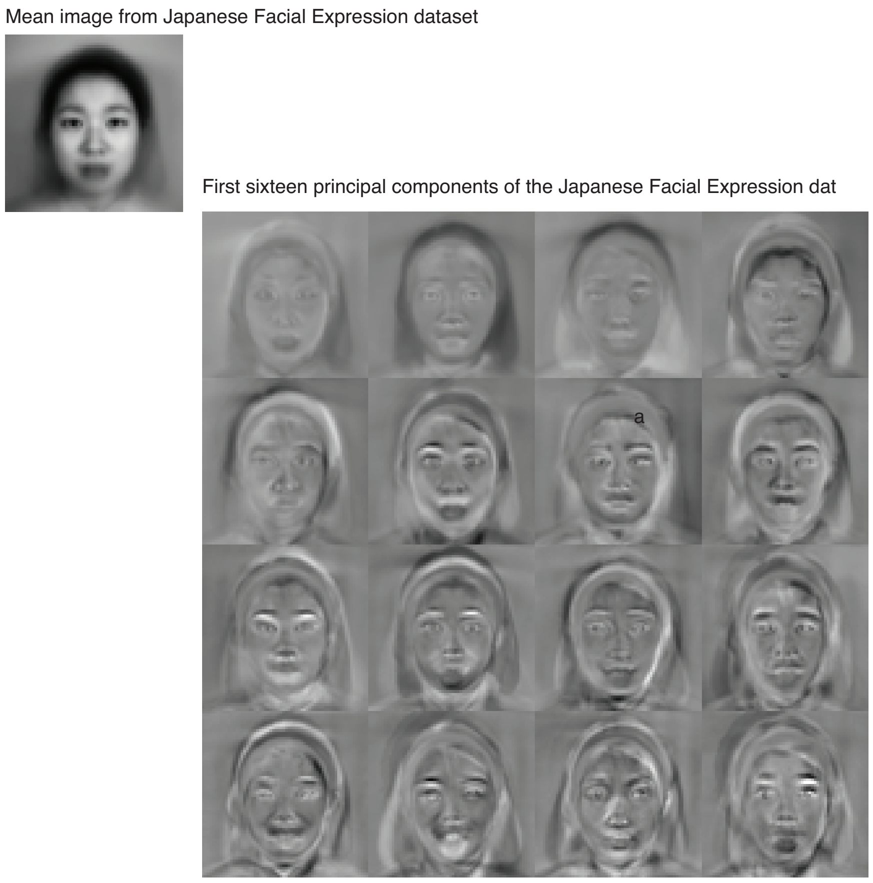

Mean image from Japanese Facial Expression dataset
First sixteen principal components of the Japanese Facial Expression dat
a

FIGURE 10.14: *The mean and first 16 principal components of the Japanese facial expression dataset.*

entry in this matrix is $D_{ij}^{(2)}(\mathbf{x})$, and the $\mathbf{x}$ argument means that the distances are between points in the high-dimensional space. Now we could choose the $\mathbf{v}_i$ to make

$$\sum_{ij} \left(D_{ij}^{(2)}(\mathbf{x}) - D_{ij}^{(2)}(\mathbf{v}) \right)^2$$

as small as possible. Doing so should mean that points that are far apart in the high dimensional space are far apart in the plot, and that points that are close in the high dimensional space are close in the plot.

In its current form, the expression is difficult to deal with, but we can refine it. Because translation does not change the distances between points, it cannot change either of the $\mathcal{D}^{(2)}$ matrices. So it is enough to solve the case when the mean of the points $\mathbf{x}_i$ is zero. We can assume that

$$\frac{1}{N} \sum_i \mathbf{x}_i = \mathbf{0}.$$

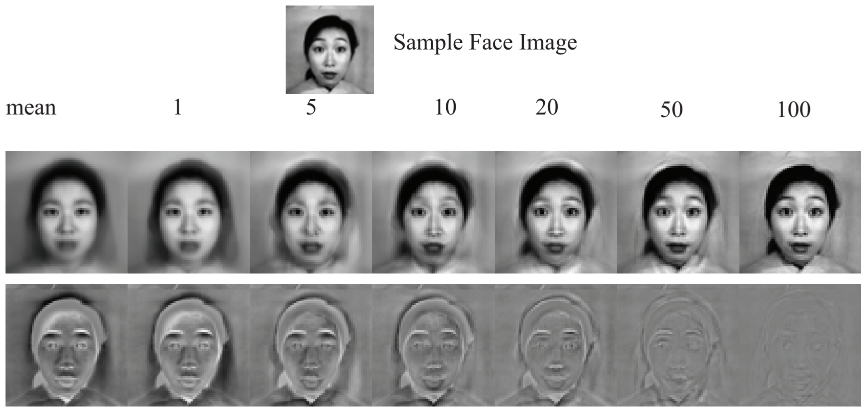


FIGURE 10.15: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*

Now write $\mathbf{1}$ for the n-dimensional vector containing all ones, and $\mathcal{I}$ for the identity matrix. Notice that

$$D_{ij}^{(2)} = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_i - 2\mathbf{x}_i \cdot \mathbf{x}_j + \mathbf{x}_j \cdot \mathbf{x}_j.$$

Now write

$$\mathcal{A} = \left[\mathcal{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right].$$

Using this expression, you can show that the matrix $\mathcal{M}$, defined below,

$$\mathcal{M}(\mathbf{x}) = -\frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}(\mathbf{x})\mathcal{A}^T$$

has i, jth entry $\mathbf{x}_i \cdot \mathbf{x}_j$ (exercises). I now argue that, to make $\mathcal{D}^{(2)}(\mathbf{v})$ is close to $\mathcal{D}^{(2)}(\mathbf{x})$, it is enough to make $\mathcal{M}(\mathbf{v})$ close to $\mathcal{M}(\mathbf{x})$. Proving this will take us out of our way unnecessarily, so I omit a proof.

We need some notation. Take the dataset of N d-dimensional column vectors $\mathbf{x}_i$, and form a matrix $\mathcal{X}$ by stacking the vectors, so

$$\mathcal{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \\ \mathbf{x}_N^T \end{bmatrix}.$$

In this notation, we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{X}\mathcal{X}^T.$$

Notice $\mathcal{M}(\mathbf{x})$ is symmetric, and it is positive semidefinite. It can't be positive definite, because the data is zero mean, so $\mathcal{M}(\mathbf{x})\mathbf{1} = 0$.

We can choose a set of $\mathbf{v}_i$ that makes $\mathcal{D}^{(2)}(\mathbf{v})$ close to $\mathcal{D}^{(2)}(\mathbf{x})$ quite easily.

To obtain a $\mathcal{M}(\mathbf{v})$ that is close to $\mathcal{M}(\mathbf{x})$, we need to choose $\mathcal{V} = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N]^T$ so that $\mathcal{V}\mathcal{V}^T$ is close to $\mathcal{M}(\mathbf{x})$. We are computing an approximate factorization of the matrix $\mathcal{M}(\mathbf{x})$.

10.4.2 Factoring a Dot-Product Matrix

We seek a set of k dimensional $\mathbf{v}$ that can be stacked into a matrix $\mathcal{V}$. This must produce a $\mathcal{M}(\mathbf{v}) = \mathcal{V}\mathcal{V}^T$ that must (a) be as close as possible to $\mathcal{M}(\mathbf{x})$ and (b) have rank at most k. It can't have rank larger than k because there must be some $\mathcal{V}$ which is $N \times k$ so that $\mathcal{M}(\mathbf{v}) = \mathcal{V}\mathcal{V}^T$. The rows of this $\mathcal{V}$ are our $\mathbf{v}_i^T$.

We can obtain the best factorization of $\mathcal{M}(\mathbf{x})$ from a diagonalization. Write write $\mathcal{U}$ for the matrix of eigenvectors of $\mathcal{M}(\mathbf{x})$ and Λ for the diagonal matrix of eigenvalues sorted in descending order, so we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{U}\Lambda\mathcal{U}^T$$

and write $\Lambda^{(1/2)}$ for the matrix of positive square roots of the eigenvalues. Now we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{U}\Lambda^{1/2}\Lambda^{1/2}\mathcal{U}^T = \left(\mathcal{U}\Lambda^{1/2}\right)\left(\mathcal{U}\Lambda^{1/2}\right)^T$$

which allows us to write

$$\mathcal{X} = \mathcal{U}\Lambda^{1/2}.$$

Now think about approximating $\mathcal{M}(\mathbf{x})$ by the matrix $\mathcal{M}(\mathbf{v})$. The error is a sum of squares of the entries,

$$\mathrm{err}(\mathcal{M}(\mathbf{x}), \mathcal{A}) = \sum_{ij}(m_{ij} - a_{ij})^2.$$

Because $\mathcal{U}$ is a rotation, it is straightforward to show that

$$\mathrm{err}(\mathcal{U}^T\mathcal{M}(\mathbf{x})\mathcal{U}, \mathcal{U}^T\mathcal{M}(\mathbf{v})\mathcal{U}) = \mathrm{err}(\mathcal{M}(\mathbf{x}), \mathcal{M}(\mathbf{v})).$$

But

$$\mathcal{U}^T\mathcal{M}(\mathbf{x})\mathcal{U} = \Lambda$$

which means that we could find $\mathcal{M}(\mathbf{v})$ from the best rank k approximation to Λ. This is obtained by setting all but the k largest entries of Λ to zero. Call the resulting matrix Λ_k. Then we have

$$\mathcal{M}(\mathbf{v}) = \mathcal{U}\Lambda_k\mathcal{U}$$

and

$$\mathcal{V} = \mathcal{U}\Lambda_k^{(1/2)}.$$

The first k columns of $\mathcal{V}$ are non-zero. We drop the remaining $N - k$ columns of zeros. The rows of the resulting matrix are our $\mathbf{v}_i$, and we can plot these. This method for constructing a plot is known as **principal coordinate analysis**.

This plot might not be perfect, because reducing the dimension of the data points should cause some distortions. In many cases, the distortions are tolerable. In other cases, we might need to use a more sophisticated scoring system that penalizes some kinds of distortion more strongly than others. There are many ways to do this; the general problem is known as **multidimensional scaling**.

Procedure: 10.3 *Principal Coordinate Analysis*

Assume we have a matrix $D^{(2)}$ consisting of the squared differences between each pair of N points. We do not need to know the points. We wish to compute a set of points in r dimensions, such that the distances between these points are as similar as possible to the distances in $D^{(2)}$.

- Form $\mathcal{A} = \left[\mathcal{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right]$.
- Form $\mathcal{W} = \frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}\mathcal{A}^T$.
- Form $\mathcal{U}$, Λ, such that $\mathcal{W}\mathcal{U} = \mathcal{U}\Lambda$ (these are the eigenvectors and eigenvalues of $\mathcal{W}$). Ensure that the entries of Λ are sorted in decreasing order.
- Choose r, the number of dimensions you wish to represent. Form Λ_r, the top left $r \times r$ block of Λ. Form $\Lambda_r^{(1/2)}$, whose entries are the positive square roots of Λ_r. Form $\mathcal{U}_r$, the matrix consisting of the first r columns of $\mathcal{U}$.

Then

$$\mathcal{V}^T = \Lambda_r^{(1/2)}\mathcal{U}_r^T = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$$

is the set of points to plot.

10.4.3 Example: Mapping with Multidimensional Scaling

Multidimensional scaling gets positions (the $\mathcal{V}$ of section 10.4.1) from distances (the $\mathcal{D}^{(2)}(\mathbf{x})$ of section 10.4.1). This means we can use the method to build maps from distances alone. I collected distance information from the web (I used http://www.distancefromto.net, but a google search on "city distances" yields a wide range of possible sources), then applied multidimensional scaling. I obtained distances between the South African provincial capitals, in kilometers. I then used principal coordinate analysis to find positions for each capital, and rotated, translated and scaled the resulting plot to check it against a real map (Figure 10.16).

One natural use of principal coordinate analysis is to see if one can spot any structure in a dataset. Does the dataset form a blob, or is it clumpy? This isn't a perfect test, but it's a good way to look and see if anything interesting is happening. In figure 10.17, I show a 3D plot of the spectral data, reduced to three dimensions using principal coordinate analysis. The plot is quite interesting. You should notice that the data points are spread out in 3D, but actually seem to lie on a complicated curved surface — they very clearly don't form a uniform blob. To me, the structure looks somewhat like a butterfly. I don't know why this occurs (perhaps the universe is doodling), but it certainly suggests that something worth investigating is going on. Perhaps the choice of samples that were measured is funny; perhaps the

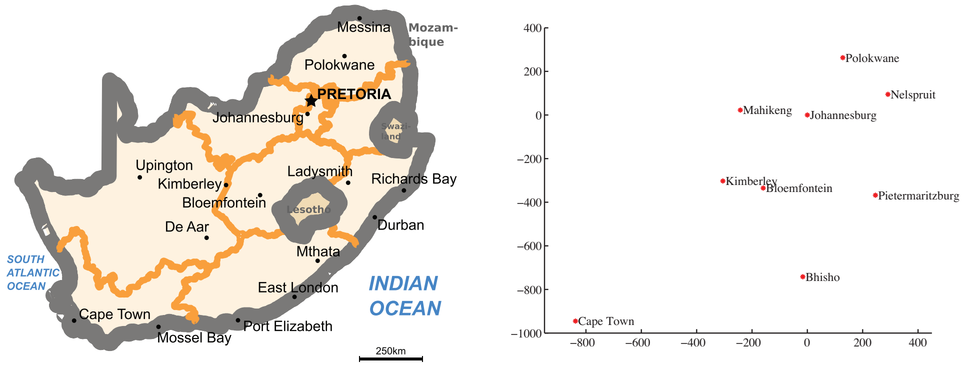


FIGURE 10.16: *On the* **left**, *a public domain map of South Africa, obtained from http://commons.wikimedia.org/wiki/File:Map_of_South_Africa.svg, and edited to remove surrounding countries. On the* **right**, *the locations of the cities inferred by multidimensional scaling, rotated, translated and scaled to allow a comparison to the map by eye. The map doesn't have all the provincial capitals on it, but it's easy to see that MDS has placed the ones that are there in the right places (use a piece of ruled tracing paper to check).*

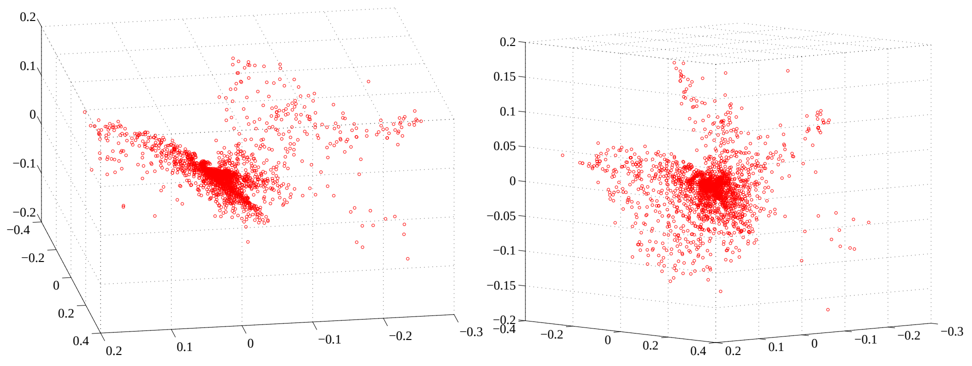


FIGURE 10.17: *Two views of the spectral data of section 10.3.1, plotted as a scatter plot by applying principal coordinate analysis to obtain a 3D set of points. Notice that the data spreads out in 3D, but seems to lie on some structure; it certainly isn't a single blob. This suggests that further investigation would be fruitful.*

measuring instrument doesn't make certain kinds of measurement; or perhaps there are physical processes that prevent the data from spreading out over the space.

Our algorithm has one really interesting property. In some cases, we do not actually know the datapoints as vectors. Instead, we *just* know distances between the datapoints. This happens often in the social sciences, but there are important cases in computer science as well. As a rather contrived example, one could survey people about breakfast foods (say, eggs, bacon, cereal, oatmeal, pancakes, toast, muffins, kippers and sausages for a total of 9 items). We ask each person to rate the similarity of each pair of distinct items on some scale. We advise people that similar

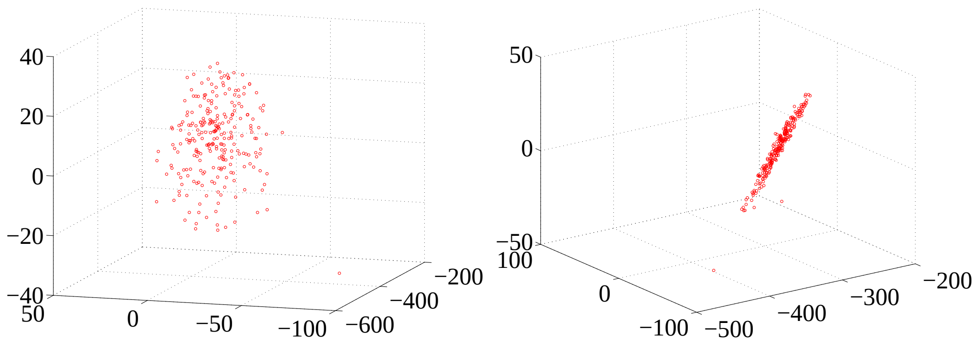


FIGURE 10.18: *Two views of a multidimensional scaling to three dimensions of the height-weight dataset. Notice how the data seems to lie in a flat structure in 3D, with one outlying data point. This means that the distances between data points can be (largely) explained by a 2D representation.*

items are ones where, if they were offered both, they would have no particular preference; but, for dissimilar items, they would have a strong preference for one over the other. The scale might be "very similar", "quite similar", "similar", "quite dissimilar", and "very dissimilar" (scales like this are often called **Likert scales**). We collect these similarities from many people for each pair of distinct items, and then average the similarity over all respondents. We compute distances from the similarities in a way that makes very similar items close and very dissimilar items distant. Now we have a table of distances between items, and can compute a $\mathcal{V}$ and produce a scatter plot. This plot is quite revealing, because items that most people think are easily substituted appear close together, and items that are hard to substitute are far apart. The neat trick here is that we did not start with a $\mathcal{X}$, but with just a set of distances; but we were able to associate a vector with "eggs", and produce a meaningful plot.

10.5 EXAMPLE: UNDERSTANDING HEIGHT AND WEIGHT

Recall the height-weight data set of section **??** (from http://www2.stetson.edu/~jrasp/data.htm; look for bodyfat.xls at that URL). This is, in fact, a 16-dimensional dataset. The entries are (in this order): *bodyfat; density; age; weight; height; adiposity; neck; chest; abdomen; hip; thigh; knee; ankle; biceps; forearm; wrist.* We know already that many of these entries are correlated, but it's hard to grasp a 16 dimensional dataset in one go. The first step is to investigate with a multidimensional scaling.

Figure **??** shows a multidimensional scaling of this dataset down to three dimensions. The dataset seems to lie on a (fairly) flat structure in 3D, meaning that inter-point distances are relatively well explained by a 2D representation. Two points seem to be special, and lie far away from the flat structure. The structure isn't perfectly flat, so there will be small errors in a 2D representation; but it's clear that a lot of dimensions are redundant. Figure 10.19 shows a 2D representation of these points. They form a blob that is stretched along one axis, and there is no sign

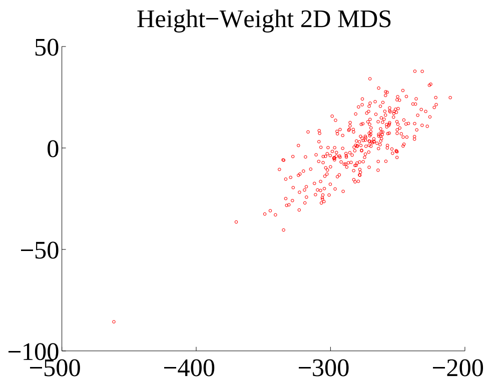


FIGURE 10.19: *A multidimensional scaling to two dimensions of the height-weight dataset. One data point is clearly special, and another looks pretty special. The data seems to form a blob, with one axis quite a lot more important than another.*

of multiple blobs. There's still at least one special point, which we shall ignore but might be worth investigating further. The distortions involved in squashing this dataset down to 2D seem to have made the second special point less obvious than it was in figure **??**.

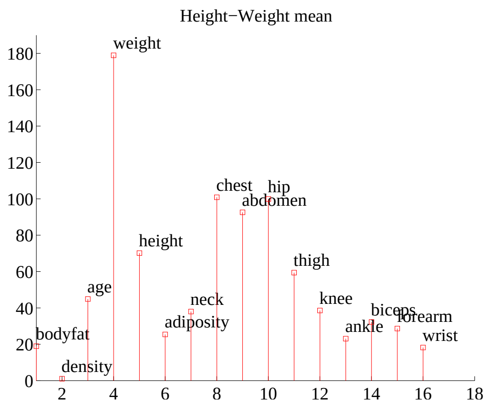


FIGURE 10.20: *The mean of the bodyfat.xls dataset. Each component is likely in a different unit (though I don't know the units), making it difficult to plot the data without being misleading. I've adopted one solution here, by plotting a stem plot. You shouldn't try to compare the values to one another. Instead, think of this plot as a compact version of a table.*

The next step is to try a principal component analysis. Figure 10.20 shows the mean of the dataset. The components of the dataset have different units, and shouldn't really be compared. But it is difficult to interpret a table of 16 numbers, so I have plotted the mean as a stem plot. Figure 10.21 shows the eigenvalues of the covariance for this dataset. Notice how one dimension is very important, and after the third principal component, the contributions become small. Of course, I could have said "fourth", or "fifth", or whatever — the precise choice depends on how small a number you think is "small".

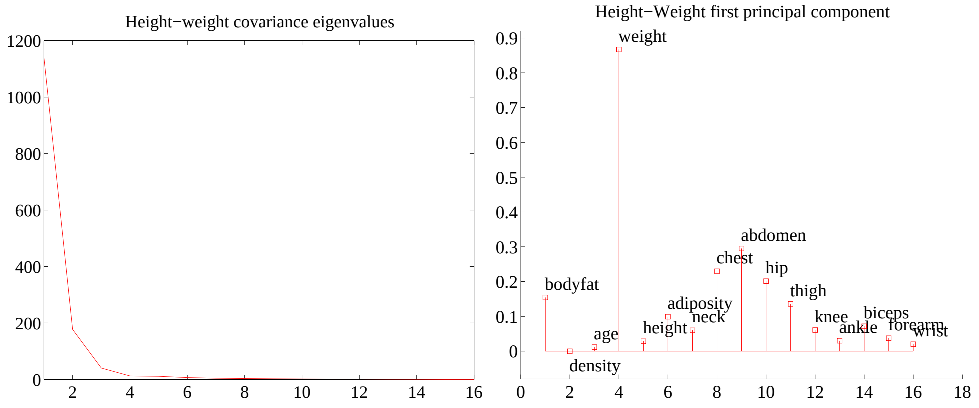


FIGURE 10.21: *On the* **left**, *the eigenvalues of the covariance matrix for the bodyfat data set. Notice how fast the eigenvalues fall off; this means that most principal components have very small variance, so that data can be represented well with a small number of principal components. On the* **right**, *the first principal component for this dataset, plotted using the same convention as for figure 10.20.*

Figure 10.21 also shows the first principal component. The eigenvalues justify thinking of each data item as (roughly) the mean plus some weight times this principal component. From this plot you can see that data items with a larger value of *weight* will also have larger values of most other measurements, except *age* and *density*. You can also see how much larger; if the weight goes up by 8.5 units, then the abdomen will go up by 3 units, and so on. This explains the main variation in the dataset.

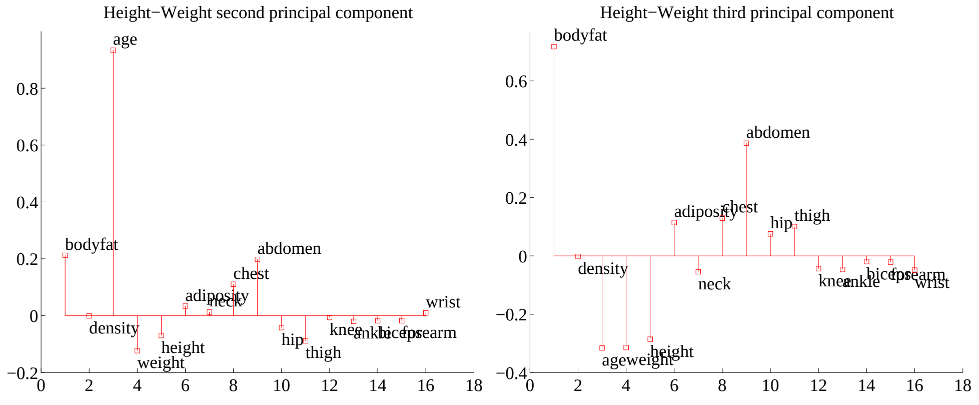


FIGURE 10.22: *On the* **left**, *the second principal component, and on the* **right** *the third principal component of the height-weight dataset.*

In the rotated coordinate system, the components are not correlated, and they have different variances (which are the eigenvalues of the covariance matrix). You can get some sense of the data by adding these variances; in this case, we get 1404. This means that, in the translated and rotated coordinate system, the average data point is about $37 = \sqrt{1404}$ units away from the center (the origin). Translations and rotations do not change distances, so the average data point is about 37 units from the center in the original dataset, too. If we represent a datapoint by using the mean and the first three principal components, there will be some error. We can estimate the average error from the component variances. In this case, the sum of the first three eigenvalues is 1357, so the mean square error in representing a datapoint by the first three principal components is $\sqrt{(1404 - 1357)}$, or 6.8. The relative error is $6.8/37 = 0.18$. Another way to represent this information, which is more widely used, is to say that the first three principal components explain all but $(1404 - 1357)/1404 = 0.034$, or 3.4% of the variance; notice that this is the square of the relative error, which will be a much smaller number.

All this means that explaining a data point as the mean and the first three principal components produces relatively small errors. Figure 10.23 shows the second and third principal component of the data. These two principal components suggest some further conclusions. As *age* gets larger, *height* and *weight* get slightly smaller, but the weight is redistributed; *abdomen* gets larger, whereas *thigh* gets smaller. A smaller effect (the third principal component) links *bodyfat* and *abdomen*. As *bodyfat* goes up, so does *abdomen*.

10.6 YOU SHOULD

10.6.1 remember these definitions:

Covariance	305
Covariance Matrix	306

10.6.2 remember these terms:

symmetric	311
eigenvector	311
eigenvalue	311
principal components	318
color constancy	320
principal coordinate analysis	325
multidimensional scaling	325
Likert scales	328

10.6.3 remember these facts:

Correlation from covariance	305
Properties of the covariance matrix	306
Orthonormal matrices are rotations	312
You can transform data to zero mean and diagonal covariance	313
A d-D dataset can be represented with s principal components, $s < d$	318

10.6.4 remember these procedures:

Diagonalizing a symmetric matrix	311
Principal Components Analysis	319
Principal Coordinate Analysis	326

10.6.5 be able to:

- Create, plot and interpret the first few principal components of a dataset.
- Compute the error resulting from ignoring some principal components.

CHAPTER 12

More Neural Networks

12.1 LEARNING TO MAP

Imagine we have a high dimensional dataset. As usual, there are N d-dimensional points $\mathbf{x}$, where the i'th point is $\mathbf{x}_i$. We would like to build a map of this dataset, to try and visualize its major features. We would like to know, for example, whether it contains many or few blobs; whether there are many scattered points; and so on. We might also want to plot this map using different plotting symbols for different kinds of data points. For example, if the data consists of images, we might be interested in whether images of cats form blobs that are distinct from images of dogs, and so on. I will write $\mathbf{y}_i$ for the point in the map corresponding the $\mathbf{x}_i$. The map is an M dimensional space (though M is almost always two or three in applications).

We have seen one tool for this exercise (section 8.1). This used eigenvectors to identify a linear projection of the data that made low dimensional distances similar to high dimensional distances. I argued that the choice of map should minimize

$$\sum_{i,j}\left(\| \mathbf{y}_i - \mathbf{y}_j \|^2 - \| \mathbf{x}_i - \mathbf{x}_j \|^2\right)^2$$

then rearranged terms to produce a solution that minimized

$$\sum_{i,j}\left(\mathbf{y}_i^T\mathbf{y}_j - \mathbf{x}_i^T\mathbf{x}_j\right)^2.$$

The solution produces a $\mathbf{y}_i$ that is a linear function of $\mathbf{x}_i$, just as a by-product of the mathematics. There are two problems with this approach (apart from the fact that I suppressed a bunch of detail). If the data lies on a curved structure in the high dimensional space, then a linear projection can distort the map very badly. Figure **??**) sketches one example.

You should notice that the original choice of cost function is not a particularly good idea, because our choice of map is almost entirely determined by points that are very far apart. This happens because squared differences between big numbers tend to be a lot bigger than squared differences between small numbers, and so distances between points that are far apart will be the most important terms in the cost function. In turn, this could mean our map does not really show the structure of the data – for example, a small number of scattered points in the original data could break up clusters in the map (the points in clusters are pushed apart to get a map that places the scattered points in about the right place with respect to each other).

12.1.1 Sammon Mapping

Sammon mapping is a method to fix these problems by modifying the cost function. We attempt to make the small distances more significant in the solution by minimizing

$$C(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \left(\frac{1}{\sum_{i<j} \| \mathbf{x}_i - \mathbf{x}_j \|} \right) \left[\frac{\sum_{i<j} \left(\| \mathbf{y}_i - \mathbf{y}_j \| - \| \mathbf{x}_i - \mathbf{x}_j \| \right)^2}{\| \mathbf{x}_i - \mathbf{x}_j \|} \right].$$

The first term is a constant that makes the gradient cleaner, but has no other effect. What is important is we are biasing the cost function to make the error in small distances much more significant. Unlike straightforward multidimensional scaling, the range of the sum matters here – if i equals j in the sum, then there will be a divide by zero.

No closed form solution is known for this cost function. Instead, choosing the $\mathbf{y}$ for each $\mathbf{x}$ is by gradient descent on the cost function. You should notice there is no unique solution here, because rotating, translating or reflecting all the $\mathbf{y}_i$ will not change the value of the cost function. Furthermore, there is no reason to believe that gradient descent necessarily produces the best value of the cost function. Experience has shown that Sammon mapping works rather well, but has one annoying feature. If one pair of high dimensional points is very much closer together than any other, then getting the mapping right for that pair of points is extremely important to obtain a low value of the cost function. This should seem like a problem to you, because a distortion in a very tiny distance should not be much more important than a distortion in a small distance.

12.1.2 T-SNE

We will now build a model by reasoning about probability rather than about distance (although this story could likely be told as a metric story, too). We will build a model of the probability that two points in the high dimensional space are neighbors, and another model of the probability that two points in the low dimensional space are neighbors. We will then adjust the locations of the points in the low dimensional space so that the KL divergence between these two models is small.

We reason first about the probability that points in the high dimensional space are neighbors. Write the conditional probability that $\mathbf{x}_j$ is a neighbor of $\mathbf{x}_i$ as $p_{j|i}$. Write

$$w_{j|i} = \exp\left(\frac{\| \mathbf{x}_j - \mathbf{x}_i \|^2}{2\sigma_i^2} \right)$$

We use the model

$$p_{j|i} = \frac{w_{j|i}}{\sum_k w_{k|i}}.$$

Notice this depends on the scale at point i, written σ_i. For the moment, we assume this is known. Now we define p_{ij} the joint probability that $\mathbf{x}_i$ and $\mathbf{x}_j$ are neighbors by assuming $p_{ii} = 0$, and for all other pairs

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}.$$

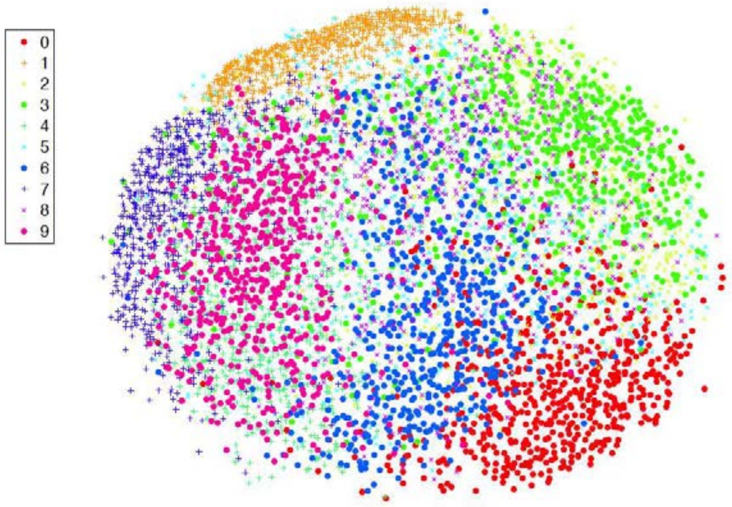


FIGURE 12.1: *A Sammon mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a Sammon mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 classes* $(0, \ldots 9)$*. The class labels were not used in training, but the plot shows class labels. This helps determine whether the visualization is any good – you could reasonably expect a visualization to put items in the same class close together and items in very different classes far apart. As the legend on the side shows, the classes are quite well separated. Figure from* Visualizing Data using t-SNE *Journal of Machine Learning Research 9 (2008) 2579-2605 Laurens van der Maaten and Geoffrey Hinton, to be replaced with a homemade figure in time.*

This is an $N \times N$ table of probabilities; you should check that this table represents a joint probability distribution (i.e. it's non-negative, and sums to one).

We use a slightly different probability model in the low dimensional space. We know that, in a high dimensional space, there is "more room" near a given point (think of this as a base point) than there is in a low dimensional space. This means that mapping a set of points from a high dimensional space to a low dimensional space is almost certain to move some points further away from the base point than we would like. In turn, this means there is a higher probability that a distant point in the low dimensional space is still a neighbor of the base point. Our probability model needs to have "long tails" – the probability that two points are neighbors should not fall off too quickly with distance. Write q_{ij} for the probability that $\mathbf{y}_i$ and $\mathbf{y}_j$ are neighbors. We assume that $q_{ii} = 0$ for all i. For other pairs, we use the model

$$q_{ij}(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \frac{{}^{1}/{}_{1+\|\mathbf{y}_i - \mathbf{y}_j\|^2}}{\sum_{k,l,k\neq l} {}^{1}/{}_{1+\|\mathbf{y}_i - \mathbf{y}_k\|^2}}$$

(where you might recognize the form of Student's t-distribution if you have seen that before). You should think about the situation like this. We have a table representing the probabilities that two points in the high dimensional space are neighbors, from our model of p_{ij}. The values of the **y** can be used to fill in an $N \times N$ joint probability table representing the probabilities that two points are neighbors. We would like this tables to be like one another. A natural metric of similarity is the KL-divergence, of section 8.1. So we will choose **y** to minimize

$$C_{tsne}(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}(\mathbf{y}_1, \ldots, \mathbf{y}_N)}.$$

Remember that $p_{ii} = q_{ii} = 0$, so adopt the convention that $0 \log 0/0 = 0$ to avoid embarrassment (or, if you don't like that, omit the diagonal terms from the sum). Gradient descent with a fixed steplength and momentum was be sufficient to minimize this in the original papers, though likely the other tricks of section 8.1 might help.

There are two missing details. First, the gradient has a quite simple form (which I shall not derive). We have

$$\nabla_{\mathbf{y}_i} C_{tsne} = 4 \sum_j \left[(p_{ij} - q_{ij}) \frac{(\mathbf{y}_i - \mathbf{y}_j)}{1 + \| \mathbf{y}_i - \mathbf{y}_j \|^2} \right].$$

Second, we need to choose σ_i. There is one such parameter per data point, and we need them to compute the model of p_{ij}. This is usually done by search, but to understand the search, we need a new term. The **perplexity** of a probability distribution with entropy $H(P)$ is defined by

$$\mathrm{Perp}(P) = 2^{H(P)}.$$

The search works as follows: the user chooses a value of perplexity; then, for each i, a binary search is used to choose σ_i such that $p_{j|i}$ has that perplexity. Experiments currently suggest that the results are quite robust to wide changes in the users choice.

In practical examples, it is quite usual to use PCA to get a somewhat reduced dimensional version of the **x**. So, for example, one might reduce dimension from 1,024 to 30 with PCA, then apply t-SNE to the result.

12.2 ENCODERS, DECODERS AND AUTO-ENCODERS

An **encoder** is a network that can take a signal and produce a code. Typically, this code is a description of the signal. For us, signals have been images and I will continue to use images as examples, but you should be aware that all I will say can be applied to sound and other signals. The code might be "smaller" than the original signal – in the sense it contains fewer numbers – or it might even be "bigger" – it will have more numbers, a case referred to as an **overcomplete** representation. You should see our image classification networks as encoders. They take images and produce short representations. A **decoder** is a network that can take a code and produce a signal. We have not seen decoders to date.

An **auto-encoder** is a learned pair of coupled encoder and decoder; the encoder maps signals into codes, and the decoder reconstructs signals from those

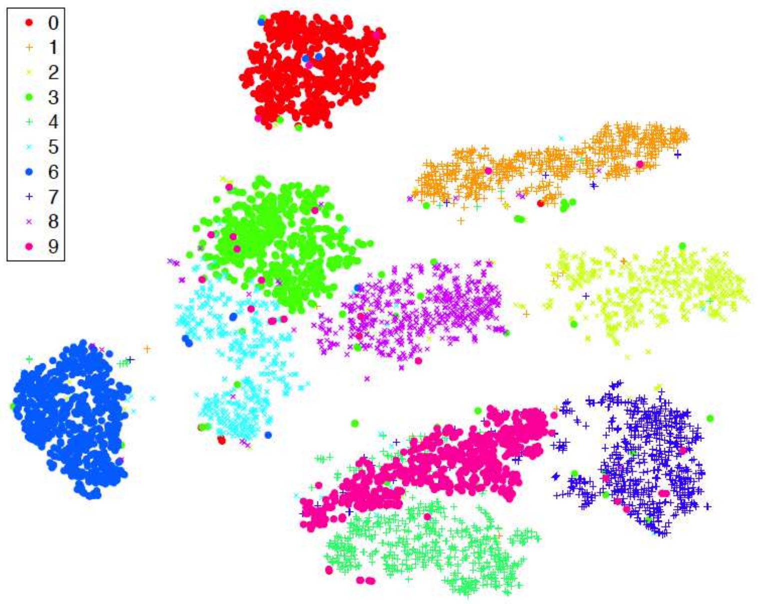


FIGURE 12.2: *A t-sne mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a t-sne mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 classes* $(0, \ldots 9)$. *The class labels were not used in training, but the plot shows class labels. This helps determine whether the visualization is any good – you could reasonably expect a visualization to put items in the same class close together and items in very different classes far apart. As the legend on the side shows, the classes are quite well separated. Figure from* Visualizing Data using t-SNE *Journal of Machine Learning Research 9 (2008) 2579-2605 Laurens van der Maaten and Geoffrey Hinton, to be replaced with a homemade figure in time.*

codes. Auto-encoders have great potential to be useful, which we will explore in the following sections. You should be aware that this potential has been around for some time, but has been largely unrealized in practice. One application is in unsupervised feature learning, where we try to construct a useful feature set from a set of unlabelled images. We could use the code produced by the auto-encoder as a source of features. Another possible use for an auto-encoder is to produce a clustering method – we use the auto-encoder codes to cluster the data. Yet another possible use for an auto-encoder is to generate images. Imagine we can train an auto-encoder so that (a) you can reconstruct the image from the codes and (b) the codes have a specific distribution. Then we could try to produce new images by feeding random samples from the code distribution into the decoder.

12.2.1 Auto-encoder Problems

Assume we wish to classify images, but have relatively few examples from each class. We can't use a deep network, and would likely use an SVM on some set of features, but we don't know what feature vectors to use. We could build an auto-encoder that produced an overcomplete representation, and use that overcomplete representation as a set of feature vectors. The decoder isn't of much interest, but we need to train with a decoder. The decoder ensures that the features actually describe the image (because you can reconstruct the image from the features). The big advantage of this approach is we could train the auto-encoder with a very large number of unlabelled images. We can then reasonably expect that, because the features describe the images in a quite general way, the SVM can find something discriminative in the set of features.

We will describe one procedure to produce an auto-encoder. The encoder is a layer that produces a code. For concreteness, we will discuss grey-level images, and assume the encoder is one convolutional layer. Write $\mathcal{I}_i$ for the i'th input image. All images will have dimension $m \times m \times 1$. We will assume that the encoder has r distinct units, and so produces a block of data that is $s \times s \times r$. Because there may be stride and convolution edge effects in the encoder, we may have that s is a lot smaller than m. Alternatively, we may have $s = m$. Write $\mathcal{E}(\mathcal{I}, \theta_e)$ for the encoder applied to image $\mathcal{I}$; here θ_e are the weights and biases of the units in the encoder. Write $Z_i = \mathcal{E}(\mathcal{I}_i, \theta_e)$ for the code produced by the encoder for the i'th image. The decoder must accept the output of the encoder and produce an $m \times m \times l$ image. Write $\mathcal{D}(Z, \theta_d)$ for the decoder applied to a code Z.

We have $Z_i = \mathcal{E}(\mathcal{I}_i, \theta_e)$, and would like to have $\mathcal{D}(Z_i, \theta_d)$ close to $\mathcal{I}_i$. We could enforce this by training the system, by stochastic gradient descent on θ_e, θ_d, to minimize $\| \mathcal{D}(Z_i, \theta_d) - \mathcal{I}_i \|^2$. One thing should worry you. If $s \times s \times r$ is larger than $m \times m$, then there is the possibility that the code is redundant in uninteresting ways. For example, if $s = m$, the encoder could consist of units that just pass on the input, and the decoder would pass on the input too – in this case, the code is the original image, and nothing of interest has happened.

12.2.2 The denoising auto-encoder

There is a clever trick to avoid this problem. We can require the codes to be robust, in the sense that if we feed a noisy image to the encoder, it will produce a code that recovers *the original image*. This means that we are requiring a code that not only describes the image, but is not disrupted by noise. Training an auto-encoder like this results in a **denoising auto-encoder**. Now the encoder and decoder can't just pass on the image, because the result would be the noisy image. Instead, the encoder has to try and produce a code that isn't affected (much) by noise, and the decoder has to take the possibility of noise into account while decoding.

Depending on the application, we could use one (or more) of a variety of different noise models. These impose slightly different requirements on the behavior of the encoder and decoder. There are three natural noise models: add independent samples of a normal random variable at each pixel (this is sometimes known as additive gaussian noise); take randomly selected pixels, and replace their values with 0 (masking noise); and take randomly selected pixels and replace their values with a random choice of brightest or darkest value (salt and pepper noise).

In the context of images, it is natural to use the least-squares error as a loss for training the auto-encoder. I will write noise($\mathcal{I}_i$) to mean the result of applying noise to image I_i. We can write out the training loss for example i as

$$\|\mathcal{D}(Z_i, \theta_d) - \mathcal{I}_i\|^2 \text{ where } Z_i = \mathcal{E}(\text{noise}(\mathcal{I}_i), \theta_e)$$

You should notice that masking noise and salt and pepper noise are different to additive gaussian noise, because for masking noise and salt and pepper noise *only some pixels* are affected by noise. It is natural to weight the least-square error at these pixels *higher* in the reconstruction loss – when we do so, we are insisting that the encoder learn a representation that is really quite good at predicting missing pixels. Training is by stochastic gradient descent, using one of the gradient tricks of section 8.1. Note that each time we draw a training example, we construct a new instance of noise for that version of the training example, so the encoding and decoding layer may see the same example with different sets of pixels removed, etc.

12.2.3 Stacking Denoising Auto-encoders

An encoder that consists of a single convolutional layer likely will not produce a rich enough representation to do anything useful. After all, the output of each unit depends only on a small neighborhood of pixels. We would like to train a multi-layer encoder. Experimental evidence over many years suggests that just building a multi-layer encoder network, hooking it to a multi-layer decoder network, and proceeding to train with stochastic gradient descent just doesn't work well. It is tough to be crisp about the reasons, but the most likely problem seems to be that interactions between the layers make the problem wildly ambiguous. For example, each layer could act to undo much of what the previous layer has done.

Here is a strategy that works for several different types of auto-encoder (though I will describe it only in the context of a denoising auto-encoder). First, we build a single layer encoder $\mathcal{E}$ and decoder $\mathcal{D}$ using the denoising auto-encoder strategy to get parameters θ_{e1} and θ_{d1}. The number of units, stride, support of units, etc. are chosen by experiment. We train this auto-encoder to get an acceptable reconstruction loss in the face of noise, as above.

Now I can think of each block of data $Z_{i1} = \mathcal{E}(I_i, \theta_{e1})$ as being "like" an image; it's just $s \times s \times r$ rather than $m \times m \times 1$. Notice that $Z_{i1} = \mathcal{E}(I_i, \theta_{e1})$ is the output of the encoder on a real image (rather than a real image with noise). I could build *another* denoising auto-encoder that handles Z_1's. In particular, I will build single layer encoder $\mathcal{E}$ and decoder $\mathcal{D}$ using the denoising auto-encoder strategy to get parameters θ_{e2} and θ_{d2}. This encoder/decoder pair must auto-encode the objects produced by the first pair. So I fix θ_{e1}, θ_{d1}, and the loss for image i as a function of θ_{e2}, θ_{d2} becomes

$$\begin{array}{lll} \|\mathcal{D}(Z_{i2}, \theta_{d2}) - Z_{i1}\|^2 & \text{where} & Z_{i2} = \mathcal{E}(\text{noise}(Z_{i1}), \theta_{e2}) \\ & \text{and} & Z_{i1} = \mathcal{E}(\mathcal{I}_1, \theta_{e1}) \end{array}$$

Again, training is by stochastic gradient descent using one of the tricks of section 8.1.

We can clearly apply this approach recursively, to stack train multiple layers. But more work is required to produce the best auto-encoder. In the two layer

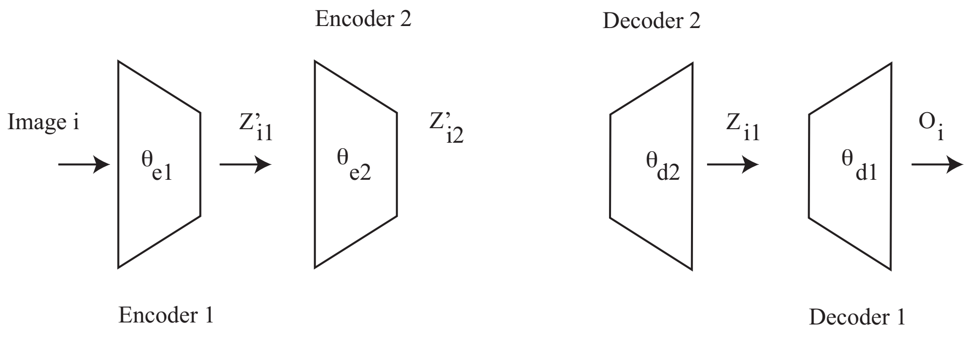


FIGURE 12.3: *Two layers of denoising auto-encoder, ready for fine tuning. This figure should help with the notation in the text.*

example, notice that the error does not take into account the effect of the first decoder on errors made by the second. We can fix this once all the layers have been trained *if* we need to use the result as an auto-encoder. This is sometimes referred to as *fine tuning*. We now train all the θ's. So, in the two layer case, the image passes into the first encoder, the result passes into the second encoder, then into the second decoder, then into the first decoder, and what emerges should be similar to the image. This gives a loss for image i in the two layer case as

$$\begin{array}{lll} \|\mathcal{D}(Z_{i1}, \theta_{d1}) - I_i\|^2 & \text{where} & Z_{i1} = \mathcal{D}((Z'_{i2}), \theta_{d2}) \\ & \text{and} & Z'_{i2} = \mathcal{E}(Z'_{i1}, \theta_{e2}) \\ & \text{and} & Z'_{i1} = \mathcal{E}(\mathcal{I}_i, \theta_{e1}) \end{array}$$

(Figure 12.3 might be helpful here).

12.2.4 Classification using an Auto-encoder

It isn't usually the case that we want to use an auto-encoder as a compression device. Instead, it's a way to learn features that we hope will be useful for some other purpose. One important case occurs when we have little labelled image data. There aren't enough labels to learn a full convolutional neural network, but we could hope that using an auto-encoder would produce usable features. The process involves: fit an auto-encoder to a large set of likely relevant image data; now discard the decoders, and regard the encoder stack as something that produces features; pass the code produced by the last layer of the stack into a fully connected layer; and fine-tune the whole system using labelled training data. There is good evidence that denoising auto-encoders work rather well as a way of producing features, at least for MNIST data.

12.3 MAKING IMAGES FROM SCRATCH WITH VARIATIONAL AUTO-ENCODERS

*** This isn't right - need to explain why I would try to generate from scratch? *** we talk about himages here, but pretty much everything applies to other signals too

12.3.1 Auto-Encoding and Latent Variable Models

There is a crucial, basic difficulty building a model to generate images. There is a lot of structure in an image. For most pixels, the colors nearby are about the same as the colors at that pixel. At some pixels, there are sharp changes in color. But these **edge points** are very highly organized spatially, too – they (largely) demarcate shapes. There is coherence at quite long spatial scales in images, too. For example, in an image of a donut sitting on a table, the color of the table inside the hole is about the same as the color outside. All this means that the overwhelming majority of arrays of numbers are not images. If you're suspicious, and not easily bored, draw samples from a multivariate normal distribution with unit covariance and see how long it will take before one of them even roughly looks like an image (hint: it won't happen in your lifetime, but looking at a few million samples is a fairly harmless way to spend time).

The structure in an image suggests a strategy. We could try to decode "short" codes to produce images. Write X for a random variable representing an image, and z for a code representing a compressed version of the image. Assume we can find a "good" model for $P(X|z, \theta)$. This might be built using a decoder network whose parameters are θ. Assume also that we can build codes and a decoder such that anything that comes out of the decoder looks like an image, *and* the probability distribution of codes corresponding to images is "easy". Then we could model $P(X)$ as

$$\int P(X|z, \theta)P(z)dz.$$

Such a model is known as a **latent variable model**. The codes z are **latent variables** – hidden values which, if known, would "explain" the image. In the first instance, assume we have a model of this form. Then generating an image would be simple in principle. We draw a sample from $P(z)$, then pass this through the network and regard the result as a sample from $P(X)$. This means that, for the model to be useful, we need to be able to actually draw these samples, and this constrains an appropriate choice of models. It is very natural to choose that $P(z)$ be a distribution that is easy to draw samples from. We will assume that $P(z)$ is a standard multivariate normal distribution (i.e. it has mean $\mathbf{0}$, and its covariance matrix is the identity). This is *by choice* – it's my model, and I made that choice.

However, we need to think very carefully about how to train such a model. One strategy might be to pass in samples from a normal distribution, then adjust the network parameters (by stochastic gradient descent, as always) to ensure what comes out is always an image. This isn't going to work, because it remains a remarkably difficult research problem to tell whether some array is an image or not. An alternative strategy is to build an encoder to make codes out of example images. We then train so that (a) the encoder produces codes that have a standard normal distribution and (b) the decoder takes the code computed from the i'th image and turns it into the i'th image. This isn't going to work either, because we're not taking account of the gaps between codes. We need to be sure that, if we present the decoder with *any* sample from a standard normal distribution (not just the ones we've seen), it will give us an image.

The correct strategy is as follows. We train an encoder and a decoder. Write

X_i for the i'th image, $\mathrm{E}(X_i) = z_i$ for the code produced by the decoder applied to X_i, $\mathrm{D}(z)$ for the image produced by the decoder on code z. For some image X_i, we produce $\mathrm{E}(X_i) = z_i$. We then obtain z *close to* z_i. Finally, we produce $\mathrm{D}(z)$. We train the encoder by requiring that the z "look like" IID samples from a standard normal distribution. We train the decoder by requiring that $\mathrm{D}(z)$ is close to X_i. Actually doing this will require some wading through probability, but the idea is quite clean.

12.3.2 Building a Model

Now, at least in principle, we could try to choose θ to maximize

$$\sum_i \log P(X_i|\theta).$$

But we have no way to evaluate the probability model, so this is hopeless. Recall the variational methods of chapters 8.1 and 8.1. Now choose some variational distribution $Q(z|X)$. This will have parameters, too, but I will suppress these and other parameters in the notation until we need to deal with them. Notice that

$$\begin{aligned}\mathbb{D}(Q(z|X) \,\|\, P(z|X)) &= \mathbb{E}_Q[\log Q(z|X) - \log P(z|X)] \\ &= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z) - \log P(X)] \\ &= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z)] + \log P(X))\end{aligned}$$

where the last line works because $\log P(X)$ doesn't depend on z. Recall the definition of the variational free energy from chapter 8.1. Write

$$\mathsf{E}_Q = \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(X|z) + \log P(z)]$$

and so we have

$$\log P(X) - \mathbb{D}(Q(z|X) \,\|\, P(z|X)) = -\mathsf{E}_Q.$$

We would like to maximize $\log P(X)$ by choice of parameters, but we can't because we can't compute it. But we do know that $\mathbb{D}(Q(z|X) \,\|\, P(z|X)) \geq 0$. This means that $-\mathsf{E}_Q$ is a *lower* bound on $\log P(X)$. If we maximize this lower bound (equivalently, minimize the variational free energy), then we can reasonably hope that we have a large value of $\log P(X)$. The big advantage of this observation is that we *can* work with $-\mathsf{E}_Q$.

12.3.3 Turning the VFE into a Loss

The best case occurs when $Q(z|X) = P(z|X)$ (because then $\mathbb{D}(Q(z|X) \,\|\, P(z|X)) = 0$, and the lower bound is tight). We don't expect this to occur in practice, but it suggests a way of thinking about the problem. We can build our model of $Q(z|X)$ around an encoder that predicts a code from an image. Similarly, our model of $P(X|z)$ would be built around a decoder that predicts an image from a code.

We can simplify matters by rewriting the expression for the variational free energy. We have

$$\begin{aligned}-\mathsf{E}_Q &= -\mathbb{E}_Q[\log Q] + \mathbb{E}_Q[\log P(X|z) + \log P(z)] \\ &= \mathbb{E}_Q[\log P(X|z)] - \mathbb{D}(Q(z|X) \,\|\, P(z)).\end{aligned}$$

We want to build a model of $Q(z|X)$, which is a probability distribution, using a neural network. This model accepts an image, X, and needs to produce a *random* code z which depends on X. We will do this by using the network to predict the mean and covariance of a normal distribution, then drawing the code z from a normal distribution with that mean and covariance. I will write $\mu(X)$ for the mean and $\Sigma(X)$ for the covariance, where the (X) is there to remind you that these are functions of the input, and they are modelled by the neural network. We choose the covariance to be diagonal, because the code might be quite large and we do not wish to try and learn large covariance matrices.

Now consider the term $\mathbb{D}(Q(z|X) \| P(z))$. We get to choose the prior on the code, and we choose $P(z)$ to be a standard normal distribution (i.e. mean $\mathbf{0}$, covariance matrix the identity; I'll duck the question of the dimension of z for the moment). We can write

$$Q(z|X) = \mathcal{N}(\mu(X); \Sigma(X)).$$

We need to compute the KL-divergence between this distribution and a standard normal distribution. This can be done in closed form. For reference (if you don't feel like doing the integrals yourself, and can't look it up elsewhere), the KL-divergence between two multivariate normal distributions for k dimensional vectors is

$$\mathbb{D}(\mathcal{N}(\mu_0; \Sigma_0) \,\|\, \mathcal{N}(\mu_1; \Sigma_1)) = \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^T \sigma_1^{-1} (\mu_1 - \mu_0) \\ -k + \log\left(\frac{\mathsf{Det}(\Sigma_1)}{\mathsf{Det}(\Sigma_0)}\right) \end{array} \right).$$

In turn, this means that

$$\mathbb{D}(\mathcal{N}(\mu(X); \Sigma(X)) \,\|\, \mathcal{N}(\mathbf{0}; \mathcal{I})) = \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma(X)\right) + \mu(X)^T \mu(X) \\ -k - \log\left(\mathsf{Det}\left(\Sigma\right)\right) \end{array} \right).$$

At this point, we are close to having an expression for a loss that we can actually minimize. We must deal with the term $\mathbb{E}_Q[\log P(X|z)]$. Recall that we modelled $Q(z|X)$ by drawing z from a normal distribution with mean $\mu(X)$ and covariance $\Sigma(X)$. We can obtain such a z by drawing from a standard normal distribution, then multiplying by $\Sigma(X)^{1/2}$ and adding back the mean $\mu(X)$. In equations, we have

$$\begin{array}{rcl} \mathbf{u} & \sim & \mathcal{N}(\mathbf{0}; \mathcal{I}) \\ z & = & \mu(X) + \Sigma(X)^{1/2}\mathbf{u} \\ \log P(X|z) & = & \log P(X|\mu(X) + \Sigma(X)^{1/2}\mathbf{u}). \end{array}$$

Our data X consists of a collection of images which we believe are IID samples from $P(X)$. I will write X_i for the i'th image. Originally, we wanted to choose parameters to maximize

$$\begin{array}{rcl} \log P(X) & = & \sum_i \log P(X_i) \\ & = & \mathbb{D}(Q(z|X) \,\|\, P(z|X)) - \mathsf{E}_{Q(z|X)} \\ & = & \sum_i \left[\mathbb{D}(Q(z|X_i) \,\|\, P(z|X_i)) - \mathsf{E}_{Q(z|X_i)}\right]. \end{array}$$

It's usual to train networks to minimize losses. We can write the loss as

$$\begin{aligned}\mathsf{E}_Q &= -\mathbb{E}_Q[\log Q] + \mathbb{E}_Q[\log P(X|z) + \log P(z)]\\ &= \mathbb{D}(Q(z|X) \,\|\, P(z)) - \mathbb{E}_Q[\log P(X|z)]\\ &= \sum_i \left[\mathbb{D}(Q(z|X_i) \,\|\, P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right].\end{aligned}$$

I am now going to insert parameters. I will write parameters θ, with a subscript that tells you what the parameters are for. Recall we modelled Q with a network that took an image X_i and produced a mean $\mu(X_i;\theta_\mu)$ and a covariance $\Sigma(X_i;\theta_\Sigma)$. This network is an encoder - it makes codes (the means) from images. We will need a decoder to model $P(X|z)$. We will write $\mathrm{D}(z;\theta_D)$ for a network that produces an image from a code. We assume that images are given by $P(X|z) = \mathcal{N}(\mathrm{D}(z;\theta_D);\mathcal{I})$, so that

$$\log P(X_i|z) = \frac{-\left(\| X_i - \mathrm{D}(z;\theta_D)\|^2\right)}{2}.$$

So the loss becomes

$$\begin{aligned}\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \,\|\, P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right]\\ &= \sum_i \left[\begin{array}{c}\frac{1}{2}\left(\begin{array}{c}\mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu)\\ -k - \log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right)\end{array}\right)\\ -\mathbb{E}_{Q(z|X_i)}\left[\frac{-\left(\|X_i - \mathrm{D}(z;\theta_D)\|^2\right)}{2}\right]\end{array}\right].\end{aligned}$$

The expectation term is a nuisance. We will approximate the expectation by drawing one sample from $Q(z|X)$ and averaging over that one sample. Assume $\mathbf{u}_i$ is an IID sample of $\mathcal{N}(\mathbf{0};\mathcal{I})$. Then we write

$$\begin{aligned}\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \,\|\, P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right]\\ &\approx \sum_i \left[\begin{array}{c}\frac{1}{2}\left(\begin{array}{c}\mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu) - k\\ - \log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right)\end{array}\right)\\ -\frac{-\left(\|X_i - \mathrm{D}(\mu(X_i;\theta_\mu) + \Sigma(X_i;\theta_\Sigma)^{1/2}\mathbf{u}_i;\theta_D)\|^2\right)}{2}\end{array}\right].\end{aligned}$$

This is a loss, and it can be differentiated in θ_μ, θ_Σ, and θ_D. To train a variational auto-encoder, we use stochastic gradient descent with a variety of tricks on this loss.

12.3.4 Some Caveats

As of writing, variational auto-encoders are the cutting edge of generative models. They seem to be better at generating images than any other technology. However, they are interesting because a really strong generative model for images would be extremely useful, not because they're particularly good at generating images.

There are a variety of important problems. Solutions to any, or all, of these problems would be very exciting, because it is extremely useful to be able to generate images.

Training: Variational auto-encoders are notoriously hard to train. There's a strong tendency to get no descent in the initial stages of training. The usual way to manage this is to weight the loss terms. You can break the loss into two terms. One measures the similarity of the code distribution to the normal distribution, the other measures the accuracy of reconstruction. Current practice weights the reconstruction loss very high in the early stages of training, then reduces that weight as training proceeds. This seems to help, for reasons I can't explain and have never seen explained.

Small images: Variational auto-encoders produce small images. Images bigger than 64×64 are tough to produce.

Mysterious code properties: There seems to be some limit to the complexity of the family of images that a variational auto-encoder can produce. This means that MNIST (for example) pretty much always works quite convincingly, but auto-encoding all the images in (say) ImageNet doesn't produce particularly good results. There is likely some relationship between the size of the code and the complexity of the family of images, but the effectiveness of training has something to do with it as well.

Blurry reconstructions: Variational auto-encoders produce blurry images. This is somewhat predictable from the loss and the training process. I know two arguments, neither completely rigorous. First, the image loss is L_2 error, which always produces blurry images because it regards a sharp edge in the wrong place as interchangeable with a slower edge in the right place. Second, the code distributions predicted by the encoder for two similar images must overlap; this means that the decoder is being trained to produce two distinct images for the same z, which must mean it averages and so loses detail.

Gaps in the code space: Codes are typically 32 dimensional. Expecting to produce a good estimate of an expectation with a single sample in a 32 dimensional space is a bit ambitious. This means that, in turn, there are many points in the code space that have never been explored by the encoder, or used in training the decoder. As a result, it is likely that a small search around a code can produce another code that generates a truly awful image. Of course, this result will only appear during an important live demo...