Chapter 15

Linear Factor Models and Auto-Encoders

Linear factor models are generative unsupervised learning models in which we imagine that some unobserved factors $\boldsymbol{h}$ explain the observed variables $\boldsymbol{x}$ through a linear transformation. Auto-encoders are unsupervised learning methods that learn a representation of the data, typically obtained by a non-linear parametric transformation of the data, i.e., from $\boldsymbol{x}$ to $\boldsymbol{h}$, typically a feedforward neural network, but not necessarily. They also learn a transformation going backwards from the representation to the data, from $\boldsymbol{h}$ to $\boldsymbol{x}$, like the linear factor models. Linear factor models therefore only specify a parametric decoder, whereas auto-encoder also specify a parametric encoder. Some linear factor models, like PCA, actually correspond to an auto-encoder (a linear one), but for others the encoder is implicitly defined via an inference mechanism that searches for an $\boldsymbol{h}$ that could have generated the observed $\boldsymbol{x}$.

The idea of auto-encoders has been part of the historical landscape of neural networks for decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994) but has really picked up speed in recent years. They remained somewhat marginal for many years, in part due to what was an incomplete understanding of the mathematical interpretation and geometrical underpinnings of auto-encoders, which are developed further in Chapters 17 and 20.12.

An auto-encoder is simply a neural network that tries to copy its input to its output. The architecture of an auto-encoder is typically decomposed into the following parts, illustrated in Figure 15.1:

- an input, $\boldsymbol{x}$
- an encoder function f
- a "*code*" or internal representation $\boldsymbol{h} = f(\boldsymbol{x})$

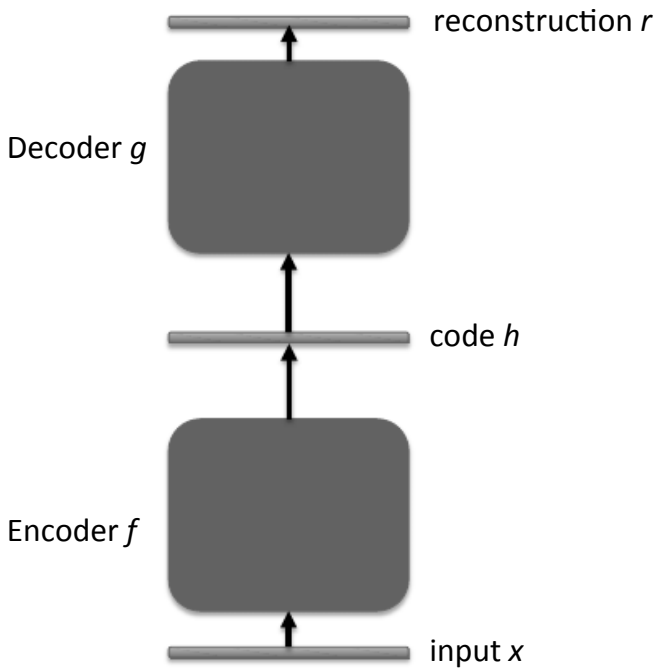

Figure 15.1: General schema of an auto-encoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The auto-encoder has two components: the encoder f (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder g (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

- a decoder function g
- an output, also called "*reconstruction*" $\boldsymbol{r} = g(\boldsymbol{h}) = g(f(\boldsymbol{x}))$
- a loss function L computing a scalar $L(\boldsymbol{r}, \boldsymbol{x})$ measuring how good of a reconstruction $\boldsymbol{r}$ is of the given input $\boldsymbol{x}$. The objective is to minimize the expected value of L over the training set of examples $\{\boldsymbol{x}\}$.

15.1 Regularized Auto-Encoders

Predicting the input may sound useless: what could prevent the auto-encoder from simply copying its input into its output? In the 20th century, this was achieved by constraining the architecture of the auto-encoder to avoid this, by forcing the dimension of the code $\boldsymbol{h}$ to be smaller than the dimension of the input $\boldsymbol{x}$.

Figure 15.2 illustrates the two typical cases of auto-encoders: undercomplete (with the dimension of the representation $\boldsymbol{h}$ smaller than the dimension of the input $\boldsymbol{x}$), andovercomplete (with the dimension of $\boldsymbol{h}$ larger than that of $\boldsymbol{x}$). Whereas early work with auto-encoders, just like PCA, uses an undercomplete bottleneck in the sequence of layers to avoid learning the identity function, more recent work

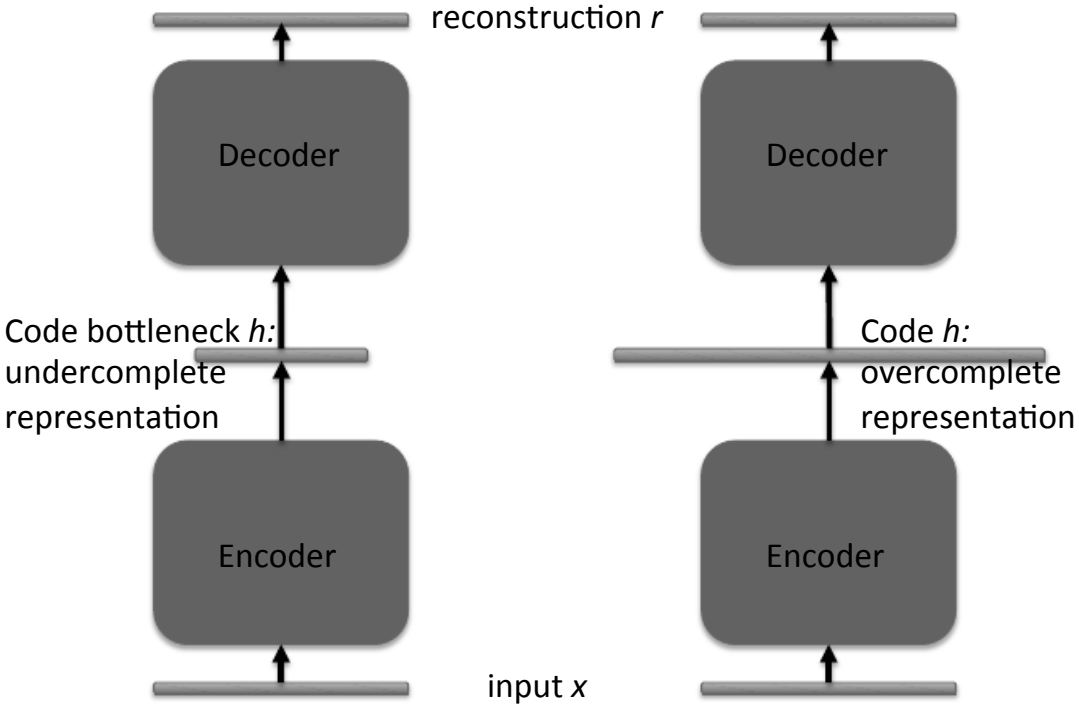

Figure 15.2: Left: undercomplete representation (dimension of code $\boldsymbol{h}$ is less than dimension of input $\boldsymbol{x}$). Right: overcomplete representation. Overcomplete auto-encoders require some other form of regularization (instead of the constraint on the dimension of $\boldsymbol{h}$) to avoid the trivial solution where $\boldsymbol{r} = \boldsymbol{x}$ for all $\boldsymbol{x}$.

allows overcomplete representations. What we have learned in recent years is that it is possible to make the auto-encoder meaningfully capture the structure of the input distribution even if the representation is overcomplete, with other forms of constraint or regularization. In fact, once you realize that auto-encoders can capture the input distribution (indirectly, not as a an explicit probability function), you also realize that it should need more capacity as one increases the complexity of the distribution to be captured (and the amount of data available): it should not be limited by the input dimension. This is a problem in particular with the shallow auto-encoders, which have a single hidden layer (for the code). Indeed, that hidden layer size controls both the dimensionality reduction constraint (the code size at the bottleneck) and the capacity (which allows to learn a more complex distribution).

Besides the **bottleneck** constraint, alternative constraints or regularization methods have been explored and can guarantee that the auto-encoder does something useful and not just learn some trivial identity-like function:

- **Sparsity of the representation or of its derivative**: even if the intermediate representation has a very high dimensionality, the effective local dimensionality (number of degrees of freedom that capture a coordinate sys-

tem among the probable x's) could be much smaller if most of the elements of h are zero (or any other constant, such that $||\frac{\partial h_i}{\partial \boldsymbol{x}}||$ is close to zero). When $||\frac{\partial h_i}{\partial \boldsymbol{x}}||$ is close to zero, h_i does not participate in encoding local changes in $\boldsymbol{x}$. There is a geometrical interpretation of this situation in terms of *manifold learning* that is discussed in more depth in Chapter 17. The discussion in Chapter 16 also explains how an auto-encoder naturally tends towards learning a coordinate system for the actual factors of variation in the data. At least four types of "auto-encoders" clearly fall in this category of sparse representation:

- **Sparse coding** (Olshausen and Field, 1996) has been heavily studied as an unsupervised feature learning and feature inference mechanism. It is a linear factor model rather than an auto-encoder, because it has no explicit parametric encoder, and instead uses an iterative optimization procedure to compute the maximally likely code. Sparse coding looks for representations that are both sparse and explain the input through the decoder. Instead of the code being a parametric function of the input, it is considered like free variable that is obtained through an optimization, i.e., a particular form of inference:

$$\boldsymbol{h}^* = f(\boldsymbol{x}) = \arg\min_{\boldsymbol{h}} L(g(\boldsymbol{h}), \boldsymbol{x})) + \lambda\Omega(\boldsymbol{h}) \tag{15.1}$$

  where L is the reconstruction loss, f the (non-parametric) encoder, g the (parametric) decoder, $\Omega(\boldsymbol{h})$ is a sparsity regularizer, and in practice the minimization can be approximate. Sparse coding has a manifold or geometric interpretation that is discussed in Section 15.8. It also has an interpretation as a directed graphical model, described in more details in Section 19.3. To achieve sparsity, the objective function to optimize includes a term that is minimized when the representation has many zero or near-zero values, such as the L1 penalty $|\boldsymbol{h}|_1 = \sum_i |h_i|$.
- An interesting variation of sparse coding combines the freedom to choose the representation through optimization and a parametric encoder. It is called **predictive sparse decomposition** (PSD) (Kavukcuoglu *et al.*, 2008a) and is briefly described in Section 15.8.2.
- At the other end of the spectrum are simply **sparse auto-encoders**, which combine with the standard auto-encoder schema a sparsity penalty which encourages the output of the encoder to be sparse. These are described in Section 15.8.1. Besides the L1 penalty, other sparsity penalties that have been explored include the Student-t penalty (Olshausen and Field, 1996; Bergstra, 2011), TODO: should the t be in

math mode, perhaps?

$$\sum_i \log(1 + \alpha^2 h_i^2)$$

(i.e. where αh_i has a Student-t prior density) and the KL-divergence penalty (Lee *et al.*, 2008; Goodfellow *et al.*, 2009; Larochelle and Bengio, 2008a)

$$-\sum_i (t \log h_i + (1-t)\log(1-h_i)),$$

with a target sparsity level t, for $h_i \in (0, 1)$, e.g. through a sigmoid non-linearity.

- **Contractive autoencoders** (Rifai *et al.*, 2011b), covered in Section 15.10, explicitly penalize $||\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}||_F^2$, i.e., the sum of the squared norm of the vectors $\frac{\partial h_i(\boldsymbol{x})}{\partial \boldsymbol{x}}$ (each indicating how much each hidden unit h_i responds to changes in $\boldsymbol{x}$ and what direction of change in $\boldsymbol{x}$ that unit is most sensitive to, around a particular $\boldsymbol{x}$). With such a regularization penalty, the auto-encoder is called **contractive**[1] because the mapping from input $\boldsymbol{x}$ to representation $\boldsymbol{h}$ is encouraged to be contractive, i.e., to have small derivatives in all directions. Note that a sparsity regularization indirectly leads to a contractive mapping as well, when the non-linearity used happens to have a zero derivative at $h_i = 0$ (which is the case for the sigmoid non-linearity).

- **Robustness to injected noise or missing information**: if noise is injected in inputs or hidden units, or if some inputs are missing, while the neural network is asked to *reconstruct the clean and complete input*, then it cannot simply learn the identity function. It has to capture the structure of the data distribution in order to optimally perform this reconstruction. Such auto-encoders are called *denoising auto-encoders* and are discussed in more detail in Section 15.9.

15.2 Denoising Auto-encoders

There is a tight connection between the denoising auto-encoders and the contractive auto-encoders: it can be shown (Alain and Bengio, 2013) that in the limit of small Gaussian injected input noise, the denoising reconstruction error is equivalent to a contractive penalty on the reconstruction function that maps $\boldsymbol{x}$ to $\boldsymbol{r} = g(f(\boldsymbol{x}))$. In other words, since both $\boldsymbol{x}$ and

[1] A function $f(\boldsymbol{x})$ is contractive if $||f(\boldsymbol{x}) - f(\boldsymbol{y})|| < ||\boldsymbol{x} - \boldsymbol{y}||$ for nearby $\boldsymbol{x}$ and $\boldsymbol{y}$, or equivalently if its derivative $||f'(\boldsymbol{x})|| < 1$.

$\boldsymbol{x}+\boldsymbol{\epsilon}$ (where $\boldsymbol{\epsilon}$ is some small noise vector) must yield the same target output $\boldsymbol{x}$, the reconstruction function is encouraged to be insensitive to changes in all directions ϵ. The only thing that prevents reconstruction $\boldsymbol{r}$ from simply being a constant (completely insensitive to the input $\boldsymbol{x}$), is that one also has to reconstruct correctly for different training examples $\boldsymbol{x}$. However, the auto-encoder can learn to be approximately constant around training examples $\boldsymbol{x}$ while producing a different answer for different training examples. As discussed in Section 17.4, if the examples are near a low-dimensional manifold, this encourages the representation to vary only on the manifold and be locally constant in directions orthogonal to the manifold, i.e., the representation locally captures a (not necessarily Euclidean, not necessarily orthogonal) coordinate system for the manifold. In addition to the denoising auto-encoder, the *variational auto-encoder* (Section 20.9.3) and the *generative stochastic networks* (Section 20.12) also involve the injection of noise, but typically in the representation-space itself, thus introducing the notion of $\boldsymbol{h}$ as a *latent variable*.

- **Pressure of a Prior on the Representation:** an interesting way to generalize the notion of regularization applied to the representation is to introduce in the cost function for the auto-encoder a log-prior term

$$-\log P(\boldsymbol{h})$$

which captures the assumption that we would like to find a representation that has a simple distribution (if $P(\boldsymbol{h})$ has a simple form, such as a factorized distribution[2]), or at least one that is simpler than the original data distribution. Among all the encoding functions f, we would like to pick one that

1. can be inverted (easily), and this is achieved by minimizing some reconstruction loss, and
2. yields representations $\boldsymbol{h}$ whose distribution is "simpler", i.e., can be captured with less capacity than the original training distribution itself.

The sparse variants described above clearly fall in that framework. The variational auto-encoder (Section 20.9.3) provides a clean mathematical framework for justifying the above pressure of a top-level prior when the objective is to model the data generating distribution.

From the point of view of regularization (Chapter 7), adding the $-\log P(\boldsymbol{h})$ term to the objective function (e.g. for encouraging sparsity) or adding a contractive penalty do not fit the traditional view of a prior on the parameters. Instead,

[2] all the sparse priors we have described correspond to a factorized distribution

the prior on the latent variables acts like a *data-dependent prior*, in the sense that it depends on the particular values $\boldsymbol{h}$ that are going to be sampled (usually from a posterior or an encoder), based on the input example $\boldsymbol{x}$. Of course, indirectly, this is also a regularization on the parameters, but one that depends on the particular data distribution.

15.3 Representational Power, Layer Size and Depth

Autoencoders are often trained with only a single layer encoder and a single layer decoder. However, this is not a requirement, and using deep encoders and decoders offers many advantages.

Recall from Sec. 6.6 that there are many advantages to depth in a feed-forward network. Because auto-encoders are feed-forward networks, these advantages also apply to auto-encoders. Moreover, the encoder is itself a feed-forward network as is the decoder, so each of these components of the auto-encoder can individually benefit from depth.

One major advantage of non-trivial depth is that the universal approximator theorem guarantees that a feedforward neural network with at least one hidden layer can represent an approximation of any function (within a broad class) to an arbitrary degree of accuracy, provided that it has enough hidden units. This means that an autoencoder with a single hidden layer is able to represent the identity function along the domain of the data arbitrarily well. However, the mapping from input to code is shallow. This means that we are not able to enforce arbitrary constraints, such as that the code should be sparse. A deep autoencoder, with at least one additional hidden layer inside the encoder itself, can approximate any mapping from input to code arbitrarily well, given enough hidden units.

The above viewpoint also motivates overcomplete autoencoders, that is, autoencoders with very wide layers, in order to achieve a rich family of possible functions.

Depth can exponentially reduce the computational cost of evaluating a representation of some functions, and can also exponentially decrease the amount of training data needed to learn some functions.

Experimentally, deep auto-encoders yield much better compression than corresponding shallow or linear auto-encoders (Hinton and Salakhutdinov, 2006).

A common strategy for training a deep autoencoder is to greedily pre-train the deep architecture by training a stack of shallow auto-encoders, so we often encounter shallow auto-encoders, even when the ultimate goal is to train a deep auto-encoder.

15.4 Reconstruction Distribution

The above "parts" (encoder function f, decoder function g, reconstruction loss L) make sense when the loss L is simply the squared reconstruction error, but there are many cases where this is not appropriate, e.g., when $\boldsymbol{x}$ is a vector of discrete variables or when $P(\boldsymbol{x} \mid \boldsymbol{h})$ is not well approximated by a Gaussian distribution [3]. Just like in the case of other types of neural networks (starting with the feedforward neural networks, Section 6.3.2), it is convenient to define the loss L as a negative log-likelihood over some target random variables. This probabilistic interpretation is particularly important for the discussion in Sections 20.9.3, 20.11 and 20.12 about generative extensions of auto-encoders and stochastic recurrent networks, where the output of the auto-encoder is interpreted as a probability distribution $P(\boldsymbol{x} \mid \boldsymbol{h})$, for reconstructing $\boldsymbol{x}$, given hidden units $\boldsymbol{h}$. This distribution captures not just the expected reconstruction but also the *uncertainty* about the original $\boldsymbol{x}$ (which gave rise to $\boldsymbol{h}$, either deterministically or stochastically, given $\boldsymbol{h}$). In the simplest and most ordinary cases, this distribution factorizes, i.e., $P(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_i P(x_i \mid \boldsymbol{h})$. This covers the usual cases of $x_i \mid \boldsymbol{h}$ being Gaussian (for unbounded real values) and $x_i | \boldsymbol{h}$ having a Bernoulli distribution (for binary values x_i), but one can readily generalize this to other distributions, such as mixtures (see Sections 3.10.6 and 6.3.2).

Thus we can generalize the notion of *decoding function* $g(\boldsymbol{h})$ to *decoding distribution* $P(\boldsymbol{x} \mid \boldsymbol{h})$. Similarly, we can generalize the notion of *encoding function* $f(\boldsymbol{x})$ to *encoding distribution* $Q(\boldsymbol{h} \mid \boldsymbol{x})$, as illustrated in Figure 15.3. We use this to capture the fact that noise is injected at the level of the representation $\boldsymbol{h}$, now considered like a latent variable. This generalization is crucial in the development of the variational auto-encoder (Section 20.9.3) and the generalized stochastic networks (Section 20.12).

We also find a stochastic encoder and a stochastic decoder in the RBM, described in Section 20.2. In that case, the encoding distribution $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$ "match", in the sense that $Q(\boldsymbol{h} \mid \boldsymbol{x}) = P(\boldsymbol{h} \mid \boldsymbol{x})$, i.e., there is a unique joint distribution which has both $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$ as conditionals. This is not true in general for two independently parametrized conditionals like $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$, although the work on generative stochastic networks (Alain *et al.*, 2015) shows that learning will tend to make them compatible asymptotically (with enough capacity and examples).

[3] See the link between squared error and normal density in Sections 5.6 and 6.3.2

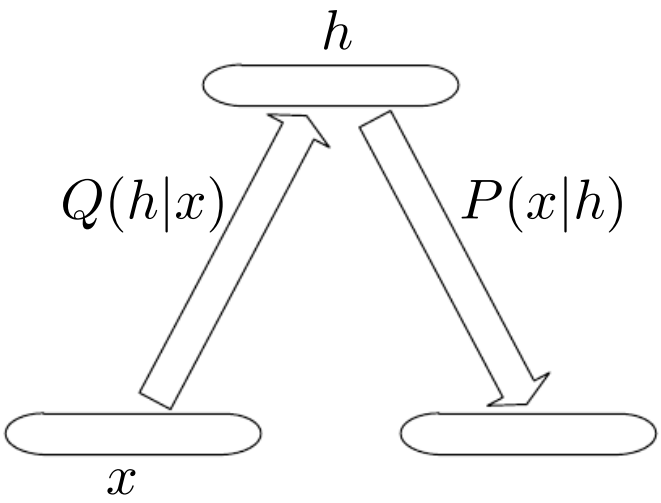

Figure 15.3: Basic scheme of a stochastic auto-encoder, in which both the encoder and the decoder are not simple functions but instead involve some noise injection, meaning that their output can be seen as sampled from a distribution, $Q(\boldsymbol{h} \mid \boldsymbol{x})$ for the encoder and $P(\boldsymbol{x} \mid \boldsymbol{h})$ for the decoder. RBMs are a special case where $P = Q$ (in the sense of a unique joint corresponding to both conditinals) but in general these two distributions are not necessarily conditional distributions compatible with a unique joint distribution $P(\boldsymbol{x}, \boldsymbol{h})$.

15.5 Linear Factor Models

Now that we have introduced the notion of a probabilistic decoder, let us focus on a very special case where the latent variable $\boldsymbol{h}$ generates $\boldsymbol{x}$ via a linear transformation plus noise, i.e., classical linear factor models, which do not necessarily have a corresponding parametric encoder.

The idea of discovering explanatory factors that have a simple joint distribution among themselves is old, e.g., see Factor Analysis (see below), and has been explored first in the context where the relationship between factors and data is linear, i.e., we assume that the data was generated as follows. First, sample the real-valued factors,

$$\boldsymbol{h} \sim P(\boldsymbol{h}), \tag{15.2}$$

and then sample the real-valued observable variables given the factors:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{15.3}$$

where the noise is typically Gaussian and diagonal (independent across dimensions). This is illustrated in Figure 15.4.

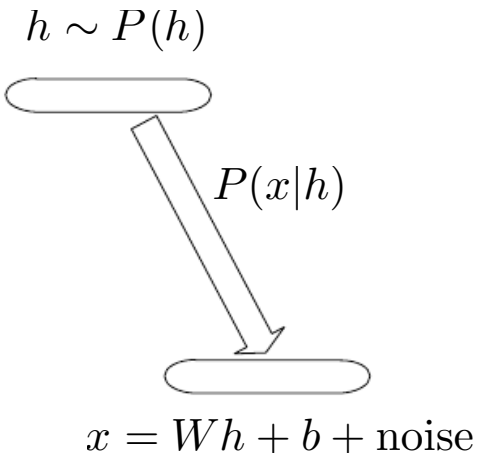

Figure 15.4: Basic scheme of a linear factors model, in which we assume that an observed data vector $\boldsymbol{x}$ is obtained by a linear combination of latent factors $\boldsymbol{h}$, plus some noise. Different models, such as probabilistic PCA, factor analysis or ICA, make different choices about the form of the noise and of the prior $P(\boldsymbol{h})$.

15.6 Probabilistic PCA and Factor Analysis

Probabilistic PCA (Principal Components Analysis), factor analysis and other linear factor models are special cases of the above equations (15.2 and 15.3) and only differ in the choices made for the prior (over latent, not parameters) and noise distributions.

In factor analysis (Bartholomew, 1987; Basilevsky, 1994), the latent variable prior is just the unit variance Gaussian

$$\boldsymbol{h} \sim \mathcal{N}(0, \boldsymbol{I})$$

while the observed variables x_i are assumed to be *conditionally independent*, given $\boldsymbol{h}$, i.e., the noise is assumed to be coming from a diagonal covariance Gaussian distribution, with covariance matrix $\boldsymbol{\psi} = \text{diag}(\boldsymbol{\sigma}^2)$, with $\boldsymbol{\sigma}^2 = (\sigma_1^2, \sigma_2^2, \ldots)$ a vector of per-variable variances.

The role of the latent variables is thus to *capture the dependencies* between the different observed variables x_i. Indeed, it can easily be shown that $\boldsymbol{x}$ is just a Gaussian-distribution (multivariate normal) random variable, with

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \boldsymbol{\psi})$$

where we see that the weights $\boldsymbol{W}$ induce a dependency between two variables x_i and x_j through a kind of auto-encoder path, whereby x_i influences $\hat{\boldsymbol{h}}_k = \boldsymbol{W}_k\boldsymbol{x}$ via w_{ki} (for every k) and $\hat{\boldsymbol{h}}_k$ influences x_j via w_{kj}.

In order to cast PCA in a probabilistic framework, we can make a slight modification to the factor analysis model, making the conditional variances σ_i

equal to each other. In that case the covariance of $\boldsymbol{x}$ is just $\boldsymbol{W}\boldsymbol{W}^\top + \sigma^2 \boldsymbol{I}$, where σ^2 is now a scalar, i.e.,

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2 \boldsymbol{I})$$

or equivalently

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \sigma \boldsymbol{z}$$

where $\boldsymbol{z} \sim \mathcal{N}(0, \boldsymbol{I})$ is white noise. Tipping and Bishop (1999) then show an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$ and σ^2.

What the probabilistic PCA model is basically saying is that the covariance is mostly captured by the latent variables $\boldsymbol{h}$, up to some small residual *reconstruction error* σ^2. As shown by Tipping and Bishop (1999), probabilistic PCA becomes PCA as $\sigma \to 0$. In that case, the conditional expected value of $\boldsymbol{h}$ given $\boldsymbol{x}$ becomes an orthogonal projection onto the space spanned by the d columns of $\boldsymbol{W}$, like in PCA. See Section 17.1 for a discussion of the "inference" mechanism associated with PCA (probabilistic or not), i.e., recovering the expected value of the latent factors h_i given the observed input $\boldsymbol{x}$. That section also explains the very insightful *geometric and manifold interpretation* of PCA.

However, as $\sigma \to 0$, the density model becomes very sharp around these d dimensions spanned the columns of $\boldsymbol{W}$, as discussed in Section 17.1, which would not make it a very faithful model of the data, in general (not just because the data may live on a higher-dimensional manifold, but more importantly because the real data manifold may not be a flat hyperplane - see Chapter 17 for more).

15.6.1 ICA

Independent Component Analysis (ICA) is among the oldest representation learning algorithms (Herault and Ans, 1984; Jutten and Herault, 1991; Comon, 1994; Hyvärinen, 1999; Hyvärinen *et al.*, 2001). It is an approach to modeling linear factors that seeks non-Gaussian projections of the data. Like probabilistic PCA and factor analysis, it also fits the linear factor model of Eqs. 15.2 and 15.3. What is particular about ICA is that unlike PCA and factor analysis it *does not assume that the latent variable prior is Gaussian.* It only assumes that it is *factorized*, i.e.,

$$P(\boldsymbol{h}) = \prod_i P(h_i). \tag{15.4}$$

Since there is no parametric assumption behind the prior, we are really in front of a so-called *semi-parametric model*, with parts of the model being parametric ($P(\boldsymbol{x} \mid \boldsymbol{h})$) and parts being non-specified or non-parametric ($P(\boldsymbol{h})$). In fact, this typically yields to *non-Gaussian* priors: if the priors were Gaussian, then one could not distinguish between the factors $\boldsymbol{h}$ and a rotation of $\boldsymbol{h}$. Indeed, note

that if

$$\boldsymbol{h} = \boldsymbol{U}\boldsymbol{z}$$

with $\boldsymbol{U}$ an orthonormal (rotation) square matrix, i.e.,

$$\boldsymbol{z} = \boldsymbol{U}^\top \boldsymbol{h},$$

then, although $\boldsymbol{h}$ might have a Normal$(0, \boldsymbol{I})$ distribution, the $\boldsymbol{z}$ *also have a unit covariance*, i.e., they are uncorrelated:

$$Var[\boldsymbol{z}] = \mathbb{E}[\boldsymbol{z}\boldsymbol{z}^\top] = \mathbb{E}[\boldsymbol{U}^\top \boldsymbol{h}\boldsymbol{h}^\top \boldsymbol{U}] = \boldsymbol{U}^\top Var[\boldsymbol{h}]\boldsymbol{U} = \boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}.$$

In other words, imposing independence among Gaussian factors does not allow one to disentangle them, and we could as well recover any linear rotation of these factors. It means that, given the observed $\boldsymbol{x}$, even though we might assume the right generative model, PCA cannot recover the original generative factors. However, if we assume that the latent variables are *non-Gaussian*, then we can recover them, and this is what ICA is trying to achieve. In fact, under these generative model assumptions, the true underlying factors can be recovered (Comon, 1994). In fact, many ICA algorithms are looking for projections of the data $\boldsymbol{s} = \boldsymbol{V}\boldsymbol{x}$ such that they are *maximally non-Gaussian*. An intuitive explanation for these approaches is that although the true latent variables $\boldsymbol{h}$ may be non-Gaussian, almost any linear combination of them will look more Gaussian, because of the central limit theorem. Since linear combinations of the x_i's are also linear combinations of the h_j's, to recover the h_j's we just need to find the linear combinations that are maximally non-Gaussian (while keeping these different projections orthogonal to each other).

There is an interesting connection between ICA and sparsity, since the dominant form of non-Gaussianity in real data is due to sparsity, i.e., concentration of probability at or near 0. Non-Gaussian distributions typically have more mass around zero, although you can also get non-Gaussianity by increasing skewness, asymmetry, or kurtosis.

Like PCA can be generalized to non-linear auto-encoders described later in this chapter, ICA can be generalized to a non-linear generative model, e.g., $\boldsymbol{x} = f(\boldsymbol{h}) +$ noise. See Hyvärinen and Pajunen (1999) for the initial work on non-linear ICA and its successful use with ensemble learning by Roberts and Everson (2001); Lappalainen *et al.* (2000).

15.6.2 Sparse Coding as a Generative Model

One particularly interesting form of non-Gaussianity arises with distributions that are sparse. These typically have not just a peak at 0 but also a fat tail[4]. Like the

[4] with probability going to 0 as the values increase in magnitude at a rate that is slower than the Gaussian, i.e., less than quadratic in the log-domain.

other linear factor models (Eq. 15.3), sparse coding corresponds to a linear factor model, but one with a "sparse" latent variable $\boldsymbol{h}$, i.e., $P(\boldsymbol{h})$ puts high probability at or around 0. Unlike with ICA (previous section), the latent variable prior is parametric. For example the factorized Laplace density prior is

$$P(\boldsymbol{h}) = \prod_i P(h_i) = \prod_i \frac{\lambda}{2} e^{-\lambda |h_i|} \tag{15.5}$$

and the factorized Student-t prior is

$$P(\boldsymbol{h}) = \prod_i P(h_i) \propto \prod_i \frac{1}{1 + \frac{h_i^2}{\nu}}^{\frac{\nu+1}{2}}. \tag{15.6}$$

Both of these densities have a strong preference for near-zero values but, unlike the Gaussian, accomodate large values. In the standard sparse coding models, the reconstruction noise is assumed to be Gaussian, so that the corresponding reconstruction error is the squared error.

Regarding sparsity, note that the actual value $h_i = 0$ has zero measure under both densities, meaning that the posterior distribution $P(\boldsymbol{h} \mid \boldsymbol{x})$ will not generate values $\boldsymbol{h} = 0$. However, sparse coding is normally considered under a maximum a posteriori (MAP) inference framework, in which the inferred values of $\boldsymbol{h}$ are those that maximize the posterior, and these tend to often be zero if the prior is sufficiently concentrated around 0. The inferred values are those defined in Eq. 15.1, reproduced here,

$$\boldsymbol{h} = f(\boldsymbol{x}) = \arg\min_{\boldsymbol{h}} L(g(\boldsymbol{h}), \boldsymbol{x})) + \lambda \Omega(\boldsymbol{h})$$

where $L(g(\boldsymbol{h}), \boldsymbol{x})$ is interpreted as $-\log P(\boldsymbol{x} \mid g(\boldsymbol{h}))$ and $\Omega(\boldsymbol{h})$ as $-\log P(\boldsymbol{h})$. This MAP inference view of sparse coding and an interesting probabilistic interpretation of sparse coding are further discussed in Section 19.3.

To relate the generative model of sparse coding to ICA, note how the prior imposes not just sparsity but also independence of the latent variables h_i under $P(\boldsymbol{h})$, which may help to separate different explanatory factors, unlike PCA, factor analysis or probabilistic PCA, because these rely on a Gaussian prior, which yields a factorized prior under any rotation of the factors, multiplication by an orthonormal matrix, as demonstrated in Section 15.6.1.

See Section 17.2 about the manifold interpretation of sparse coding.

TODO: relate to and point to Spike-and-slab sparse coding (Goodfellow *et al.*, 2012) (section?)

15.7 Reconstruction Error as Log-Likelihood

Although traditional auto-encoders (like traditional neural networks) were introduced with an associated training loss, just like for neural networks, that training loss can generally be given a probabilistic interpretation as a conditional log-likelihood of the original input $\boldsymbol{x}$, given the reprensentation $\boldsymbol{h}$.

We have already covered negative log-likelihood as a loss function in general for feedforward neural networks in Section 6.3.2. Like prediction error for regular feedforward neural networks, reconstruction error for auto-encoders does not have to be squared error. When we view the loss as negative log-likelihood, we interpret the reconstruction error as

$$L = -\log P(\boldsymbol{x} \mid \boldsymbol{h})$$

where $\boldsymbol{h}$ is the representation, which may generally be obtained through an encoder taking $\boldsymbol{x}$ as input.

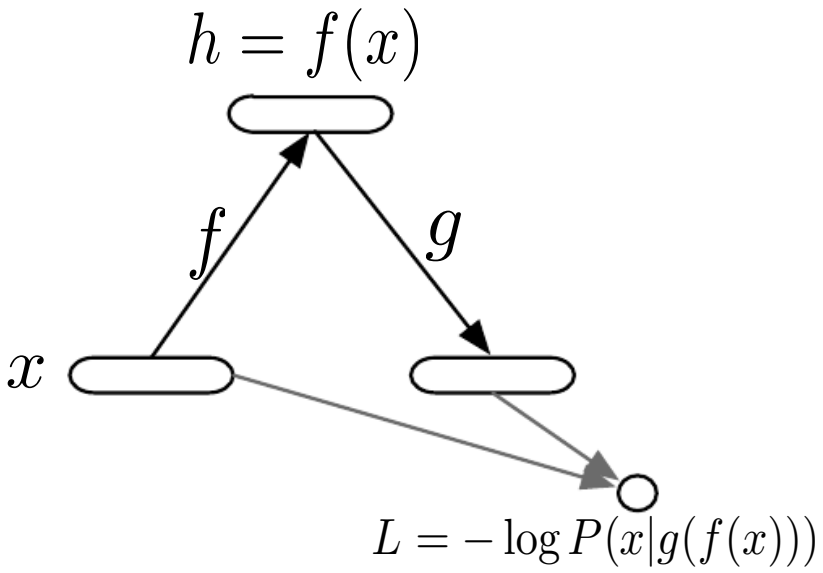

Figure 15.5: The computational graph of an auto-encoder, which is trained to maximize the probability assigned by the decoder g to the data point $\boldsymbol{x}$, given the output of the encoder $\boldsymbol{h} = f(\boldsymbol{x})$. The training objective is thus $L = -\log P(\boldsymbol{x} \mid g(f(\boldsymbol{x})))$, which ends up being squared reconstruction error if we choose a Gaussian reconstruction distribution with mean $g(f(\boldsymbol{x}))$, and cross-entropy if we choose a factorized Bernoulli reconstruction distribution with means $g(f(\boldsymbol{x}))$.

An advantage of this view is that it immediately tells us what kind of loss function one should use depending on the nature of the input. If the input is real-valued and unbounded, then squared error is a reasonable choice of reconstruction

error, and corresponds to $P(\boldsymbol{x} \mid \boldsymbol{h})$ being Normal. If the input is a vector of bits, then cross-entropy is a more reasonable choice, and corresponds to $P(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_i P(x_i \mid \boldsymbol{h})$ with $x_i \mid \boldsymbol{h}$ being Bernoulli-distributed. We then view the decoder $g(\boldsymbol{h})$ as computing the *parameters* of the reconstruction distribution, i.e., $P(\boldsymbol{x} \mid \boldsymbol{h}) = P(\boldsymbol{x} \mid g(\boldsymbol{h}))$.

Another advantage of this view is that we can think about the training of the decoder as estimating the conditional distribution $P(\boldsymbol{x} \mid \boldsymbol{h})$, which comes handy in the probabilistic interpretation of denoising auto-encoders, allowing us to talk about the distribution $P(\boldsymbol{x})$ explicitly or implicitly represented by the auto-encoder (see Sections 15.9, 20.9.3 and 20.11 for more details). In the same spirit, we can rethink the notion of encoder from a simple function to a conditional distribution $Q(\boldsymbol{h} \mid \boldsymbol{x})$, with a special case being when $Q(\boldsymbol{h} \mid \boldsymbol{x})$ is a Dirac at some particular value. Equivalently, thinking about the encoder as a distribution corresponds to *injecting noise* inside the auto-encoder. This view is developed further in Sections 20.9.3 and 20.12.

15.8 Sparse Representations

Sparse auto-encoders are auto-encoders which learn a sparse representation, i.e., one whose elements are often either zero or close to zero. Sparse coding was introduced in Section 15.6.2 as a linear factor model in which the prior $P(\boldsymbol{h})$ on the representation $\boldsymbol{h} = f(\boldsymbol{x})$ encourages values at or near 0. In Section 15.8.1, we see how ordinary auto-encoders can be prevented from learning a useless identity transformation by using a sparsity penalty rather than a bottleneck. The main difference between a sparse auto-encoder and sparse coding is that sparse coding has no explicit parametric encoder, whereas sparse auto-encoders have one. The "encoder" of sparse coding is the algorithm that performs the approximate inference, i.e., looks for

$$h^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{h}} \log P(\boldsymbol{h} \mid \boldsymbol{x}) = \arg\min_{\boldsymbol{h}} \frac{||\boldsymbol{x} - (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h})||^2}{\sigma^2} - \log P(\boldsymbol{h}) \quad (15.7)$$

where σ^2 is a reconstruction variance parameter (which should equal the average squared reconstruction error[5]), and $P(\boldsymbol{h})$ is a "sparse" prior that puts more probability mass around $\boldsymbol{h} = 0$, such as the Laplacian prior, with factorized marginals

$$P(h_i) = \frac{\lambda}{2} e^{\lambda |h_i|} \quad (15.8)$$

[5] but can be lumped into the regularizer λ which controls the strength of the sparsity prior, defined in Eq. 15.8, for example.

or the Student-t prior, with factorized marginals

$$P(h_i) \propto \frac{1}{(1 + \frac{h_i^2}{\nu})^{\frac{\nu+1}{2}}}. \tag{15.9}$$

The advantages of such a non-parametric encoder and the sparse coding approach over sparse auto-encoders are that

1. it can in principle minimize the combination of reconstruction error and log-prior better than any parametric encoder,

2. it performs what is called *explaining away* (see Figure 13.8), i.e., it allows to "choose" some "explanations" (hidden factors) and inhibits the others.

The disadvantages are that

1. computing time for encoding the given input $\boldsymbol{x}$, i.e., performing inference (computing the representation $\boldsymbol{h}$ that goes with the given $\boldsymbol{x}$) can be substantially larger than with a parametric encoder (because an optimization must be performed *for each example* $\boldsymbol{x}$), and

2. the resulting encoder function could be non-smooth and possibly too non-linear (with two nearby $\boldsymbol{x}$'s being associated with very different $\boldsymbol{h}$'s), potentially making it more difficult for the downstream layers to properly generalize.

In Section 15.8.2, we describe PSD (Predictive Sparse Decomposition), which combines a non-parametric encoder (as in sparse coding, with the representation obtained via an optimization) and a parametric encoder (like in the sparse auto-encoder). Section 15.9 introduces the Denoising Auto-Encoder (DAE), which puts pressure on the representation by requiring it to extract information about the underlying distribution and where it concentrates, so as to be able to denoise a corrupted input. Section 15.10 describes the Contractive Auto-Encoder (CAE), which optimizes an explicit regularization penalty that aims at making the representation as insensitive as possible to the input, while keeping the information sufficient to reconstruct the training examples.

15.8.1 Sparse Auto-Encoders

A sparse auto-encoder is simply an auto-encoder whose training criterion involves a sparsity penalty $\Omega(\boldsymbol{h})$ in addition to the reconstruction error:

$$L = -\log P(\boldsymbol{x} \mid g(\boldsymbol{h})) + \Omega(\boldsymbol{h}) \tag{15.10}$$

where $g(\boldsymbol{h})$ is the decoder output and typically we have $\boldsymbol{h} = f(\boldsymbol{x})$, the encoder output.

We can think of that penalty $\Omega(\boldsymbol{h})$ simply as a regularizer or as a log-prior on the representations $\boldsymbol{h}$. For example, the sparsity penalty corresponding to the Laplace prior ($\frac{\lambda}{2} e^{-\lambda|h_i|}$) is the absolute value sparsity penalty (see also Eq. 15.8 above):

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|$$
$$-\log P(\boldsymbol{h}) = \sum_i \log \frac{\lambda}{2} + \lambda|h_i| = \text{const} + \Omega(\boldsymbol{h}) \tag{15.11}$$

where the constant term depends only of λ and not $\boldsymbol{h}$ (which we typically ignore in the training criterion because we consider λ as a hyperparameter rather than a parameter). Similarly (as per Eq. 15.9), the sparsity penalty corresponding to the Student-t prior (Olshausen and Field, 1997) is

$$\Omega(\boldsymbol{h}) = \sum_i \frac{\nu + 1}{2} \log(1 + \frac{h_i^2}{\nu}) \tag{15.12}$$

where ν is considered to be a hyperparameter.

The early work on sparse auto-encoders (Ranzato *et al.*, 2007a, 2008) considered various forms of sparsity and proposed a connection between sparsity regularization and the partition function gradient in energy-based models (see Section TODO). The idea is that a regularizer such as sparsity makes it difficult for an auto-encoder to achieve zero reconstruction error everywhere. If we consider reconstruction error as a proxy for energy (unnormalized log-probability of the data), then minimizing the training set reconstruction error forces the energy to be low on training examples, while the regularizer prevents it from being low everywhere. The same role is played by the gradient of the partition function in energy-based models such as the RBM (Section TODO).

However, the sparsity penalty of sparse auto-encoders does not need to have a probabilistic interpretation. For example, Goodfellow *et al.* (2009) successfully used the following sparsity penalty, which does not try to bring h_i all the way down to 0, but only towards some low target value such as $\rho = 0.05$.

$$\Omega(\boldsymbol{h}) = \sum_i \rho \log h_i + (1 - \rho) \log(1 - h_i) \tag{15.13}$$

where $0 < h_i < 1$, usually with $h_i = \text{sigmoid}(a_i)$. This is just the cross-entropy between the Bernoulli distributions with probability $p = h_i$ and the target Bernoulli distribution with probability $p = \rho$.

One way to achieve *actual zeros* in $\boldsymbol{h}$ for sparse (and denoising) auto-encoders was introduced in Glorot *et al.* (2011c). The idea is to use a half-rectifier (a.k.a. simply as "rectifier") or ReLU (Rectified Linear Unit, introduced in Glorot *et al.* (2011b) for deep supervised networks and earlier in Nair and Hinton (2010a) in the context of RBMs) as the output non-linearity of the encoder. With a prior that actually pushes the representations to zero (like the absolute value penalty), one can thus indirectly control the average number of zeros in the representation. ReLUs were first successfully used for *deep feedforward networks* in Glorot *et al.* (2011a), achieving for the first time the ability to *train fairly deep supervised networks without the need for unsupervised pre-training*, and this turned out to be an important component in the 2012 object recognition breakthrough with deep convolutional networks (Krizhevsky *et al.*, 2012b).

Interestingly, the "regularizer" used in sparse auto-encoders does not conform to the classical interpretation of regularizers as priors on the parameters. That classical interpretation of the regularizer comes from the MAP (Maximum A Posteriori) point estimation (see Section 5.5.1) of parameters associated with the Bayesian view of parameters as random variables and considering the joint distribution of data $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$ (see Section 5.7):

$$\arg\max_{\boldsymbol{\theta}} P(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \left(\log P(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log P(\boldsymbol{\theta})\right)$$

where the first term on the right is the usual data log-likelihood term and the second term, the log-prior over parameters, incorporates the preference over particular values of $\boldsymbol{\theta}$.

With regularized auto-encoders such as sparse auto-encoders and contractive auto-encoders, instead, the regularizer corresponds to a *log-prior over the representation, or over latent variables*. In the case of sparse auto-encoders, predictive sparse decomposition and contractive auto-encoders, the regularizer specifies a *preference over functions* of the data, rather than over parameters. This makes such a regularizer *data-dependent*, unlike the classical parameter log-prior. Specifically, in the case of the sparse auto-encoder, it says that we prefer an encoder whose output produces values closer to 0. Indirectly (when we marginalize over the training distribution), this is also indicating a preference over parameters, of course.

15.8.2 Predictive Sparse Decomposition

TODO: we have too many forward refs to this section. There are 150 lines about PSD in this section and at least 20 lines of forward references to this section in this chapter, some of which are just 100 lines away. Predictive sparse decomposition (PSD) is a variant that combines sparse coding and a parametric

encoder (Kavukcuoglu *et al.*, 2008b), i.e., it has both a parametric encoder and iterative inference. It has been applied to unsupervised feature learning for object recognition in images and video (Kavukcuoglu *et al.*, 2009, 2010b; Jarrett *et al.*, 2009a; Farabet *et al.*, 2011), as well as for audio (Henaff *et al.*, 2011). The representation is considered to be a free variable (possibly a latent variable if we choose a probabilistic interpretation) and the training criterion combines a sparse coding criterion with a term that encourages the optimized sparse representation $\boldsymbol{h}$ (after inference) to be close to the output of the encoder $f(\boldsymbol{x})$:

$$L = \arg\min_{\boldsymbol{h}} \left(||\boldsymbol{x} - g(\boldsymbol{h})||^2 + \lambda |\boldsymbol{h}|_1 + \gamma ||\boldsymbol{h} - f(\boldsymbol{x})||^2 \right) \tag{15.14}$$

where f is the encoder and g is the decoder. Like in sparse coding, for each example $\boldsymbol{x}$ an iterative optimization is performed in order to obtain a representation $\boldsymbol{h}$. However, because the iterations can be initialized from the output of the encoder, i.e., with $\boldsymbol{h} = f(\boldsymbol{x})$, only a few steps (e.g. 10) are necessary to obtain good results. Simple gradient descent on $\boldsymbol{h}$ has been used by the authors. After $\boldsymbol{h}$ is settled, both g and f are updated towards minimizing the above criterion. The first two terms are the same as in L1 sparse coding while the third one encourages f to predict the outcome of the sparse coding optimization, making it a better choice for the initialization of the iterative optimization. Hence f can be used as a parametric approximation to the non-parametric encoder implicitly defined by sparse coding. It is one of the first instances of *learned approximate inference* (see also Sec. 19.8). Note that this is different from separately doing sparse coding (i.e., training g) and then training an approximate inference mechanism f, since both the encoder and decoder are trained together to be "compatible" with each other. Hence the decoder will be learned in such a way that inference will tend to find solutions that can be well approximated by the approximate inference. TODO: this is probably too much forward reference, when we bring these things in we can remind people that they resemble PSD, but it doesn't really help the reader to say that the thing we are describing now is similar to things they haven't seen yet A similar example is the variational auto-encoder, in which the encoder acts as approximate inference for the decoder, and both are trained jointly (Section 20.9.3). See also Section 20.9.4 for a probabilistic interpretation of PSD in terms of a variational lower bound on the log-likelihood.

In practical applications of PSD, the iterative optimization is only used during training, and f is used to compute the learned features. It makes computation fast at recognition time and also makes it easy to use the trained features f as initialization (unsupervised pre-training) for the lower layers of a deep net. Like other unsupervised feature learning schemes, PSD can be stacked greedily, e.g., training a second PSD on top of the features extracted by the first one, etc.

15.9 Denoising Auto-Encoders

The Denoising Auto-Encoder (DAE) was first proposed (Vincent *et al.*, 2008, 2010) as a means of forcing an auto-encoder to learn to capture the data distribution without an explicit constraint on either the dimension or the sparsity of the learned representation. It was motivated by the idea that in order to fully capture a complex distribution, an auto-encoder needs to have at least as many hidden units as needed by the complexity of that distribution. Hence its dimensionality should not be restricted to the input dimension.

The principle of the denoising auto-encoder is deceptively simple and illustrated in Figure 15.6: the encoder sees as input a corrupted version of the input, but the decoder tries to reconstruct the clean uncorrupted input.

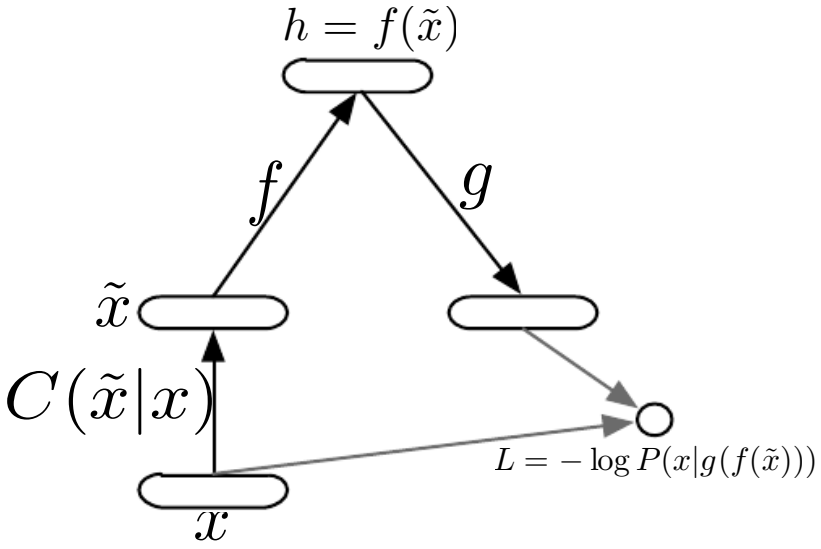

Figure 15.6: The computational graph of a denoising auto-encoder, which is trained to reconstruct the clean data point $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, i.e., to minimize the loss $L = -\log P(\boldsymbol{x} \mid g(f(\tilde{\boldsymbol{x}})))$, where $\tilde{\boldsymbol{x}}$ is a corrupted version of the data example $\boldsymbol{x}$, obtained through a given corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.

Mathematically, and following the notations used in this chapter, this can be formalized as follows. We introduce a corruption process $C(\tilde{\mathbf{x}} \mid \mathbf{x})$ which represents a conditional distribution over corrupted samples $\tilde{\mathbf{x}}$, given a data sample $\mathbf{x}$. The auto-encoder then learns a *reconstruction distribution* $P(\mathbf{x} \mid \tilde{\mathbf{x}})$ estimated from training pairs $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$, as follows:

1. Sample a training example $\mathbf{x} = \boldsymbol{x}$ from the data generating distribution (the training set).

2. Sample a corrupted version $\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}$ from the conditional distribution $C(\tilde{}$

$\mathbf{x} = \boldsymbol{x}$).

3. Use $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ as a training example for estimating the auto-encoder reconstruction distribution $P(\boldsymbol{x} \mid \tilde{\boldsymbol{x}}) = P(\boldsymbol{x} \mid g(\boldsymbol{h}))$ with $\boldsymbol{h}$ the output of encoder $f(\tilde{\boldsymbol{x}})$ and $g(\boldsymbol{h})$ the output of the decoder.

Typically we can simply perform gradient-based approximate minimization (such as minibatch gradient descent) on the negative log-likelihood $-\log P(\boldsymbol{x} \mid \boldsymbol{h})$, i.e., the denoising reconstruction error, using back-propagation to compute gradients, just like for regular feedforward neural networks (the only difference being the corruption of the input and the choice of target output).

We can view this training objective as performing stochastic gradient descent on the denoising reconstruction error, but where the "noise" now has two sources:

1. the choice of training sample $\boldsymbol{x}$ from the data set, and

2. the random corruption applied to $\boldsymbol{x}$ to obtain $\tilde{\boldsymbol{x}}$.

We can therefore consider that the DAE is performing stochastic gradient descent on the following expectation:

$$-E_{\boldsymbol{x} \sim Q(\mathbf{x})} E_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \boldsymbol{x})} \log P(\boldsymbol{x} \mid g(f(\tilde{\boldsymbol{x}})))$$

where $Q(\mathbf{x})$ is the training distribution.

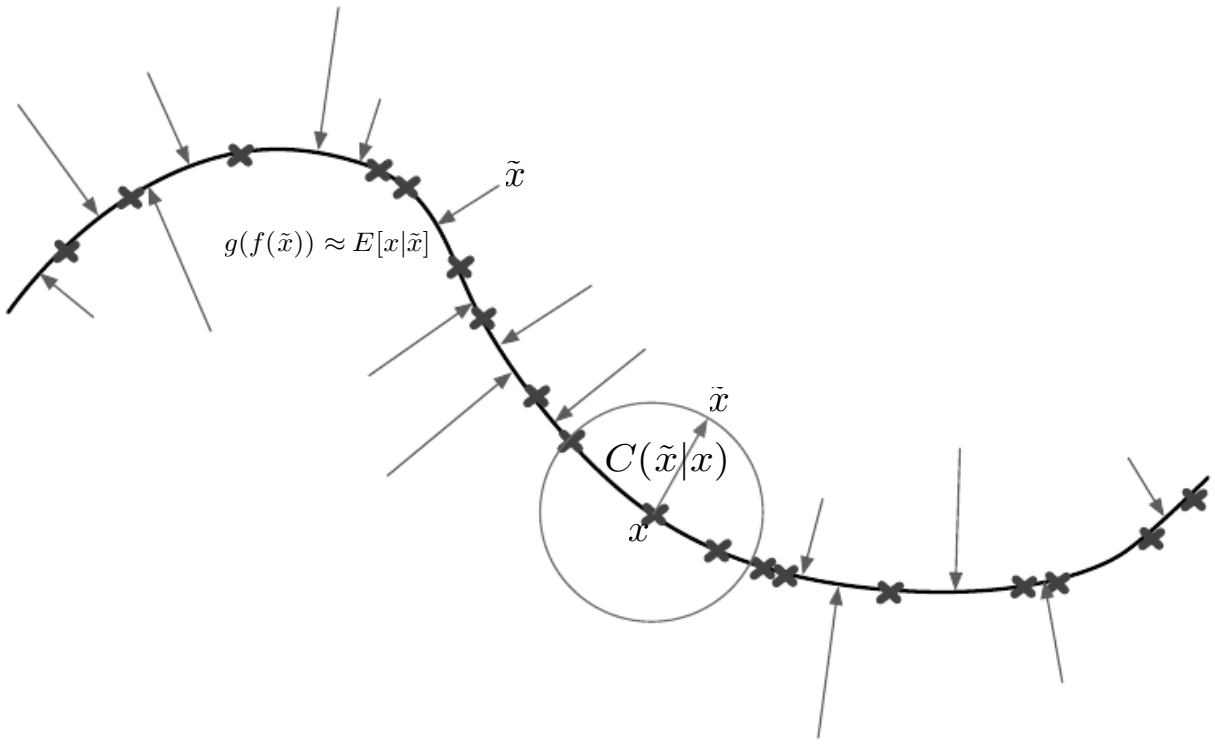

Figure 15.7: A denoising auto-encoder is trained to reconstruct the clean data point $\boldsymbol{x}$ from it corrupted version $\tilde{\boldsymbol{x}}$. In the figure, we illustrate the corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ by a grey circle of equiprobable corruptions, and grey arrow for the corruption process) acting on examples $\boldsymbol{x}$ (red crosses) lying near a low-dimensional manifold near which probability concentrates. When the denoising auto-encoder is trained to minimize the average of squared errors $||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2$, the reconstruction $g(f(\tilde{\boldsymbol{x}}))$ estimates $\mathbb{E}[\mathbf{x} \mid \tilde{\boldsymbol{x}}]$, which approximately points orthogonally towards the manifold, since it estimates the center of mass of the clean points $\boldsymbol{x}$ which could have given rise to $\tilde{\boldsymbol{x}}$. The auto-encoder thus learns a vector field $g(f(\boldsymbol{x})) - \boldsymbol{x}$ (the green arrows) and it turns out that this vector field estimates the gradient field $\frac{\partial \log Q(\boldsymbol{x})}{\partial \boldsymbol{x}}$ (up to a multiplicative factor that is the average root mean square reconstruction error), where Q is the unknown data generating distribution.

15.9.1 Learning a Vector Field that Estimates a Gradient Field

As illustrated in Figure 15.7, a very important property of DAEs is that their training criterion makes the auto-encoder learn a vector field $(g(f(\boldsymbol{x})) - \boldsymbol{x})$ that estimates the gradient field (or *score*) $\frac{\partial \log Q(\boldsymbol{x})}{\partial x}$, as per Eq. 15.15. A first result in this direction was proven by Vincent (2011a), showing that minimizing squared reconstruction error in a denoising auto-encoder with Gaussian noise was related to *score matching* (Hyvärinen, 2005a), making the denoising criterion a regularized form of score matching called *denoising score matching* (Kingma and LeCun, 2010a). Score matching is an alternative to maximum likelihood and provides a consistent estimator. It is discussed further in Section 18.4. The denoising version

is discussed in Section 18.5.

The connection between denoising auto-encoders and score matching was first made (Vincent, 2011a) in the case where the denoising auto-encoder has a particular parametrization (one hidden layer, sigmoid activation functions on hidden units, linear reconstruction), in which case the denoising criterion actually corresponds to a regularized form of score matching on a Gaussian RBM (with binomial hidden units and Gaussian visible units). The connection between ordinary auto-encoders and Gaussian RBMs had previously been made by Bengio and Delalleau (2009), which showed that contrastive divergence training of RBMs was related to an associated auto-encoder gradient, and later by Swersky (2010), which showed that non-denoising reconstruction error corresponded to score matching plus a regularizer.

The fact that the denoising criterion yields an estimator of the score for general encoder/decoder parametrizations has been proven (Alain and Bengio, 2012, 2013) in the case where the corruption and the reconstruction distributions are Gaussian (and of course $\boldsymbol{x}$ is continuous-valued), i.e., with the squared error denoising error

$$||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2$$

and corruption

$$C(\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}|\boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}; \mu = \boldsymbol{x}, \Sigma = \sigma^2 I)$$

with noise variance σ^2.

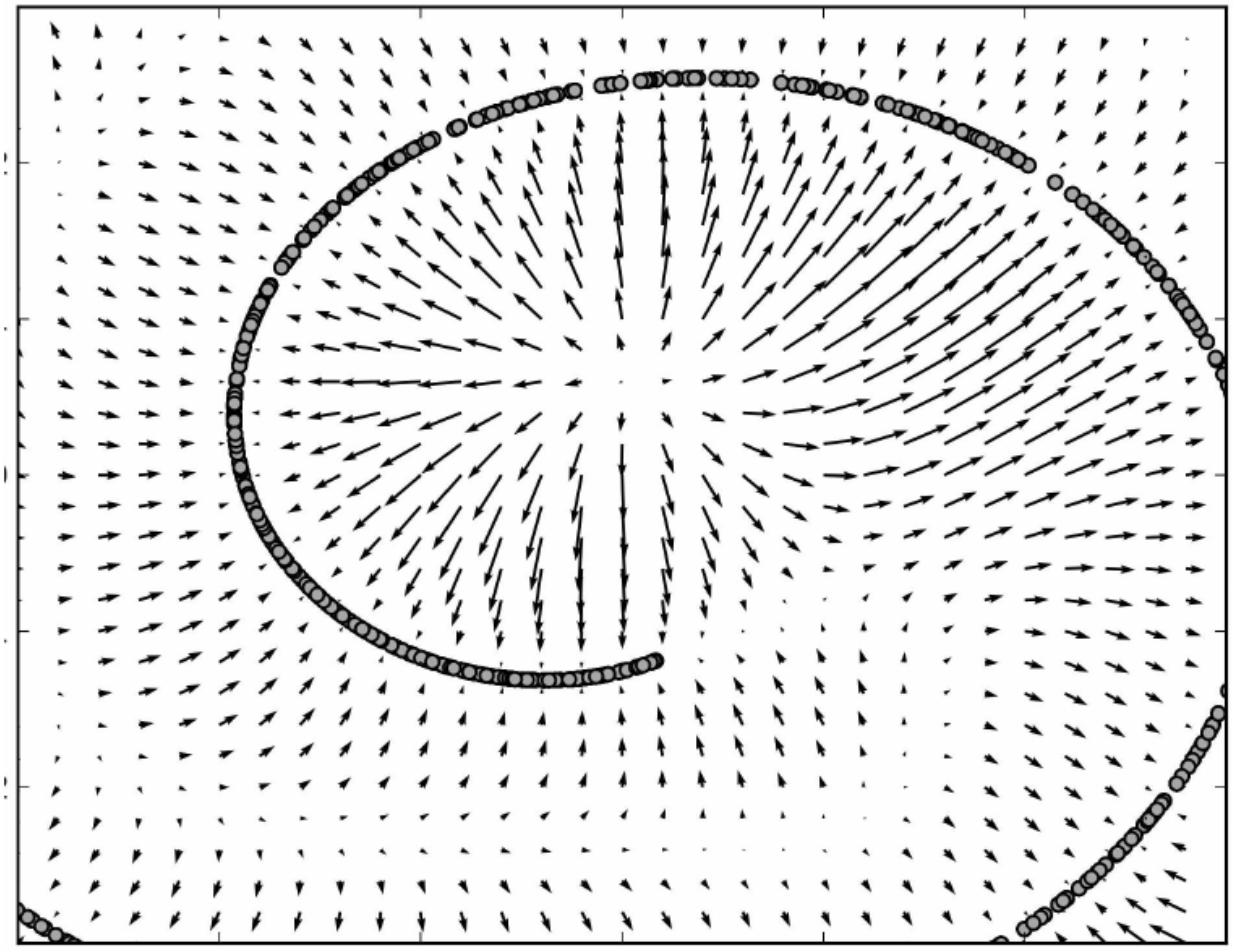

Figure 15.8: Vector field learned by a denoising auto-encoder around a 1-D curved manifold near which the data (orange circles) concentrates in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the auto-encoder and points towards higher probability according to the implicitly estimated probability distribution. Note that the vector field has zeros at both peaks of the estimated density function (on the data manifolds) and at troughs (local minima) of that density function, e.g., on the curve that separates different arms of the spiral or in the middle of it.

More precisely, the main theorem states that $\frac{g(f(\boldsymbol{x}))-\boldsymbol{x}}{\sigma^2}$ is a consistent estimator of $\frac{\partial \log Q(\boldsymbol{x})}{\partial \boldsymbol{x}}$, where $Q(\boldsymbol{x})$ is the data generating distribution,

$$\frac{g(f(\boldsymbol{x})) - \boldsymbol{x}}{\sigma^2} \rightarrow \frac{\partial \log Q(\boldsymbol{x})}{\partial \boldsymbol{x}}, \tag{15.15}$$

so long as f and g have sufficient capacity to represent the true score (and assuming that the expected training criterion can be minimized, as usual when proving consistency associated with a training objective).

Note that in general, there is no guarantee that the reconstruction $g(f(\boldsymbol{x}))$ minus the input $\boldsymbol{x}$ corresponds to the gradient of something (the estimated score should be the gradient of the estimated log-density with respect to the input

$\boldsymbol{x}$). That is why the early results (Vincent, 2011a) are specialized to particular parametrizations where $g(f(\boldsymbol{x})) - \boldsymbol{x}$ is the derivative of something. See a more general treatment by Kamyshanska and Memisevic (2015).

Although it was intuitively appealing that in order to denoise correctly one must capture the training distribution, the above consistency result makes it mathematically very clear in what sense the DAE is capturing the input distribution: it is estimating the gradient of its energy function (i.e., of its log-density), i.e., learning to point towards more probable (lower energy) configurations. Figure 15.8 (see details of experiment in Alain and Bengio (2013)) illustrates this. Note how the norm of reconstruction error (i.e. the norm of the vectors shown in the figure) is related to but *different* from the energy (unnormalized log-density) associated with the estimated model. The energy should be low only where the probability is high. The reconstruction error (norm of the estimated score vector) is low where probability is near a peak of probability (or a trough of energy), but it can also be low at *maxima* of energy (minima of probability).

Section 20.11 continues the discussion of the relationship between denoising auto-encoders and probabilistic modeling by showing how one can *generate* from the distribution implicitly estimated by a denoising auto-encoder. Whereas (Alain and Bengio, 2013) generalized the score estimation result of Vincent (2011a) to arbitrary parametrizations, the result from Bengio *et al.* (2013b), discussed in Section 20.11, provides a probabilistic – and in fact generative – interpretation to every denoising auto-encoder.

15.10 Contractive Auto-Encoders

The Contractive Auto-Encoder or CAE (Rifai *et al.*, 2011a,c) introduces an explicit regularizer on the code $\boldsymbol{h} = f(\boldsymbol{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\boldsymbol{h}) = \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 \tag{15.16}$$

which is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function. Whereas the denoising auto-encoder learns to contract the reconstruction function (the composition of the encoder and decoder), the CAE learns to specifically contract the encoder. See Figure 17.13 for a view of how contraction near the data points makes the auto-encoder capture the manifold structure.

If it weren't for the opposing force of reconstruction error, which attempts to make the code $\boldsymbol{h}$ keep all the information necessary to *reconstruct training examples*, the CAE penalty would yield a code $\boldsymbol{h}$ that is constant and does not

depend on the input $\boldsymbol{x}$. The compromise between these two forces yields an auto-encoder whose derivatives $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ are tiny in most directions, except those that are needed to reconstruct training examples, i.e., the directions that are tangent to the manifold near which data concentrate. Indeed, in order to distinguish (and thus, reconstruct correctly) two nearby examples on the manifold, one must assign them a different code, i.e., $f(\boldsymbol{x})$ must vary as $\boldsymbol{x}$ moves from one to the other, i.e., in the direction of a tangent to the manifold.

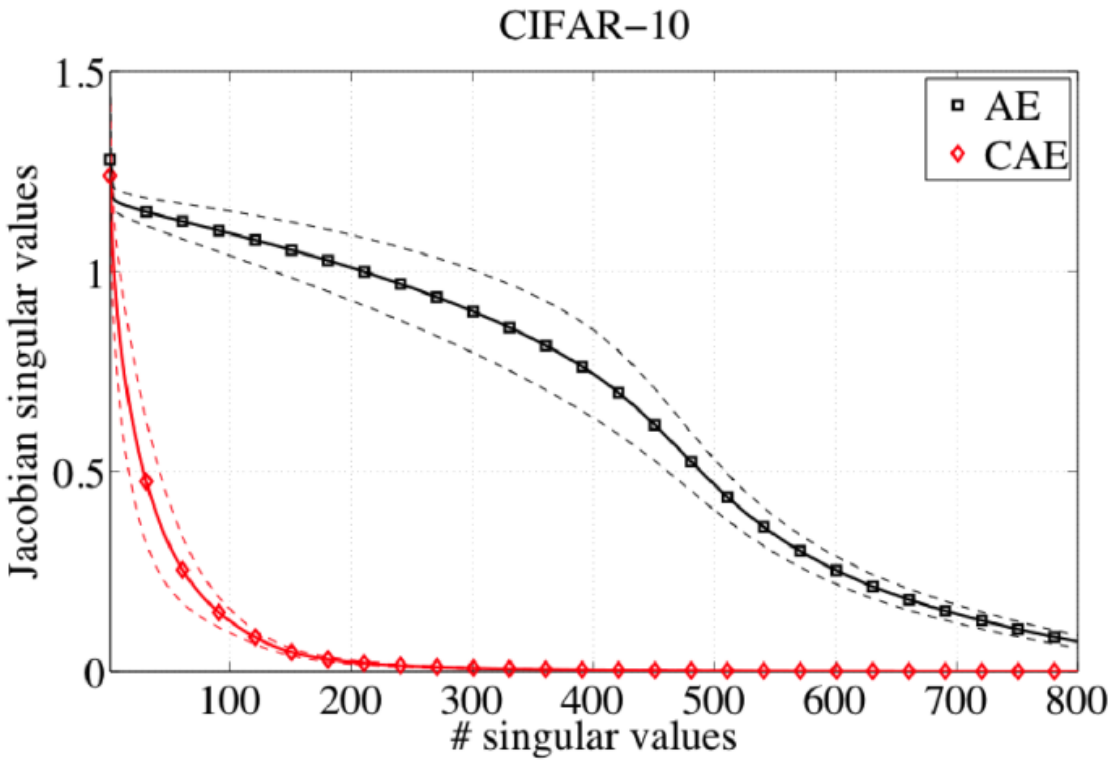

Figure 15.9: Average (over test examples) of the singular value spectrum of the Jacobian matrix $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ for the encoder f learned by a regular auto-encoder (AE) versus a contractive auto-encoder (CAE). This illustrates how the contractive regularizer yields a smaller set of directions in input space (those corresponding to large singular value of the Jacobian) which provoke a response in the representation $\boldsymbol{h}$ while the representation remains almost insensitive for most directions of change in the input.

What is interesting is that this penalty forces more strongly the representation to be invariant in directions orthogonal to the manifold. This can be seen clearly by comparing the singular value spectrum of the Jacobian $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ for different auto-encoders, as shown in Figure 15.9. We see that the CAE manages to concentrate the sensitivity of the representation in fewer dimensions than a regular (or sparse) auto-encoder. Figure 17.3 illustrates tangent vectors obtained by a CAE on the MNIST digits dataset, showing that the leading tangent vectors correspond to small deformations such as translation. More impressively, Figure 15.10 shows

tangent vectors learned on 32×32 color (RGB) CIFAR-10 images by a CAE,

compared to the tangent vectors by a non-distributed representation learner (a mixture of local PCAs).

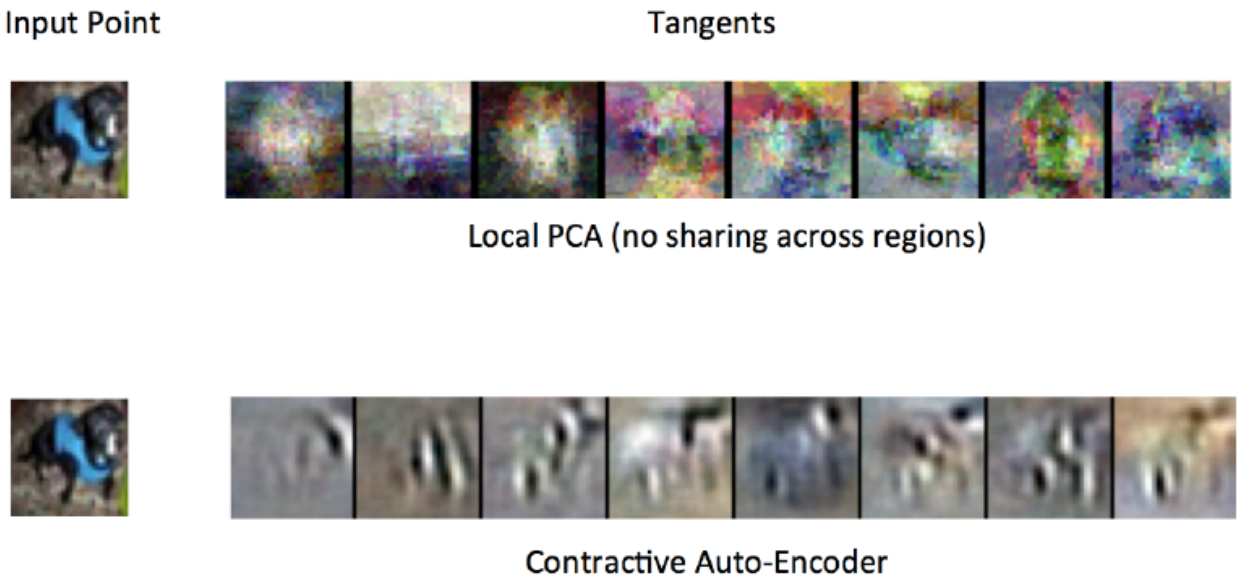

Figure 15.10: Illustration of tangent vectors (bottom) of the manifold estimated by a contractive auto-encoder (CAE), at some input point (left, CIFAR-10 image of a dog). See also Fig. 17.3. Each image on the right corresponds to a tangent vector, either estimated by a local PCA (equivalent to a Gaussian mixture), top, or by a CAE (bottom). The tangent vectors are estimated by the leading singular vectors of the Jacobian matrix $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ of the input-to-code mappiing. Although both local PCA and CAE can capture local tangents that are different in different points, the local PCA does not have enough training data to meaningful capture good tangent directions, whereas the CAE does (because it exploits parameter sharing across different locations that share a subset of active hidden units). The CAE tangent directions typically correspond to moving or changing parts of the object (such as the head or legs), which corresponds to plausible changes in the input image.

One practical issue with the CAE regularization criterion is that although it is cheap to compute in the case of a single hidden layer auto-encoder, it becomes much more expensive in the case of deeper auto-encoders. The strategy followed by Rifai *et al.* (2011a) is to separately pre-train each single-layer auto-encoder stacked to form a deeper auto-encoder. However, a deeper encoder could be advantageous in spite of the computational overhead, as argued by Schulz and Behnke (2012).

Another practical issue is that the contraction penalty on the encoder f could yield useless results if the decoder g would exactly compensate (e.g. by being scaled up by exactly the same amount as f is scaled down). In Rifai *et al.* (2011a), this is compensated by tying the weights of f and g, both being of the form of an affine transformation followed by a non-linearity (e.g. sigmoid), i.e., the weights of g and the transpose of the weights of f.