Chapter 14

Autoencoders

An **autoencoder** is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer $\boldsymbol{h}$ that describes a **code** used to represent the input. The network may be viewed as consisting of two parts: an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces a reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$. This architecture is presented in figure 14.1. If an autoencoder succeeds in simply learning to set $g(f(\boldsymbol{x})) = \boldsymbol{x}$ everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$.

The idea of autoencoders has been part of the historical landscape of neural networks for decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994). Traditionally, autoencoders were used for dimensionality reduction or feature learning. Recently, theoretical connections between autoencoders and latent variable models have brought autoencoders to the forefront of generative modeling, as we will see in chapter 20. Autoencoders may be thought of as being a special case of feedforward networks and may be trained with all the same techniques, typically minibatch gradient descent following gradients computed by back-propagation. Unlike general feedforward networks, autoencoders may also be trained using **recirculation** (Hinton and McClelland, 1988), a learning algorithm based on comparing the activations of the network on the original input

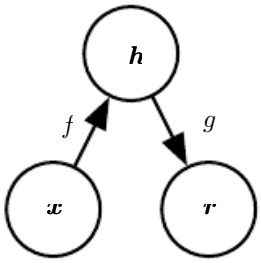

Figure 14.1: The general structure of an autoencoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The autoencoder has two components: the encoder f (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder g (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

to the activations on the reconstructed input. Recirculation is regarded as more biologically plausible than back-propagation but is rarely used for machine learning applications.

14.1 Undercomplete Autoencoders

Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in $\boldsymbol{h}$ taking on useful properties.

One way to obtain useful features from the autoencoder is to constrain $\boldsymbol{h}$ to have a smaller dimension than $\boldsymbol{x}$. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))), \tag{14.1}$$

where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as the mean squared error.

When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has learned the principal subspace of the training data as a side effect.

Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA. Unfortu-

nately, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data. Theoretically, one could imagine that an autoencoder with a one-dimensional code but a very powerful nonlinear encoder could learn to represent each training example $\boldsymbol{x}^{(i)}$ with the code i. The decoder could learn to map these integer indices back to the values of specific training examples. This specific scenario does not occur in practice, but it illustrates clearly that an autoencoder trained to perform the copying task can fail to learn anything useful about the dataset if the capacity of the autoencoder is allowed to become too great.

14.2 Regularized Autoencoders

Undercomplete autoencoders, with code dimension less than the input dimension, can learn the most salient features of the data distribution. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity.

A similar problem occurs if the hidden code is allowed to have dimension equal to the input, and in the **overcomplete** case in which the hidden code has dimension greater than the input. In these cases, even a linear encoder and a linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled. Regularized autoencoders provide the ability to do so. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. These other properties include sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs. A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn a trivial identity function.

In addition to the methods described here, which are most naturally interpreted as regularized autoencoders, nearly any generative model with latent variables and equipped with an inference procedure (for computing latent representations given input) may be viewed as a particular form of autoencoder. Two generative modeling approaches that emphasize this connection with autoencoders are the descendants of the Helmholtz machine (Hinton *et al.*, 1995b), such as the variational

autoencoder (section 20.10.3) and the generative stochastic networks (section 20.12). These models naturally learn high-capacity, overcomplete encodings of the input and do not require regularization for these encodings to be useful. Their encodings are naturally useful because the models were trained to approximately maximize the probability of the training data rather than to copy the input to the output.

14.2.1 Sparse Autoencoders

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty $\Omega(\boldsymbol{h})$ on the code layer $\boldsymbol{h}$, in addition to the reconstruction error:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}), \tag{14.2}$$

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

Sparse autoencoders are typically used to learn features for another task, such as classification. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a byproduct.

We can think of the penalty $\Omega(\boldsymbol{h})$ simply as a regularizer term added to a feedforward network whose primary task is to copy the input to the output (unsupervised learning objective) and possibly also perform some supervised task (with a supervised learning objective) that depends on these sparse features. Unlike other regularizers, such as weight decay, there is not a straightforward Bayesian interpretation to this regularizer. As described in section 5.6.1, training with weight decay and other regularization penalties can be interpreted as a MAP approximation to Bayesian inference, with the added regularizing penalty corresponding to a prior probability distribution over the model parameters. In this view, regularized maximum likelihood corresponds to maximizing $p(\boldsymbol{\theta} \mid \boldsymbol{x})$, which is equivalent to maximizing $\log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$. The $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$ term is the usual data log-likelihood term, and the $\log p(\boldsymbol{\theta})$ term, the log-prior over parameters, incorporates the preference over particular values of $\boldsymbol{\theta}$. This view is described in section 5.6. Regularized autoencoders defy such an interpretation because the regularizer depends on the data and is therefore by definition not a prior in the formal sense of the word. We can still think of these regularization terms as implicitly expressing a preference over functions.

Rather than thinking of the sparsity penalty as a regularizer for the copying task, we can think of the entire sparse autoencoder framework as approximating

maximum likelihood training of a generative model that has latent variables. Suppose we have a model with visible variables $\boldsymbol{x}$ and latent variables $\boldsymbol{h}$, with an explicit joint distribution $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{h}) p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. We refer to $p_{\text{model}}(\boldsymbol{h})$ as the model's prior distribution over the latent variables, representing the model's beliefs prior to seeing $\boldsymbol{x}$. This is different from the way we have previously used the word "prior," to refer to the distribution $p(\boldsymbol{\theta})$ encoding our beliefs about the model's parameters before we have seen the training data. The log-likelihood can be decomposed as

$$\log p_{\text{model}}(\boldsymbol{x}) = \log \sum_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}). \tag{14.3}$$

We can think of the autoencoder as approximating this sum with a point estimate for just one highly likely value for $\boldsymbol{h}$. This is similar to the sparse coding generative model (section 13.4), but with $\boldsymbol{h}$ being the output of the parametric encoder rather than the result of an optimization that infers the most likely $\boldsymbol{h}$. From this point of view, with this chosen $\boldsymbol{h}$, we are maximizing

$$\log p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}) = \log p_{\text{model}}(\boldsymbol{h}) + \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.4}$$

The $\log p_{\text{model}}(\boldsymbol{h})$ term can be sparsity inducing. For example, the Laplace prior,

$$p_{\text{model}}(h_i) = \frac{\lambda}{2} e^{-\lambda |h_i|}, \tag{14.5}$$

corresponds to an absolute value sparsity penalty. Expressing the log-prior as an absolute value penalty, we obtain

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|, \tag{14.6}$$

$$-\log p_{\text{model}}(\boldsymbol{h}) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2} \right) = \Omega(\boldsymbol{h}) + \text{const}, \tag{14.7}$$

where the constant term depends only on λ and not $\boldsymbol{h}$. We typically treat λ as a hyperparameter and discard the constant term since it does not affect the parameter learning. Other priors, such as the Student t prior, can also induce sparsity. From this point of view of sparsity as resulting from the effect of $p_{\text{model}}(\boldsymbol{h})$ on approximate maximum likelihood learning, the sparsity penalty is not a regularization term at all. It is just a consequence of the model's distribution over its latent variables. This view provides a different motivation for training an autoencoder: it is a way of approximately training a generative model. It also provides a different reason for

why the features learned by the autoencoder are useful: they describe the latent variables that explain the input.

Early work on sparse autoencoders (Ranzato *et al.*, 2007a, 2008) explored various forms of sparsity and proposed a connection between the sparsity penalty and the $\log Z$ term that arises when applying maximum likelihood to an undirected probabilistic model $p(\boldsymbol{x}) = \frac{1}{Z}\tilde{p}(\boldsymbol{x})$. The idea is that minimizing $\log Z$ prevents a probabilistic model from having high probability everywhere, and imposing sparsity on an autoencoder prevents the autoencoder from having low reconstruction error everywhere. In this case, the connection is on the level of an intuitive understanding of a general mechanism rather than a mathematical correspondence. The interpretation of the sparsity penalty as corresponding to $\log p_{\text{model}}(\boldsymbol{h})$ in a directed model $p_{\text{model}}(\boldsymbol{h})p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$ is more mathematically straightforward.

One way to achieve *actual zeros* in $\boldsymbol{h}$ for sparse (and denoising) autoencoders was introduced in Glorot *et al.* (2011b). The idea is to use rectified linear units to produce the code layer. 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.

14.2.2 Denoising Autoencoders

Rather than adding a penalty Ω to the cost function, we can obtain an autoencoder that learns something useful by changing the reconstruction error term of the cost function.

Traditionally, autoencoders minimize some function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))), \tag{14.8}$$

where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as the L^2 norm of their difference. This encourages $g \circ f$ to learn to be merely an identity function if they have the capacity to do so.

A **denoising autoencoder** (DAE) instead minimizes

$$L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}}))), \tag{14.9}$$

where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ that has been corrupted by some form of noise. Denoising autoencoders must therefore undo this corruption rather than simply copying their input.

Denoising training forces f and g to implicitly learn the structure of $p_{\text{data}}(\boldsymbol{x})$, as shown by Alain and Bengio (2013) and Bengio *et al.* (2013c). Denoising

autoencoders thus provide yet another example of how useful properties can emerge as a byproduct of minimizing reconstruction error. They are also an example of how overcomplete, high-capacity models may be used as autoencoders as long as care is taken to prevent them from learning the identity function. Denoising autoencoders are presented in more detail in section 14.5.

14.2.3 Regularizing by Penalizing Derivatives

Another strategy for regularizing an autoencoder is to use a penalty Ω, as in sparse autoencoders,

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}, \boldsymbol{x}), \tag{14.10}$$

but with a different form of Ω:

$$\Omega(\boldsymbol{h}, \boldsymbol{x}) = \lambda \sum_i ||\nabla_{\boldsymbol{x}} h_i||^2. \tag{14.11}$$

This forces the model to learn a function that does not change much when $\boldsymbol{x}$ changes slightly. Because this penalty is applied only at training examples, it forces the autoencoder to learn features that capture information about the training distribution.

An autoencoder regularized in this way is called a **contractive autoencoder**, or CAE. This approach has theoretical connections to denoising autoencoders, manifold learning, and probabilistic modeling. The CAE is described in more detail in section 14.7.

14.3 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. In fact, using deep encoders and decoders offers many advantages.

Recall from section 6.4.1 that there are many advantages to depth in a feedforward network. Because autoencoders are feedforward networks, these advantages also apply to autoencoders. Moreover, the encoder is itself a feedforward network, as is the decoder, so each of these components of the autoencoder can individually benefit from depth.

One major advantage of nontrivial 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.

Depth can exponentially reduce the computational cost of representing some functions. Depth can also exponentially decrease the amount of training data needed to learn some functions. See section 6.4.1 for a review of the advantages of depth in feedforward networks.

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

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

14.4 Stochastic Encoders and Decoders

Autoencoders are just feedforward networks. The same loss functions and output unit types that can be used for traditional feedforward networks are also used for autoencoders.

As described in section 6.2.2.4, a general strategy for designing the output units and the loss function of a feedforward network is to define an output distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ and minimize the negative log-likelihood $-\log p(\boldsymbol{y} \mid \boldsymbol{x})$. In that setting, $\boldsymbol{y}$ is a vector of targets, such as class labels.

In an autoencoder, $\boldsymbol{x}$ is now the target as well as the input. Yet we can still apply the same machinery as before. Given a hidden code $\boldsymbol{h}$, we may think of the decoder as providing a conditional distribution $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. We may then train the autoencoder by minimizing $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. The exact form of this loss function will change depending on the form of p_{decoder}. As with traditional feedforward networks, we usually use linear output units to parametrize the mean of a Gaussian distribution if $\boldsymbol{x}$ is real valued. In that case, the negative log-likelihood yields a mean squared error criterion. Similarly, binary $\boldsymbol{x}$ values correspond to a Bernoulli distribution whose parameters are given by a sigmoid output unit, discrete $\boldsymbol{x}$ values correspond to a softmax distribution, and so on. Typically, the

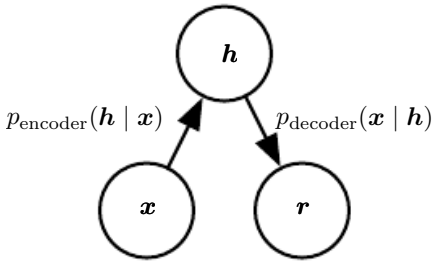

Figure 14.2: The structure of a stochastic autoencoder, in which both the encoder and the decoder are not simple functions but instead involve some noise injection, meaning that their output can be seen as sampled from a distribution, $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ for the encoder and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ for the decoder.

output variables are treated as being conditionally independent given $\boldsymbol{h}$ so that this probability distribution is inexpensive to evaluate, but some techniques, such as mixture density outputs, allow tractable modeling of outputs with correlations.

To make a more radical departure from the feedforward networks we have seen previously, we can also generalize the notion of an **encoding function** $f(\boldsymbol{x})$ to an **encoding distribution** $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$, as illustrated in figure 14.2.

Any latent variable model $p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x})$ defines a stochastic encoder

$$p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x}) = p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x}) \tag{14.12}$$

and a stochastic decoder

$$p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.13}$$

In general, the encoder and decoder distributions are not necessarily conditional distributions compatible with a unique joint distribution $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h})$. Alain *et al.* (2015) showed that training the encoder and decoder as a denoising autoencoder will tend to make them compatible asymptotically (with enough capacity and examples).

14.5 Denoising Autoencoders

The **denoising autoencoder** (DAE) is an autoencoder that receives a corrupted data point as input and is trained to predict the original, uncorrupted data point as its output.

The DAE training procedure is illustrated in figure 14.3. We introduce a corruption process $C(\tilde{\mathbf{x}} \mid \mathbf{x})$, which represents a conditional distribution over

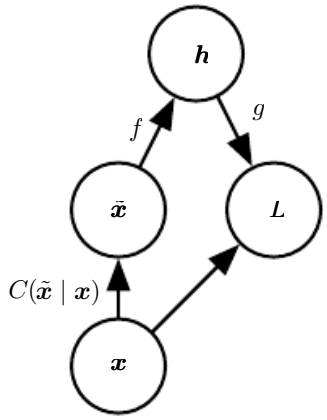

Figure 14.3: The computational graph of the cost function for a denoising autoencoder, which is trained to reconstruct the clean data point $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$. This is accomplished by minimizing the loss $L = -\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}}))$, where $\tilde{\boldsymbol{x}}$ is a corrupted version of the data example $\boldsymbol{x}$, obtained through a given corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$. Typically the distribution p_{decoder} is a factorial distribution whose mean parameters are emitted by a feedforward network g.

corrupted samples $\tilde{\mathbf{x}}$, given a data sample $\mathbf{x}$. The autoencoder then learns a **reconstruction distribution** $p_{\text{reconstruct}}(\mathbf{x} \mid \tilde{\mathbf{x}})$ estimated from training pairs $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ as follows:

1. Sample a training example $\boldsymbol{x}$ from the training data.
2. Sample a corrupted version $\tilde{\boldsymbol{x}}$ from $C(\tilde{\mathbf{x}} \mid \mathbf{x} = \boldsymbol{x})$.
3. Use $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ as a training example for estimating the autoencoder reconstruction distribution $p_{\text{reconstruct}}(\boldsymbol{x} \mid \tilde{\boldsymbol{x}}) = p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ with $\boldsymbol{h}$ the output of encoder $f(\tilde{\boldsymbol{x}})$ and p_{decoder} typically defined by a decoder $g(\boldsymbol{h})$.

Typically we can simply perform gradient-based approximate minimization (such as minibatch gradient descent) on the negative log-likelihood $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. As long as the encoder is deterministic, the denoising autoencoder is a feedforward network and may be trained with exactly the same techniques as any other feedforward network.

We can therefore view the DAE as performing stochastic gradient descent on the following expectation:

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}(\mathbf{x})} \mathbb{E}_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \boldsymbol{x})} \log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}})), \tag{14.14}$$

where $\hat{p}_{\text{data}}(\mathbf{x})$ is the training distribution.

14.5.1 Estimating the Score

Score matching (Hyvärinen, 2005) is an alternative to maximum likelihood. It provides a consistent estimator of probability distributions based on encouraging the model to have the same **score** as the data distribution at every training point $\boldsymbol{x}$. In this context, the score is a particular gradient field:

$$\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}). \tag{14.15}$$

Score matching is discussed further in section 18.4. For the present discussion, regarding autoencoders, it is sufficient to understand that learning the gradient field of $\log p_{\text{data}}$ is one way to learn the structure of p_{data} itself.

A very important property of DAEs is that their training criterion (with conditionally Gaussian $p(\boldsymbol{x} \mid \boldsymbol{h})$) makes the autoencoder learn a vector field $(g(f(\boldsymbol{x})) - \boldsymbol{x})$ that estimates the score of the data distribution. This is illustrated in figure 14.4.

Denoising training of a specific kind of autoencoder (sigmoidal hidden units, linear reconstruction units) using Gaussian noise and mean squared error as the reconstruction cost is equivalent (Vincent, 2011) to training a specific kind of undirected probabilistic model called an RBM with Gaussian visible units. This kind of model is described in detail in section 20.5.1; for the present discussion, it suffices to know that it is a model that provides an explicit $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$. When the RBM is trained using **denoising score matching** (Kingma and LeCun, 2010), its learning algorithm is equivalent to denoising training in the corresponding autoencoder. With a fixed noise level, regularized score matching is not a consistent estimator; it instead recovers a blurred version of the distribution. If the noise level is chosen to approach 0 when the number of examples approaches infinity, however, then consistency is recovered. Denoising score matching is discussed in more detail in section 18.5.

Other connections between autoencoders and RBMs exist. Score matching applied to RBMs yields a cost function that is identical to reconstruction error combined with a regularization term similar to the contractive penalty of the CAE (Swersky *et al.*, 2011). Bengio and Delalleau (2009) showed that an autoencoder gradient provides an approximation to contrastive divergence training of RBMs.

For continuous-valued $\boldsymbol{x}$, the denoising criterion with Gaussian corruption and reconstruction distribution yields an estimator of the score that is applicable to general encoder and decoder parametrizations (Alain and Bengio, 2013). This means a generic encoder-decoder architecture may be made to estimate the score

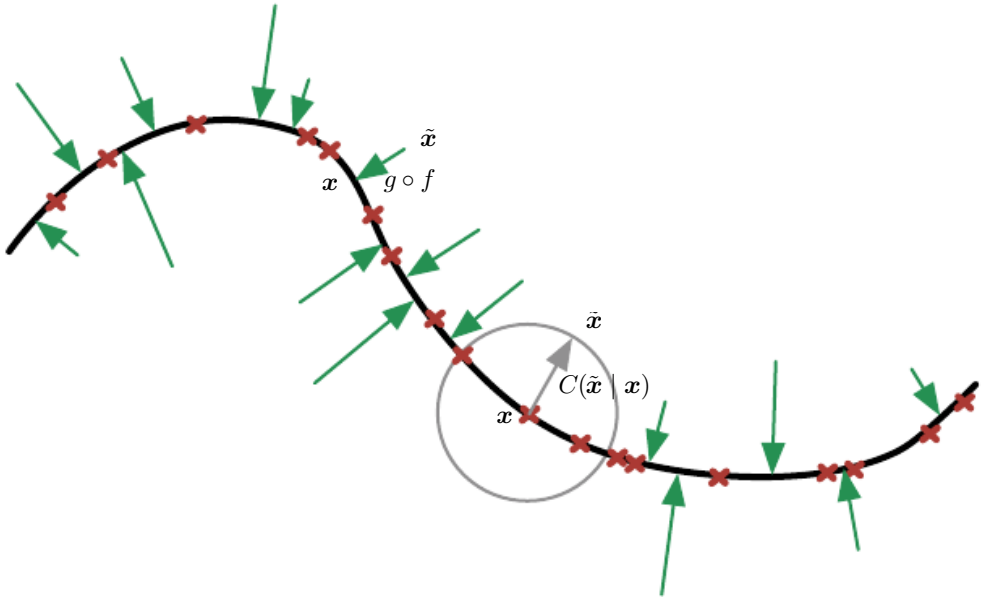

Figure 14.4: A denoising autoencoder is trained to map a corrupted data point $\tilde{\boldsymbol{x}}$ back to the original data point $\boldsymbol{x}$. We illustrate training examples $\boldsymbol{x}$ as red crosses lying near a low-dimensional manifold, illustrated with the bold black line. We illustrate the corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ with a gray circle of equiprobable corruptions. A gray arrow demonstrates how one training example is transformed into one sample from this corruption process. When the denoising autoencoder 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 $\mathbb{E}_{\mathbf{x}, \tilde{\mathbf{x}} \sim p_{\text{data}}(\mathbf{x}) C(\tilde{\mathbf{x}} \mid \mathbf{x})}[\mathbf{x} \mid \tilde{\mathbf{x}}]$. The vector $g(f(\tilde{\boldsymbol{x}})) - \tilde{\boldsymbol{x}}$ points approximately toward the nearest point on the manifold, since $g(f(\tilde{\boldsymbol{x}}))$ estimates the center of mass of the clean points $\boldsymbol{x}$ that could have given rise to $\tilde{\boldsymbol{x}}$. The autoencoder thus learns a vector field $g(f(\boldsymbol{x})) - \boldsymbol{x}$ indicated by the green arrows. This vector field estimates the score $\nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})$ up to a multiplicative factor that is the average root mean square reconstruction error.

by training with the squared error criterion

$$||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2 \tag{14.16}$$

and corruption

$$C(\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}|\boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}; \mu = \boldsymbol{x}, \Sigma = \sigma^2 I) \tag{14.17}$$

with noise variance σ^2. See figure 14.5 for an illustration of how this works.

In general, there is no guarantee that the reconstruction $g(f(\boldsymbol{x}))$ minus the input $\boldsymbol{x}$ corresponds to the gradient of any function, let alone to the score. That is why the early results (Vincent, 2011) are specialized to particular parametrizations, where $g(f(\boldsymbol{x})) - \boldsymbol{x}$ may be obtained by taking the derivative of another function. Kamyshanska and Memisevic (2015) generalized the results of Vincent (2011) by identifying a family of shallow autoencoders such that $g(f(\boldsymbol{x})) - \boldsymbol{x}$ corresponds to a score for all members of the family.

So far we have described only how the denoising autoencoder learns to represent a probability distribution. More generally, one may want to use the autoencoder as a generative model and draw samples from this distribution. This is described in section 20.11.

14.5.1.1 Historical Perspective

The idea of using MLPs for denoising dates back to the work of LeCun (1987) and Gallinari *et al.* (1987). Behnke (2001) also used recurrent networks to denoise images. Denoising autoencoders are, in some sense, just MLPs trained to denoise. The name "denoising autoencoder," however, refers to a model that is intended not merely to learn to denoise its input but to learn a good internal representation as a side effect of learning to denoise. This idea came much later (Vincent *et al.*, 2008, 2010). The learned representation may then be used to pretrain a deeper unsupervised network or a supervised network. Like sparse autoencoders, sparse coding, contractive autoencoders, and other regularized autoencoders, the motivation for DAEs was to allow the learning of a very high-capacity encoder while preventing the encoder and decoder from learning a useless identity function.

Prior to the introduction of the modern DAE, Inayoshi and Kurita (2005) explored some of the same goals with some of the same methods. Their approach minimizes reconstruction error in addition to a supervised objective while injecting noise in the hidden layer of a supervised MLP, with the objective to improve generalization by introducing the reconstruction error and the injected noise. Their method was based on a linear encoder, however, and could not learn function families as powerful as the modern DAE can.

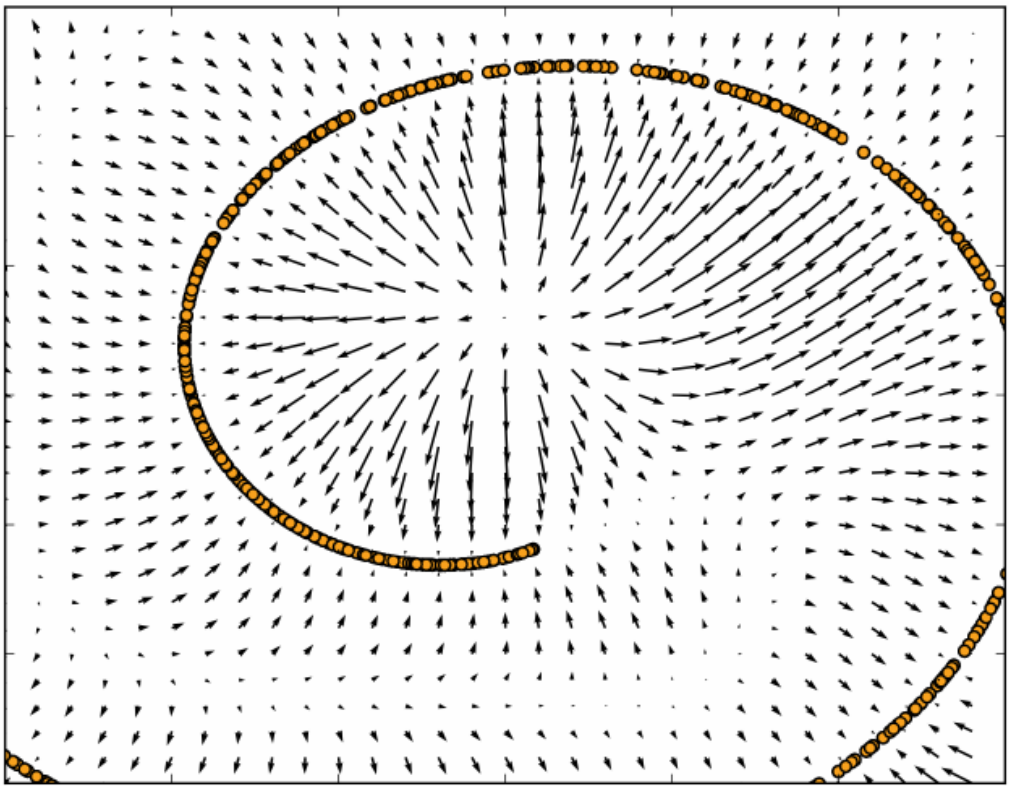

Figure 14.5: Vector field learned by a denoising autoencoder around a 1-D curved manifold near which the data concentrate in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the autoencoder and points towards higher probability according to the implicitly estimated probability distribution. The vector field has zeros at both maxima of the estimated density function (on the data manifolds) and at minima of that density function. For example, the spiral arm forms a 1-D manifold of local maxima that are connected to each other. Local minima appear near the middle of the gap between two arms. When the norm of reconstruction error (shown by the length of the arrows) is large, probability can be significantly increased by moving in the direction of the arrow, and that is mostly the case in places of low probability. The autoencoder maps these low probability points to higher probability reconstructions. Where probability is maximal, the arrows shrink because the reconstruction becomes more accurate. Figure reproduced with permission from Alain and Bengio (2013).

14.6 Learning Manifolds with Autoencoders

Like many other machine learning algorithms, autoencoders exploit the idea that data concentrates around a low-dimensional manifold or a small set of such manifolds, as described in section 5.11.3. Some machine learning algorithms exploit this idea only insofar as they learn a function that behaves correctly on the manifold but that may have unusual behavior if given an input that is off the manifold. Autoencoders take this idea further and aim to learn the structure of the manifold.

To understand how autoencoders do this, we must present some important characteristics of manifolds.

An important characterization of a manifold is the set of its **tangent planes**. At a point $\boldsymbol{x}$ on a d-dimensional manifold, the tangent plane is given by d basis vectors that span the local directions of variation allowed on the manifold. As illustrated in figure 14.6, these local directions specify how one can change $\boldsymbol{x}$ infinitesimally while staying on the manifold.

All autoencoder training procedures involve a compromise between two forces:

1. Learning a representation $\boldsymbol{h}$ of a training example $\boldsymbol{x}$ such that $\boldsymbol{x}$ can be approximately recovered from $\boldsymbol{h}$ through a decoder. The fact that $\boldsymbol{x}$ is drawn from the training data is crucial, because it means the autoencoder need not successfully reconstruct inputs that are not probable under the data-generating distribution.

2. Satisfying the constraint or regularization penalty. This can be an architectural constraint that limits the capacity of the autoencoder, or it can be a regularization term added to the reconstruction cost. These techniques generally prefer solutions that are less sensitive to the input.

Clearly, neither force alone would be useful—copying the input to the output is not useful on its own, nor is ignoring the input. Instead, the two forces together are useful because they force the hidden representation to capture information about the structure of the data-generating distribution. The important principle is that the autoencoder can afford to represent *only the variations that are needed to reconstruct training examples*. If the data-generating distribution concentrates near a low-dimensional manifold, this yields representations that implicitly capture a local coordinate system for this manifold: only the variations tangent to the manifold around $\boldsymbol{x}$ need to correspond to changes in $\boldsymbol{h} = f(\boldsymbol{x})$. Hence the encoder learns a mapping from the input space $\boldsymbol{x}$ to a representation space, a mapping that is only sensitive to changes along the manifold directions, but that is insensitive to changes orthogonal to the manifold.

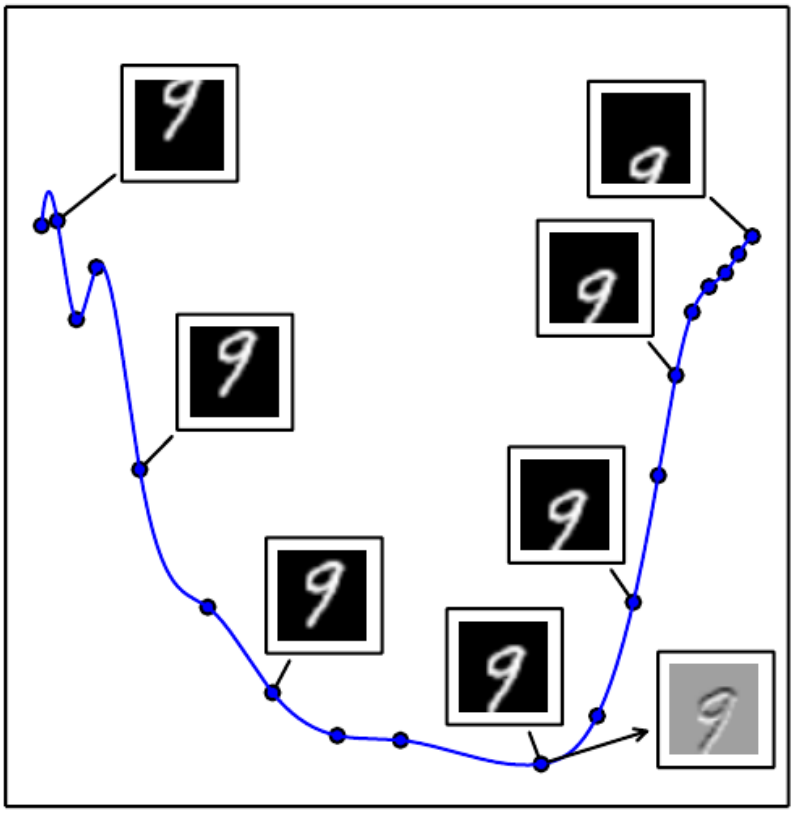

Figure 14.6: An illustration of the concept of a tangent hyperplane. Here we create a 1-D manifold in 784-D space. We take an MNIST image with 784 pixels and transform it by translating it vertically. The amount of vertical translation defines a coordinate along a 1-D manifold that traces out a curved path through image space. This plot shows a few points along this manifold. For visualization, we have projected the manifold into 2-D space using PCA. An n-dimensional manifold has an n-dimensional tangent plane at every point. This tangent plane touches the manifold exactly at that point and is oriented parallel to the surface at that point. It defines the space of directions in which it is possible to move while remaining on the manifold. This 1-D manifold has a single tangent line. We indicate an example tangent line at one point, with an image showing how this tangent direction appears in image space. Gray pixels indicate pixels that do not change as we move along the tangent line, white pixels indicate pixels that brighten, and black pixels indicate pixels that darken.

A one-dimensional example is illustrated in figure 14.7, showing that, by making the reconstruction function insensitive to perturbations of the input around the data points, we cause the autoencoder to recover the manifold structure.

To understand why autoencoders are useful for manifold learning, it is instructive to compare them to other approaches. What is most commonly learned to characterize a manifold is a **representation** of the data points on (or near) the manifold. Such a representation for a particular example is also called its embedding. It is typically given by a low-dimensional vector, with fewer dimensions than the "ambient" space of which the manifold is a low-dimensional subset. Some algorithms (nonparametric manifold learning algorithms, discussed below) directly learn an embedding for each training example, while others learn a more general mapping, sometimes called an encoder, or representation function, that maps any point in the ambient space (the input space) to its embedding.

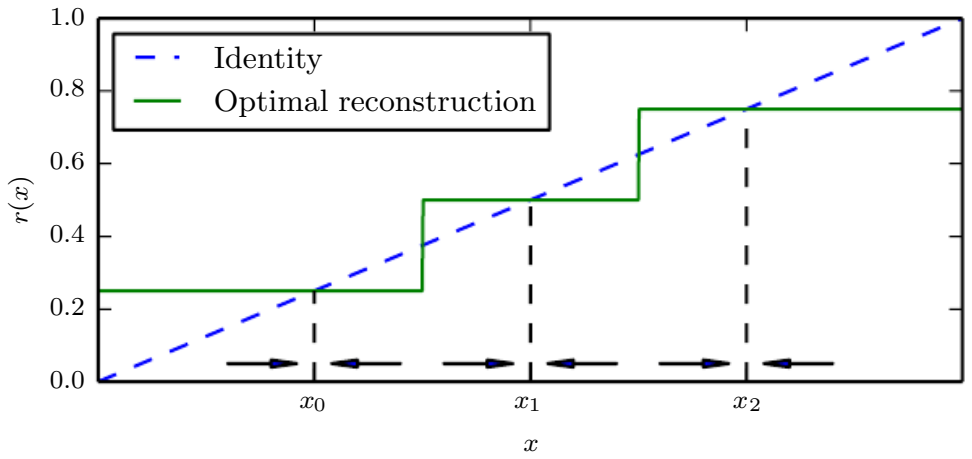

Figure 14.7: If the autoencoder learns a reconstruction function that is invariant to small perturbations near the data points, it captures the manifold structure of the data. Here the manifold structure is a collection of 0-dimensional manifolds. The dashed diagonal line indicates the identity function target for reconstruction. The optimal reconstruction function crosses the identity function wherever there is a data point. The horizontal arrows at the bottom of the plot indicate the $r(\boldsymbol{x}) - \boldsymbol{x}$ reconstruction direction vector at the base of the arrow, in input space, always pointing toward the nearest "manifold" (a single data point in the 1-D case). The denoising autoencoder explicitly tries to make the derivative of the reconstruction function $r(\boldsymbol{x})$ small around the data points. The contractive autoencoder does the same for the encoder. Although the derivative of $r(\boldsymbol{x})$ is asked to be small around the data points, it can be large between the data points. The space between the data points corresponds to the region between the manifolds, where the reconstruction function must have a large derivative to map corrupted points back onto the manifold.

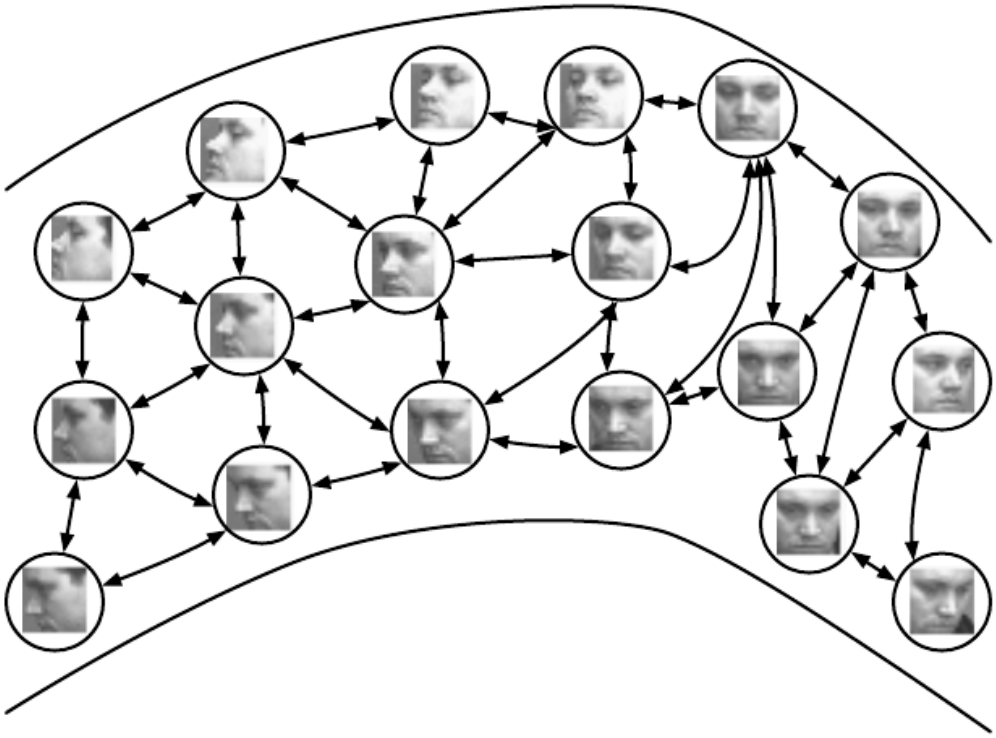

Figure 14.8: Nonparametric manifold learning procedures build a nearest neighbor graph in which nodes represent training examples and directed edges indicate nearest neighbor relationships. Various procedures can thus obtain the tangent plane associated with a neighborhood of the graph as well as a coordinate system that associates each training example with a real-valued vector position, or **embedding**. It is possible to generalize such a representation to new examples by a form of interpolation. As long as the number of examples is large enough to cover the curvature and twists of the manifold, these approaches work well. Images from the QMUL Multiview Face Dataset (Gong *et al.*, 2000).

Manifold learning has mostly focused on unsupervised learning procedures that attempt to capture these manifolds. Most of the initial machine learning research on learning nonlinear manifolds has focused on **nonparametric** methods based on the **nearest neighbor graph**. This graph has one node per training example and edges connecting near neighbors to each other. These methods (Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004; Hinton and Roweis, 2003; van der Maaten and Hinton, 2008) associate each node with a tangent plane that spans the directions of variations associated with the difference vectors between the example and its neighbors, as illustrated in figure 14.8.

A global coordinate system can then be obtained through an optimization or by solving a linear system. Figure 14.9 illustrates how a manifold can be tiled by

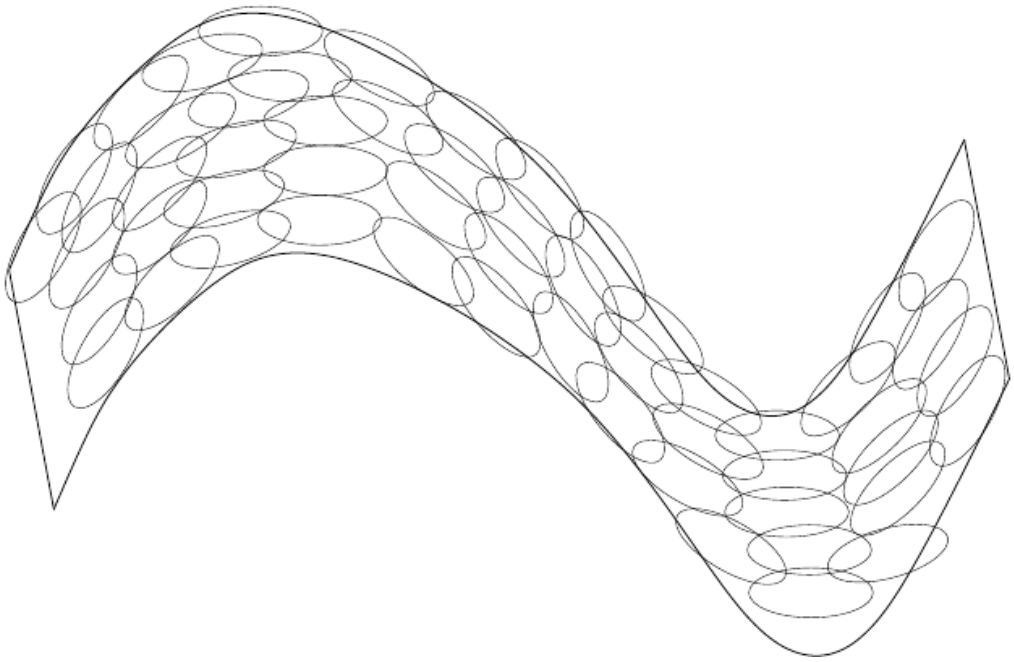

Figure 14.9: If the tangent planes (see figure 14.6) at each location are known, then they can be tiled to form a global coordinate system or a density function. Each local patch can be thought of as a local Euclidean coordinate system or as a locally flat Gaussian, or "pancake," with a very small variance in the directions orthogonal to the pancake and a very large variance in the directions defining the coordinate system on the pancake. A mixture of these Gaussians provides an estimated density function, as in the manifold Parzen window algorithm (Vincent and Bengio, 2003) or in its nonlocal neural-net-based variant (Bengio *et al.*, 2006c).

a large number of locally linear Gaussian-like patches (or "pancakes," because the Gaussians are flat in the tangent directions).

A fundamental difficulty with such local nonparametric approaches to manifold learning is raised in Bengio and Monperrus (2005): if the manifolds are not very smooth (they have many peaks and troughs and twists), one may need a very large number of training examples to cover each one of these variations, with no chance to generalize to unseen variations. Indeed, these methods can only generalize the shape of the manifold by interpolating between neighboring examples. Unfortunately, the manifolds involved in AI problems can have very complicated structures that can be difficult to capture from only local interpolation. Consider for example the manifold resulting from translation shown in figure 14.6. If we watch just one coordinate within the input vector, x_i, as the image is translated, we will observe that one coordinate encounters a peak or a trough in its value once for every peak or trough in brightness in the image. In other words, the complexity of the patterns of brightness in an underlying image template drives the complexity of the manifolds that are generated by performing simple image transformations. This motivates the use of distributed representations and deep learning for capturing manifold structure.

14.7 Contractive Autoencoders

The contractive autoencoder (Rifai *et al.*, 2011a,b) introduces an explicit regularizer on the code $\boldsymbol{h} = f(\boldsymbol{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\boldsymbol{h}) = \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 . \tag{14.18}$$

The penalty $\Omega(\boldsymbol{h})$ is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function.

There is a connection between the denoising autoencoder and the contractive autoencoder: Alain and Bengio (2013) showed that in the limit of small Gaussian 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, denoising autoencoders make the reconstruction function resist small but finite-sized perturbations of the input, while contractive autoencoders make the feature extraction function resist infinitesimal perturbations of the input. When using the Jacobian-based contractive penalty to pretrain features $f(\boldsymbol{x})$ for use with a classifier, the best classification accuracy usually results from applying the

contractive penalty to $f(\boldsymbol{x})$ rather than to $g(f(\boldsymbol{x}))$. A contractive penalty on $f(\boldsymbol{x})$ also has close connections to score matching, as discussed in section 14.5.1.

The name **contractive** arises from the way that the CAE warps space. Specifically, because the CAE is trained to resist perturbations of its input, it is encouraged to map a neighborhood of input points to a smaller neighborhood of output points. We can think of this as contracting the input neighborhood to a smaller output neighborhood.

To clarify, the CAE is contractive only locally—all perturbations of a training point $\boldsymbol{x}$ are mapped near to $f(\boldsymbol{x})$. Globally, two different points $\boldsymbol{x}$ and $\boldsymbol{x}'$ may be mapped to $f(\boldsymbol{x})$ and $f(\boldsymbol{x}')$ points that are farther apart than the original points. It is plausible that f could be expanding in-between or far from the data manifolds (see, for example, what happens in the 1-D toy example of figure 14.7). When the $\Omega(\boldsymbol{h})$ penalty is applied to sigmoidal units, one easy way to shrink the Jacobian is to make the sigmoid units saturate to 0 or 1. This encourages the CAE to encode input points with extreme values of the sigmoid, which may be interpreted as a binary code. It also ensures that the CAE will spread its code values throughout most of the hypercube that its sigmoidal hidden units can span.

We can think of the Jacobian matrix $\boldsymbol{J}$ at a point $\boldsymbol{x}$ as approximating the nonlinear encoder $f(\boldsymbol{x})$ as being a linear operator. This allows us to use the word "contractive" more formally. In the theory of linear operators, a linear operator is said to be contractive if the norm of $\boldsymbol{J}\boldsymbol{x}$ remains less than or equal to 1 for all unit-norm $\boldsymbol{x}$. In other words, $\boldsymbol{J}$ is contractive if its image of the unit sphere is completely encapsulated by the unit sphere. We can think of the CAE as penalizing the Frobenius norm of the local linear approximation of $f(\boldsymbol{x})$ at every training point $\boldsymbol{x}$ in order to encourage each of these local linear operators to become a contraction.

As described in section 14.6, regularized autoencoders learn manifolds by balancing two opposing forces. In the case of the CAE, these two forces are reconstruction error and the contractive penalty $\Omega(\boldsymbol{h})$. Reconstruction error alone would encourage the CAE to learn an identity function. The contractive penalty alone would encourage the CAE to learn features that are constant with respect to $\boldsymbol{x}$. The compromise between these two forces yields an autoencoder whose derivatives $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ are mostly tiny. Only a small number of hidden units, corresponding to a small number of directions in the input, may have significant derivatives.

The goal of the CAE is to learn the manifold structure of the data. Directions $\boldsymbol{x}$ with large $\boldsymbol{J}\boldsymbol{x}$ rapidly change $\boldsymbol{h}$, so these are likely to be directions that approximate the tangent planes of the manifold. Experiments by Rifai *et al.* (2011a,b) show that training the CAE results in most singular values of $\boldsymbol{J}$ dropping below 1 in

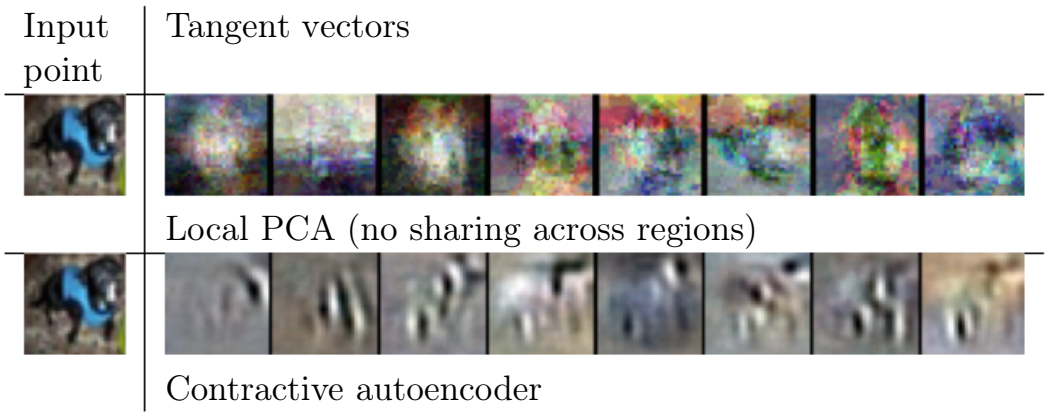

Figure 14.10: Illustration of tangent vectors of the manifold estimated by local PCA and by a contractive autoencoder. The location on the manifold is defined by the input image of a dog drawn from the CIFAR-10 dataset. The tangent vectors are estimated by the leading singular vectors of the Jacobian matrix $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ of the input-to-code mapping. Although both local PCA and the CAE can capture local tangents, the CAE is able to form more accurate estimates from limited training data 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). Images reproduced with permission from Rifai *et al.* (2011c).

magnitude and therefore becoming contractive. Some singular values remain above 1, however, because the reconstruction error penalty encourages the CAE to encode the directions with the most local variance. The directions corresponding to the largest singular values are interpreted as the tangent directions that the contractive autoencoder has learned. Ideally, these tangent directions should correspond to real variations in the data. For example, a CAE applied to images should learn tangent vectors that show how the image changes as objects in the image gradually change pose, as shown in figure 14.6. Visualizations of the experimentally obtained singular vectors do seem to correspond to meaningful transformations of the input image, as shown in figure 14.10.

One practical issue with the CAE regularization criterion is that although it is cheap to compute in the case of a single hidden layer autoencoder, it becomes much more expensive in the case of deeper autoencoders. The strategy followed by Rifai *et al.* (2011a) is to separately train a series of single-layer autoencoders, each trained to reconstruct the previous autoencoder's hidden layer. The composition of these autoencoders then forms a deep autoencoder. Because each layer was separately trained to be locally contractive, the deep autoencoder is contractive as well. The result is not the same as what would be obtained by jointly training the entire architecture with a penalty on the Jacobian of the deep model, but it

captures many of the desirable qualitative characteristics.

Another practical issue is that the contraction penalty can obtain useless results if we do not impose some sort of scale on the decoder. For example, the encoder could consist of multiplying the input by a small constant ϵ, and the decoder could consist of dividing the code by ϵ. As ϵ approaches 0, the encoder drives the contractive penalty $\Omega(\boldsymbol{h})$ to approach 0 without having learned anything about the distribution. Meanwhile, the decoder maintains perfect reconstruction. In Rifai *et al.* (2011a), this is prevented by tying the weights of f and g. Both f and g are standard neural network layers consisting of an affine transformation followed by an element-wise nonlinearity, so it is straightforward to set the weight matrix of g to be the transpose of the weight matrix of f.

14.8 Predictive Sparse Decomposition

Predictive sparse decomposition (PSD) is a model that is a hybrid of sparse coding and parametric autoencoders (Kavukcuoglu *et al.*, 2008). A parametric encoder is trained to predict the output of iterative inference. PSD has been applied to unsupervised feature learning for object recognition in images and video (Kavukcuoglu *et al.*, 2009, 2010; Jarrett *et al.*, 2009; Farabet *et al.*, 2011), as well as for audio (Henaff *et al.*, 2011). The model consists of an encoder $f(\boldsymbol{x})$ and a decoder $g(\boldsymbol{h})$ that are both parametric. During training, $\boldsymbol{h}$ is controlled by the optimization algorithm. Training proceeds by minimizing

$$||\boldsymbol{x} - g(\boldsymbol{h})||^2 + \lambda|\boldsymbol{h}|_1 + \gamma||\boldsymbol{h} - f(\boldsymbol{x})||^2. \tag{14.19}$$

As in sparse coding, the training algorithm alternates between minimization with respect to $\boldsymbol{h}$ and minimization with respect to the model parameters. Minimization with respect to $\boldsymbol{h}$ is fast because $f(\boldsymbol{x})$ provides a good initial value of $\boldsymbol{h}$, and the cost function constrains $\boldsymbol{h}$ to remain near $f(\boldsymbol{x})$ anyway. Simple gradient descent can obtain reasonable values of $\boldsymbol{h}$ in as few as ten steps.

The training procedure used by PSD is different from first training a sparse coding model and then training $f(\boldsymbol{x})$ to predict the values of the sparse coding features. The PSD training procedure regularizes the decoder to use parameters for which $f(\boldsymbol{x})$ can infer good code values.

Predictive sparse coding is an example of **learned approximate inference**. In section 19.5, this topic is developed further. The tools presented in chapter 19 make it clear that PSD can be interpreted as training a directed sparse coding probabilistic model by maximizing a lower bound on the log-likelihood of the model.

In practical applications of PSD, the iterative optimization is used only during training. The parametric encoder f is used to compute the learned features when the model is deployed. Evaluating f is computationally inexpensive compared to inferring $\boldsymbol{h}$ via gradient descent. Because f is a differentiable parametric function, PSD models may be stacked and used to initialize a deep network to be trained with another criterion.

14.9 Applications of Autoencoders

Autoencoders have been successfully applied to dimensionality reduction and information retrieval tasks. Dimensionality reduction was one of the first applications of representation learning and deep learning. It was one of the early motivations for studying autoencoders. For example, Hinton and Salakhutdinov (2006) trained a stack of RBMs and then used their weights to initialize a deep autoencoder with gradually smaller hidden layers, culminating in a bottleneck of 30 units. The resulting code yielded less reconstruction error than PCA into 30 dimensions, and the learned representation was qualitatively easier to interpret and relate to the underlying categories, with these categories manifesting as well-separated clusters.

Lower-dimensional representations can improve performance on many tasks, such as classification. Models of smaller spaces consume less memory and runtime. Many forms of dimensionality reduction place semantically related examples near each other, as observed by Salakhutdinov and Hinton (2007b) and Torralba *et al.* (2008). The hints provided by the mapping to the lower-dimensional space aid generalization.

One task that benefits even more than usual from dimensionality reduction is **information retrieval**, the task of finding entries in a database that resemble a query entry. This task derives the usual benefits from dimensionality reduction that other tasks do, but also derives the additional benefit that search can become extremely efficient in certain kinds of low-dimensional spaces. Specifically, if we train the dimensionality reduction algorithm to produce a code that is low-dimensional and *binary*, then we can store all database entries in a hash table that maps binary code vectors to entries. This hash table allows us to perform information retrieval by returning all database entries that have the same binary code as the query. We can also search over slightly less similar entries very efficiently, just by flipping individual bits from the encoding of the query. This approach to information retrieval via dimensionality reduction and binarization is called **semantic hashing** (Salakhutdinov and Hinton, 2007b, 2009b) and has been applied to both textual input (Salakhutdinov and Hinton, 2007b, 2009b) and

images (Torralba *et al.*, 2008; Weiss *et al.*, 2008; Krizhevsky and Hinton, 2011).

To produce binary codes for semantic hashing, one typically uses an encoding function with sigmoids on the final layer. The sigmoid units must be trained to be saturated to nearly 0 or nearly 1 for all input values. One trick that can accomplish this is simply to inject additive noise just before the sigmoid nonlinearity during training. The magnitude of the noise should increase over time. To fight that noise and preserve as much information as possible, the network must increase the magnitude of the inputs to the sigmoid function, until saturation occurs.

The idea of learning a hashing function has been further explored in several directions, including the idea of training the representations to optimize a loss more directly linked to the task of finding nearby examples in the hash table (Norouzi and Fleet, 2011).