Before starting this chapter: read a very basic intro to autoencoders (e.g on wikipedia) and look at some simple code to see how autoencoders are used

Restrictions on autoencoders force them to learn only the most useful properties

Modern autoencoders have generalized the idea of an encoder and decoder beyond deterministic functions to stochastic mappings penc (h | x) and pdec (x | h)

Autoencoders may be trained in the same way as regular neural nets

However, they may also be trained with recirculation, which compares activations of the original input to the activations of the reconstructions

Fig 14.1

Shows a function f which generates the code h, and a decoder g which reconstructs x from the code

**Undercomplete Autoencoders**

The purpose of an autoencoder is not necessarily to reconstruct the input, but to make h a useful code

When the dimensionality of the code h is less than x, then the autoencoder is undercomplete, forcing it to learn only useful features

The loss L(x, g(f(x)) is a function which penalizes g(f(x)) for being dissimilar from x (e.g MSE)

When the decoder is linear and the loss is MSE, an autoencoder ends up performing PCA, learning the principal subspace as a side-effect (whereas this is the main goal in pure PCA)

If the encoder f and decoder g are nonlinear, then we have a more powerful nonlinear generalization of PCA

If the encoder and decoder capacities are too large, then the autoencoder can learn to copy the data without extracting useful info about the data

Theoretically, an autoencoder with a 1-dimensional code but a very powerful nonlinear encoder can generate a code i which can be mapped back to specific examples by a powerful decoder

**Regularized Autoencoders**

An overcomplete encoder, which has a higher dimension in the code, has the same problem where too much useless information can cause the model to not learn any useful features

Regularized autoencoders focus on other properties besides the ability to copy the input to the output; these include sparsity of the representations, noise, smallness of the derivative's representation, etc.

Optimizing this things can cause the model to learn a useful distribution

Any generative model (predicts probability of x given y) with latent variables and an inference procedure (which attempts to compute latent variables given an input) can be interpreted as an autoencoder

Variational autoencoders and generative stochastic networks are overcomplete; however, they learn useful features without being regularized since they are trained to maximize the probability of training data

*Sparse Autoencoders*

A sparse autoencoder has an extra penalty term to the loss function corresponding to a sparse penalty 𝞨 on the code layer h



Sparse autoencoders are used to learn features for other tasks such as classification

They force the model to learn unique statistical features, rather than just the identity function

Useful features are a byproduct of this method

We can think of the penalty 𝞨 as a regularizer term added to a feedforward net whose job is to copy the input to output and possibly perform some supervised task which depends on sparse features

While weight decay can be expressed as a prior over the parameters, regularized autoencoders depend on the data and is therefore not a prior

However, they still implicitly express a preference over functions

Rather than thinking of the sparse penalty as a regularizer, we can think of the framework as approximating max likelihood training of a generative model with latent variables

Suppose we have a model with visible joint distribution:

pmodel(h) is the model's prior distribution over the latent variables, representing its beliefs before seeing x

This is different from how prior was previously described, to refer to the distribution p(𝚹) encoding our beliefs about the model's parameters before seeing the training data

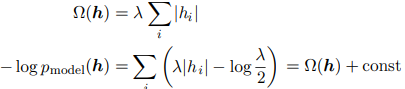
The log-likelihood can be decomposed as:



The log pmodel(h) term can be sparsity inducing; for example, the Laplace prior

corresponds to an absolute value sparsity penalty

Expressing the log prior as an absolute value penalty:



Substituting the first equation into the second and simplifying, we can see that the log prior is a function of the sparsity penalty 𝞨, and minimizing negative log likelihood correpsonds to minimizing the sparsity penalty

ƛ can be treated as a hyperparameter

From the point of view of sparsity as resulting from the effect of pmodel(h) on approximate max likelihood learning, the sparsity penalty is not a regularization term, but rather a byproduct of the model's distribution over its latent variables

This lets us view an autoencoder as approximately training a generative model; it also explains why features learned by an autoencoder are useful => they describe latent variables which represent the input

Early work on sparse autoencoders proposed a connection between sparsity penalty and the log Z term when applying max likelihood to an undirected probabilistic model p(x) = 1/Z p(x)

Minimizing log Z prevents probablistic model from having a high probability everywhere, and imposing sparsity prevents the autoencoder from having low reconstruction error everywhere

Using ReLUs to produce the code layer can result in having actual zeros in h

With a prior which pushes the representations to 0, one can indirectly control the # of zeros in a representation

*Denoising Autoencoders*

Rather than using 𝞨, we can obtain a useful autoencoder by changing the reconstruction error term of the cost function

The loss function L(x, g(f(x)) which typically uses the L2 norm encourages g(f) to be an identity function, with the given capacity

A denoising autoencoder minimizes

where x~ is a copy of x which has been corrupted by some noise

Denoising autoencoders must therefore remove the noise as well

This forces f and g to implicitly learn the structure of pdata(x)

Denoising autoencoders can be overcomplete, if caution is taken

*Regularizing by Penalizing Derivatives*

Use a penalty 𝞨 as in sparse encoders but with a different form of 𝞨





This forces the model to learn a function which does not change much when x changes slightly; it forces the autoencoder to learn features that capture info about the training distribution (since it is not applied to the test)

An autoencoder this way is called a contractive autoencoder (CAE)

**Representational Power, Layer Size and Depth**

Both the encoder and decoder, which are individual feedforward nets, can benefit in the same way as neural nets from increased depth

Universal approximation theorem says that one hidden layer with a size can represent any function given enough capacity

An autoencoder can represent the identity function along the domain of data well, but the mapping from input to code is shallow

Therefore, we cannot impose arbitrary constraints such as a sparse code

A deep autoencoder can approximate any mapping from the input to code arbitrarily well given enough hidden units

Deep autoencoders yield much better compression, experimentally

A common strategy for training a deep autoencoder is to greedily pretrain the shallow architecture by training a stack of shallow autoencoders

**Stochastic Encoders and Decoders**

The general strategy is to define an output distribution p(y | x) and minimize - log p(y | x) where y is a vector of targets

Given a code h, the decoder provides a conditional distribution pdecoder(x | h)

We usually use linear units to parametrize the mean of a Gaussian if x is real-valued (no nonlinearity when using the decoder)

Negative log-likelihood in this case yields a mean-squread error criterion

Binary x values correspond to a Bernoulli distribution whose params are given by a sigmoid; discrete x correspond to softmax, and so on

Typically, the outputs are treated as being conditionally independent given h so that the probability distribution is conditionally independent given h

Some techniques such as mixture density allow tractable modeling of correlated outputs

We can generalize the notion of an encoding function f(x) to an encoding distribution p(h | x)

Any latent variable model pmodel(h, x) defines a stochastic encoder

and stochastic decoder



They are not necessarily conditional distributions compatible with a unique joint distribution pmodel(x, h)

Training the encoder and decoder as a denoising autoencoder tends to make them compatible asymptotically

**Denoising Autoencoder**

A denoising autoencoder receives a corrupted version of the input and is trained to predict the original as the output

We introduce a corruption process C(x~ | x) representing a conditional distribution over corrupted samples x~ given a data sample x

Fig 14.3

Shows a computational graph of a denoising autoencoder

Minimizes loss L = -log pdecoder (x | h = f(x~))

pdecoder is a factorial distribution whose mean parameters are emitted by a feedforward network g

The autoencoder learns a reconstruction distribution pconstruct (x | x~) as follows:

* Sample a training point x, and sample a corrupted version x~
* Use (x, x~) as a training example for estimating the autoencoder reconstruction preconstruct(x | x~) = pdecoder(x | h) with h the output of encoder f(x~) and pdec defined as g(h)

We can use gradient descent; as long as the encoder is deterministic, we can train it like a feedforward network

We can view the denoising autoencoder as performing SGD on the following:

(Minimizing the negative log likelihood of x given h as an encoded version of a corruption)

Fig 14.4

Shows a low dimensional manifold; at each point on the manifold, there is a gray arrow representing the corruption radius off the manifold

When minimizing the average of squared errors, the reconstruction estimates 

The vector g(f(x)) - x~ points to the nearest point on the manifold since g(f(x~)) measures the center of mass of possible x points which could have been corrupted to generate x~

The autoencoder thus learns a vector field, which estimates the score grad(log pdata(x)) up to a multiplicative factor, the average root mean squared construction error

*Estimating the Score*

Score matching trains the model to have the same score as the distribution at each training point x

Score is defined as a gradient field 

Learning this gradient field results in the model being able to learn the structure of pdata

The criterion p(x | h) corresponds to the autoencoder learning a vector field g(f(x)) - x which estimates the score of a data distribution

Denoising training of an autoencoder (with a particulr structure) using Gaussian noise and MSE is the same as using an RBM, which defines an explicit pmodel(x; 𝚹)

An RBM being trained using denoising score matching is the same as denoising training of an autoencoder

With a fixed noise level, regularized score matching is not a consistent estimator causing it to have a blurred distribution

However, as the noise approaches 0 and examples approach ∞, the estimator retrieves its consistency

Score matching on an RBM is equivalent to reconstructor error + a CAE contractive penalty

Training using the gradient of the autoencoder is an approximation of training RBMs using contrastive divergence

If x is continuous, then denoising provides an estimator for the score which corresponds to an encoder and decoder parametriation

In other words, an encoder-decoder architecture can estimate the score, with reconstruction error criterion:

and corruption with noise variance given by sig2 (variance in each element of x, not the covariance)

There is no guarantee that the reconstruction minus the input corresponds to the gradient of a score, or any function for that matter, necessarily

This is why early definitions added constraints to the parametrizations such that g(f) - x must be a derivative; this was later generalized so that g(f(x)) was a bounded set of autoencoders with gradients corresponding to scores

Fig 14.5

Shows data in 2D which clusters near a 1-dimensional manifold

Larger arrows are proportional to a reconstruction g(f) - x, which points to a high probability

The density function (represented by the data manifold) is 0 at maxima and minima

A large arrow, indicating a high norm of the reconstruction indicates that the probability can be increased by moving in the direction of the arrow; it means that the current spot is one of low probability

The arrow is small when the reconstruction g(f) is accurate

**Learning Manifolds with Autoencoders**

Some models make use of manifolds only by learning a function which behaves well when the input is on the manifold; if the input is off the manifold, this may not apply

Autoencoders take this a step further by attempting to learn the manifold's structure

An important characteristic of a manifold is its set of tangent planes of x

These contain d bases vectors, where d is the manifold's dimensionality, which span the variation of the manifold

This means if you take an infinitesemal step with x in these directions, you remain on the manifold

Autoencoders must use both these properties simultaneously:

* Having a representation h of the input x such that the reconstruction of x through the dcoder is approximately equal to x; it is important that the autoencoder learns the data distribution only if x is drawn from the proper data distribution
* Regularization which can be used on the cost, or the model capacity; typically, regularization is not sensitive to small changes in input

Using both these, h can learn the structure of pdata with only the variation directions needed for data reconstruction (no extra information)

Moving along the tangents on the manifold should correspond to changes in the representation h, while moving orthogonal to the tangents should correspond to no changes in h

Fig 14.6

Shows how a MNIST autoencoder reconstruction is changing as it moves along the 1-D manifold, with the direction tangent to the manifold at the point x

Fig 14.7

A representation of points x on or near the manifold characterizes it

An embedding is a lower level representation, indicated by a vector with smaller dimensionality, of the ambient space (which has a high dimensionality)

Some algorithms (nonparametric manifold learning algorithms) learn an embedding for each input, while other algorithms ("encoders") learn a general mapping which generates an embedding for any point in the ambibent space

The nearest neighbor graph is an example of a nonparametric manifold learning algorithm

Each training example is a node, connected by edges (neighbor connections) to its neighbors

Each node has a tangent plane which spans the directions of variation given by the difference vectors between an example and its neigbors

Fig 14.8

Shows a nearest neighbor graph, where each point corresponds to an embedding representing an implicit coordinate system

These points (training examples) have tangent planes between them

If there are enough points, then a well defined system which accounts for curvature can interpolate between examples

A global coordinate system can be obtained by solving a linear system

However, these non-parametric approaches you would need a very large number of training examples with no chance to generalize if the manifolds are not smooth

Unfortunately, the manifolds in AI problems are very complicated; the complexity of the underlying variables drives the complexity of the manifold

Fig 14.9

The tangent planes, given by pancakes, can be "tiled" or combined together to form a global coordinate system / density function

There is very small variation in the direction perpendicular to the pancake

A mixture of these Gaussians provides an estimated density function

**Contractive Autoencoders**

Contractive autoencoder imposes a constaint on h = f(x) encouraging the derivatives to be as small as possible



The penalty is the squared frobenius norm (sum of squared elements) of the Jacobian matrix associated with the encoder function <= it's a matrix since x is a multivariable vector

In the limit of small Gaussian noise, the denoising reconstruction error is equivalent to a contrative penalty on the mapping from x to the reconstruction r = g(f(x))

Using a denoising autoencoder w/ little noise, the reconstruction r resists small but finite change sin x, while contractive encoders resists infinitesmally small changes in x

When pre-training f(x) for clasisfication, applying a contractive penalty to f is better than to g(f(x))

Contractive autoencoder on f is about the same as score matching

It is called contractive because it contracts space by making the neighborhood of x map to a smaller neighborhood in f(x)

However, this only applies locally; two points far apart in x may be even further in f

Applying the penalty to a sigmoid will cause it to saturate to 0 or 1 in order to shrink the Jacobian

This encourages the CAE to encode input points with extreme values of the sigmoid which may be interpreted as binary code

It ensures that the CAE will spread its code values throughout the hypercube that its sigmoidal units can span

We can think of the Jacobian J at a point x as approximating f(x) as being a linear operator

A linear operator is contractive if the norm of Jx remains less than or equal to 1 for all unit norm x; J is "contractive" if it shrinks the unit sphere

We can think of CAE as penalizing the Frobenius norm of the linear approximation of f at every point x in order to encourge the local linear operation to be a contraction

Contractive autoencoder attempts to compromise between the reconstruction error, which attempts to create the identity function, and the contractive penalty 𝞨 which learns features that are constant with respect to x

This means that df/dx is mostly tiny with large df/fx only in a few hidden units

These hidden units correspond to an important direction in the input space

The direction x with a large Jacobian rapidly changes h, so these directions of x are likely to form the tangent planes

If the singular members of J drop below 1, then they are becoming contractive; singular values of J which are greater than 1 due to the reconstruction error penalty encode directions with large variance

Tangent directions should ideally correspond to changes in the data

Fig 14.10

Shows an illustration of the difference of tangent vectors between PCA and CAEs

Tangent vectors are given by leading singular vectors of the Jacobian dh/dx

CAE exploits parameter sharing across different locations in the image which have the same hidden units, giving it more accurate vectors than PCA

One problem with the contractive autoencoder is that the penalty criterion is hard to compute with a deep autencoder

One way to fix this is to train single-layer autoencoders individually, with each one trying to reconstruct / approximate the previous layer, and then to stack them to make it a deep AE

If each autoencoder is locally contractive, then the whole deep autoencoder is contractive

While this is not equal to jointly training the entire architecture with a Jacobian penalty, it does capture many important features

Another issue is that the contractive penalty can be useless if we do not impose a scale on the decoder

If the encoder transforms the input x by multiplying it by some small constant 𝟄, and the decoder reconstructs it by dividing the encoder output by 𝟄, then as 𝟄 -> 0, so does 𝞨

This means the penalty does not do anything to make the decoder learning a distribution

One way to fix this is to tie the weights of f and g

* Set the weight matrix of g to be the transpose of the weight matrix of f

**Predictive Sparse Decomposition**

Predictive sparse decomposition combines sparse coding (which assumes that linear factors have an isotropic distribution w/ covariance matrix 1/𝜷 \* **I**) and parametric autoenoders, which is trained to predict the output of iterative inference

* This means that unlike the other autoencoder methods in this chapter, we are not using a learned h

PSD is used for unsupervised feature learning for things like object recognition

f(x) and g(h) are both parametric

h is controlled by the optimization algorithm



Just like sparse coding, we alternate between minimizing w.r.t h and w.r.t the model parameters

Minimizing w.r.t h is easy because f(x) provides a good initial value of h, and the cost restricts h to remain near f anyways

The decoder is regularized to use parameters for which the encoder f can find good codes

Predictive sparse coding is an example of learned approximate inference

PSD is equivalent to a sparse coding probabilistic model trained by maximizing the lower bound of the log-likelihood

There is no iteration (through grad. descent) during inference; instead, f computes the learned fatures

Since f is differentiable, multiple layers can be stacked to pretrain a DNN with a different criterion

**Applications of Autoencoders**

Autoencoders have successfully been applied to dimensionality reduction; the first few models which stacked RBMs were still better than PCA, and the learned features were easy to represent as clusters

Autoencoders can also be applied to information retrieval which gets queries from a database search (e.g google searches)

The codes can be a binary low-dimensional code which are mapped to entries using a hash table

All entries corresponding to the binary code can be displayed; also, similar entries can be shown by flipping individual bits in the code

This is known as semantic hashing

The sigmoid must be saturated in order to generate binary codes; additive noise can be added just before the nonlinearity

The magnitude of the noise should increase over time; to fight this noise, the network must increase the magnitude of the inputs to the sigmoid until saturation (due to high values) occurs