

Deep Learning

Unlocking Artificial Intelligence



Chapter 1

Introduction

Inventors have long dreamed of creating machines that think. This desire dates back to at least the time of ancient Greece. The mythical figures Pygmalion, Daedalus, and Hephaestus may all be interpreted as legendary inventors, and Galatea, Talos, and Pandora may all be regarded as artificial life ([Ovid and Martin, 2004](#); [Sparkes, 1996](#); [Tandy, 1997](#)).

When programmable computers were first conceived, people wondered whether such machines might become intelligent, over a hundred years before one was built ([Lovelace, 1842](#)). Today, **artificial intelligence** (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research.

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

This book is about a solution to these more intuitive problems. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs. The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these

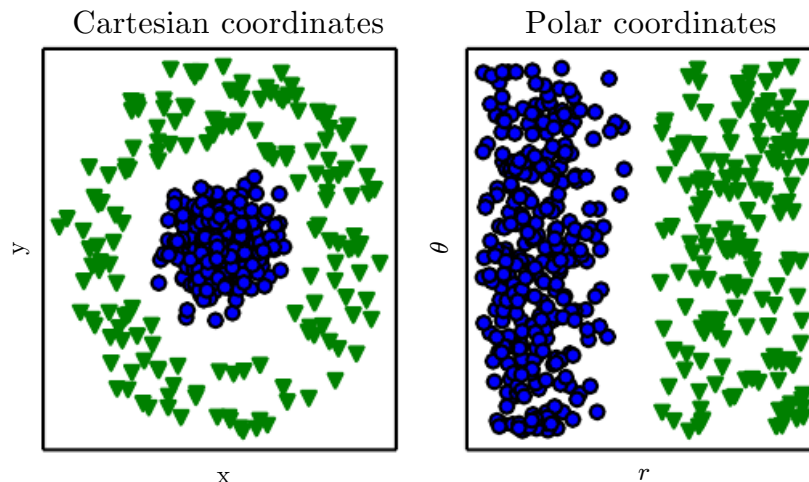


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data with polar coordinates and the task becomes simple to solve with a vertical line. Figure produced in collaboration with David Warde-Farley.

One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as **representation learning**. Learned representations often result in much better performance than can be obtained with hand-designed representations. They also allow AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

The quintessential example of a representation learning algorithm is the **autoencoder**. An autoencoder is the combination of an **encoder** function that converts the input data into a different representation, and a **decoder** function that converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties.

When designing features or algorithms for learning features, our goal is usually to separate the **factors of variation** that explain the observed data. In this context, we use the word “factors” simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not

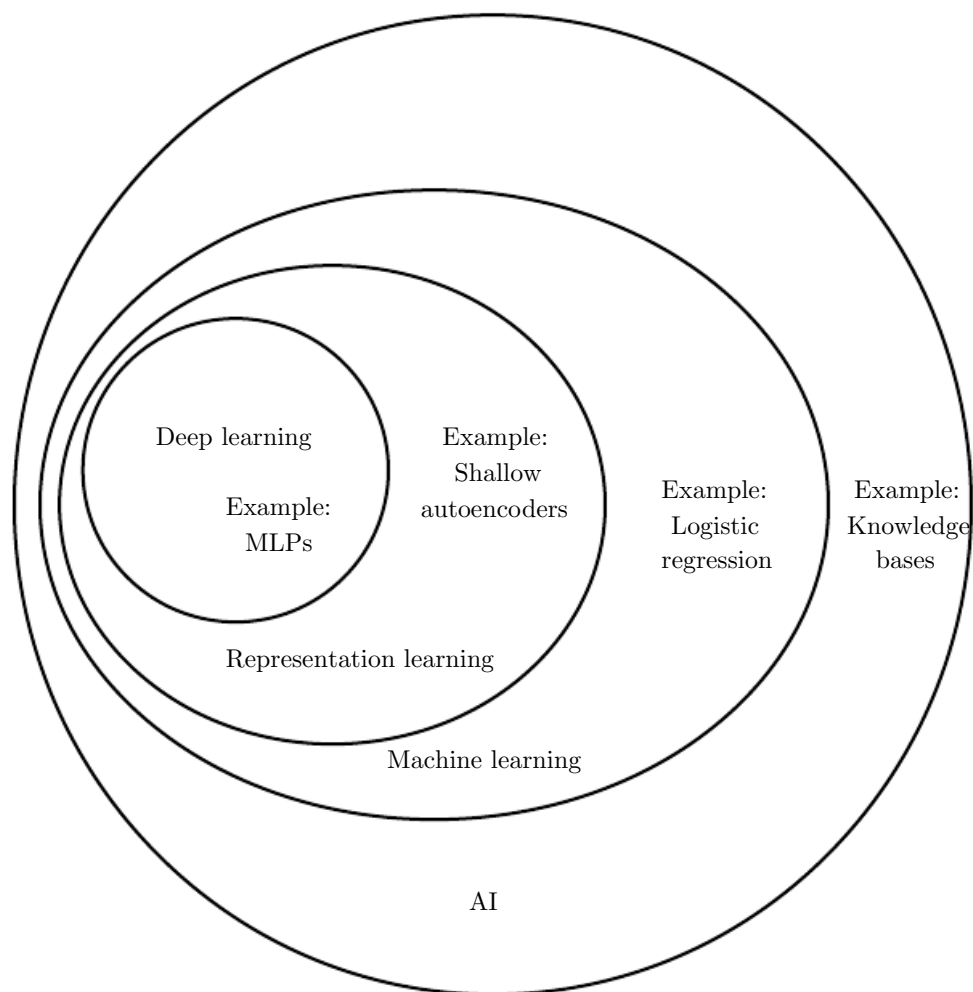


Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

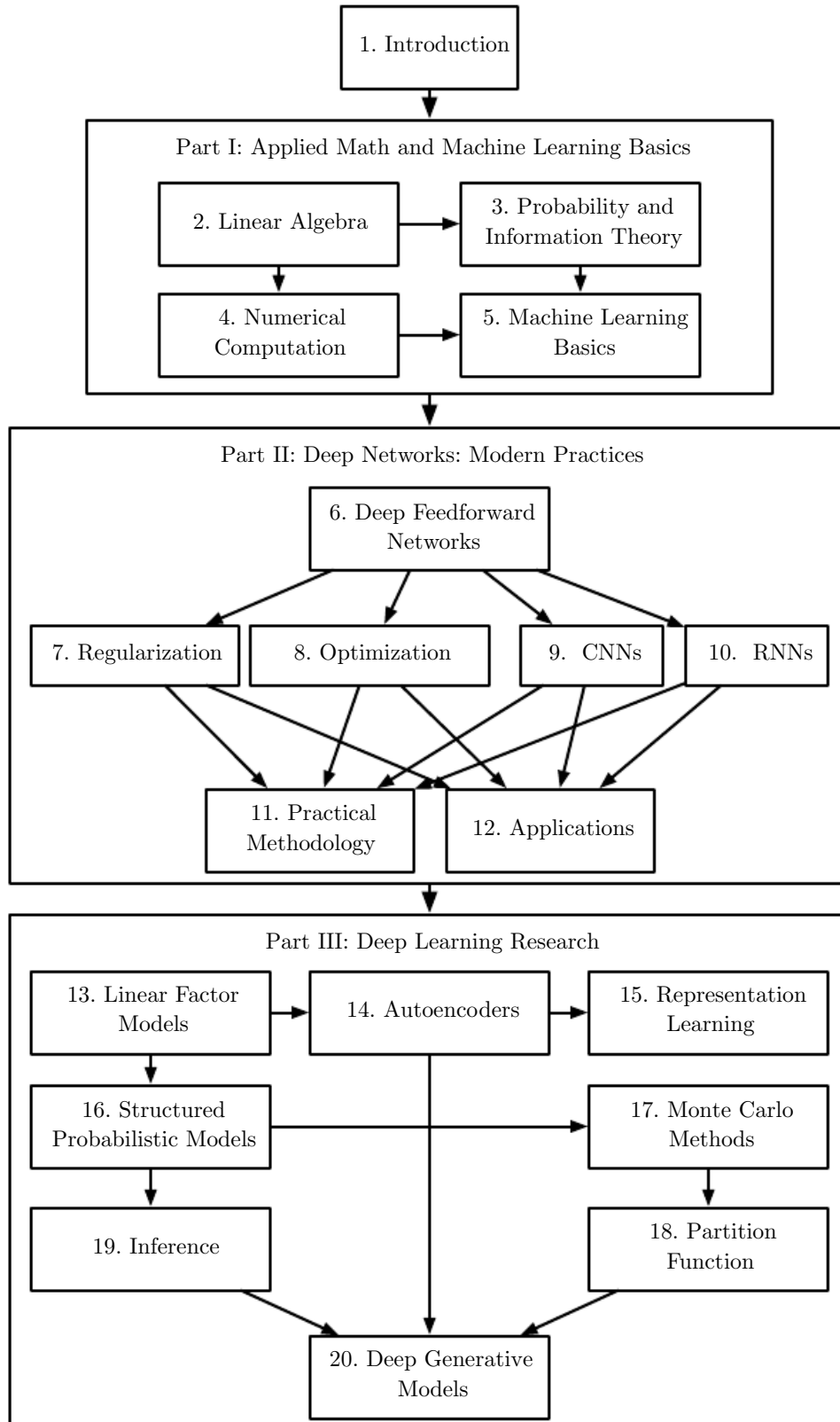


Figure 1.6: The high-level organization of the book. An arrow from one chapter to another indicates that the former chapter is prerequisite material for understanding the latter.

the advent of general purpose GPUs (described in section 12.1.2), faster network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

1.2.4 Increasing Accuracy, Complexity and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition or prediction. Moreover, deep learning has consistently been applied with success to broader and broader sets of applications.

The earliest deep models were used to recognize individual objects in tightly cropped, extremely small images (Rumelhart *et al.*, 1986a). Since then there has been a gradual increase in the size of images neural networks could process. Modern object recognition networks process rich high-resolution photographs and do not have a requirement that the photo be cropped near the object to be recognized (Krizhevsky *et al.*, 2012). Similarly, the earliest networks could only recognize two kinds of objects (or in some cases, the absence or presence of a single kind of object), while these modern networks typically recognize at least 1,000 different categories of objects. The largest contest in object recognition is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) held each year. A dramatic moment in the meteoric rise of deep learning came when a convolutional network won this challenge for the first time and by a wide margin, bringing down the state-of-the-art top-5 error rate from 26.1% to 15.3% (Krizhevsky *et al.*, 2012), meaning that the convolutional network produces a ranked list of possible categories for each image and the correct category appeared in the first five entries of this list for all but 15.3% of the test examples. Since then, these competitions are consistently won by deep convolutional nets, and as of this writing, advances in deep learning have brought the latest top-5 error rate in this contest down to 3.6%, as shown in figure 1.12.

Deep learning has also had a dramatic impact on speech recognition. After improving throughout the 1990s, the error rates for speech recognition stagnated starting in about 2000. The introduction of deep learning (Dahl *et al.*, 2010; Deng *et al.*, 2010b; Seide *et al.*, 2011; Hinton *et al.*, 2012a) to speech recognition resulted in a sudden drop of error rates, with some error rates cut in half. We will explore this history in more detail in section 12.3.

Deep networks have also had spectacular successes for pedestrian detection and image segmentation (Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013) and yielded superhuman performance in traffic sign classification (Ciresan

parametrized by a vector $\mathbf{p} \in [0, 1]^{k-1}$, where p_i gives the probability of the i -th state. The final, k -th state's probability is given by $1 - \mathbf{1}^\top \mathbf{p}$. Note that we must constrain $\mathbf{1}^\top \mathbf{p} \leq 1$. Multinoulli distributions are often used to refer to distributions over categories of objects, so we do not usually assume that state 1 has numerical value 1, etc. For this reason, we do not usually need to compute the expectation or variance of multinoulli-distributed random variables.

The Bernoulli and multinoulli distributions are sufficient to describe any distribution over their domain. They are able to describe any distribution over their domain not so much because they are particularly powerful but rather because their domain is simple; they model discrete variables for which it is feasible to enumerate all of the states. When dealing with continuous variables, there are uncountably many states, so any distribution described by a small number of parameters must impose strict limits on the distribution.

3.9.3 Gaussian Distribution

The most commonly used distribution over real numbers is the **normal distribution**, also known as the **Gaussian distribution**:

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right). \quad (3.21)$$

See figure 3.1 for a plot of the density function.

The two parameters $\mu \in \mathbb{R}$ and $\sigma \in (0, \infty)$ control the normal distribution. The parameter μ gives the coordinate of the central peak. This is also the mean of the distribution: $\mathbb{E}[x] = \mu$. The standard deviation of the distribution is given by σ , and the variance by σ^2 .

When we evaluate the PDF, we need to square and invert σ . When we need to frequently evaluate the PDF with different parameter values, a more efficient way of parametrizing the distribution is to use a parameter $\beta \in (0, \infty)$ to control the **precision** or inverse variance of the distribution:

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x - \mu)^2\right). \quad (3.22)$$

Normal distributions are a sensible choice for many applications. In the absence of prior knowledge about what form a distribution over the real numbers should take, the normal distribution is a good default choice for two major reasons.

stabilized. Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) is an example of a software package that automatically detects and stabilizes many common numerically unstable expressions that arise in the context of deep learning.

4.2 Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$. When $\mathbf{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its **condition number** is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \quad (4.2)$$

This is the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input.

This sensitivity is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion. Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself.

4.3 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(\mathbf{x})$ by altering $\mathbf{x}$. We usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\mathbf{x})$.

The function we want to minimize or maximize is called the **objective function** or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**. In this book, we use these terms interchangeably, though some machine learning publications assign special meaning to some of these terms.

We often denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $\mathbf{x}^* = \arg \min f(\mathbf{x})$.

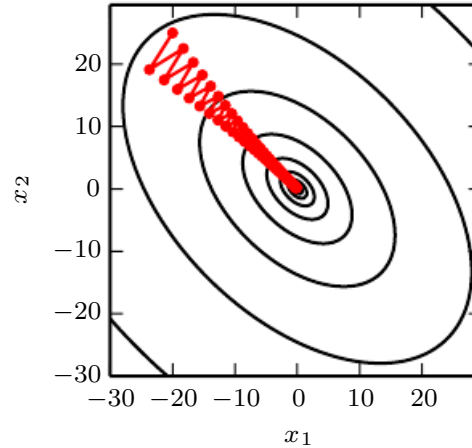


Figure 4.6: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent to minimize a quadratic function $f(\mathbf{x})$ whose Hessian matrix has condition number 5. This means that the direction of most curvature has five times more curvature than the direction of least curvature. In this case, the most curvature is in the direction $[1, 1]^\top$ and the least curvature is in the direction $[1, -1]^\top$. The red lines indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature. Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context.

$\mathbf{x} \in \mathbb{R}^2$ with $\mathbf{x}$ constrained to have exactly unit L^2 norm, we can instead minimize $g(\theta) = f([\cos \theta, \sin \theta]^\top)$ with respect to θ , then return $[\cos \theta, \sin \theta]$ as the solution to the original problem. This approach requires creativity; the transformation between optimization problems must be designed specifically for each case we encounter.

The **Karush–Kuhn–Tucker** (KKT) approach¹ provides a very general solution to constrained optimization. With the KKT approach, we introduce a new function called the **generalized Lagrangian** or **generalized Lagrange function**.

To define the Lagrangian, we first need to describe $\mathbb{S}$ in terms of equations and inequalities. We want a description of $\mathbb{S}$ in terms of m functions $g^{(i)}$ and n functions $h^{(j)}$ so that $\mathbb{S} = \{\mathbf{x} \mid \forall i, g^{(i)}(\mathbf{x}) = 0 \text{ and } \forall j, h^{(j)}(\mathbf{x}) \leq 0\}$. The equations involving $g^{(i)}$ are called the **equality constraints** and the inequalities involving $h^{(j)}$ are called **inequality constraints**.

We introduce new variables λ_i and α_j for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) + \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.14)$$

We can now solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. Observe that, so long as at least one feasible point exists and $f(\mathbf{x})$ is not permitted to have value ∞ , then

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}). \quad (4.15)$$

has the same optimal objective function value and set of optimal points $\mathbf{x}$ as

$$\min_{\mathbf{x} \in \mathbb{S}} f(\mathbf{x}). \quad (4.16)$$

This follows because any time the constraints are satisfied,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}), \quad (4.17)$$

while any time a constraint is violated,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty. \quad (4.18)$$

¹The KKT approach generalizes the method of **Lagrange multipliers** which allows equality constraints but not inequality constraints.

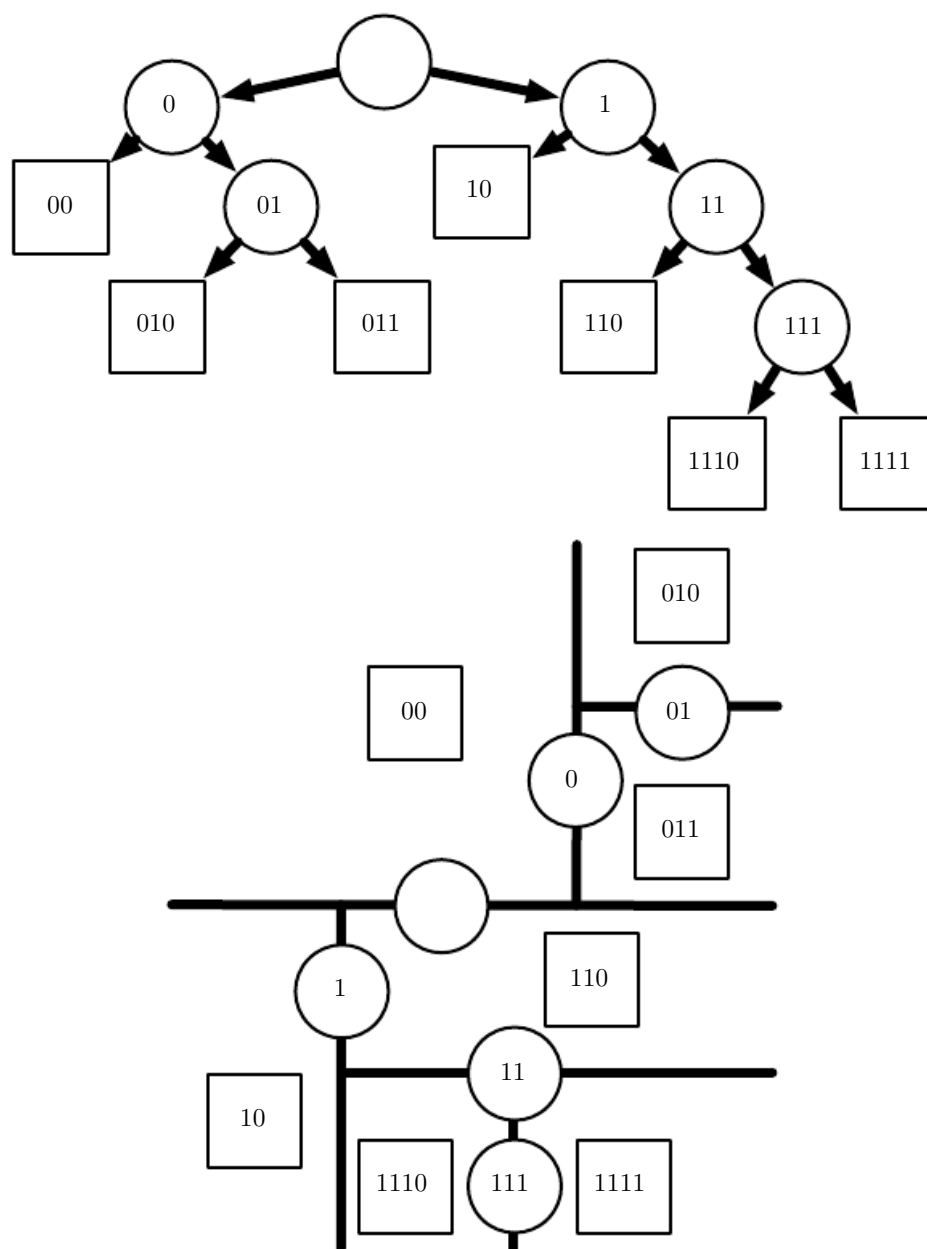


Figure 5.7: Diagrams describing how a decision tree works. (*Top*) Each node of the tree chooses to send the input example to the child node on the left (0) or to the child node on the right (1). Internal nodes are drawn as circles and leaf nodes as squares. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by appending a bit to its parent identifier (0=choose left or top, 1=choose right or bottom). (*Bottom*) The tree divides space into regions. The 2D plane shows how a decision tree might divide $\mathbb{R}^2$. The nodes of the tree are plotted in this plane, with each internal node drawn along the dividing line it uses to categorize examples, and leaf nodes drawn in the center of the region of examples they receive. The result is a piecewise-constant function, with one piece per leaf. Each leaf requires at least one training example to define, so it is not possible for the decision tree to learn a function that has more local maxima than the number of training examples.

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad (6.6)$$

and $b = 0$.

We can now walk through the way that the model processes a batch of inputs. Let $\mathbf{X}$ be the design matrix containing all four points in the binary input space, with one example per row:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \quad (6.7)$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \quad (6.8)$$

Next, we add the bias vector $\mathbf{c}$, to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.9)$$

In this space, all of the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of $\mathbf{h}$ for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.10)$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish by multiplying by the weight vector $\mathbf{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (6.11)$$

applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations that are chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

Modern software implementations are based on the generalized form of back-propagation described in section 6.5.6 below, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on the target $\mathbf{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J , the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\mathbf{W}$ and $\mathbf{b}$. For simplicity, this demonstration uses only a single input example $\mathbf{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l

Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model

Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model

Require: $\mathbf{x}$, the input to process

Require: $\mathbf{y}$, the target output

$\mathbf{h}^{(0)} = \mathbf{x}$

for $k = 1, \dots, l$ **do**

$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$

$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$

end for

$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$

$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$

6.5.5 Symbol-to-Symbol Derivatives

Algebraic expressions and computational graphs both operate on **symbols**, or variables that do not have specific values. These algebraic and graph-based representations are called **symbolic** representations. When we actually use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network $\mathbf{x}$ with a specific **numeric** value, such as $[1.2, 3.765, -1.8]^\top$.

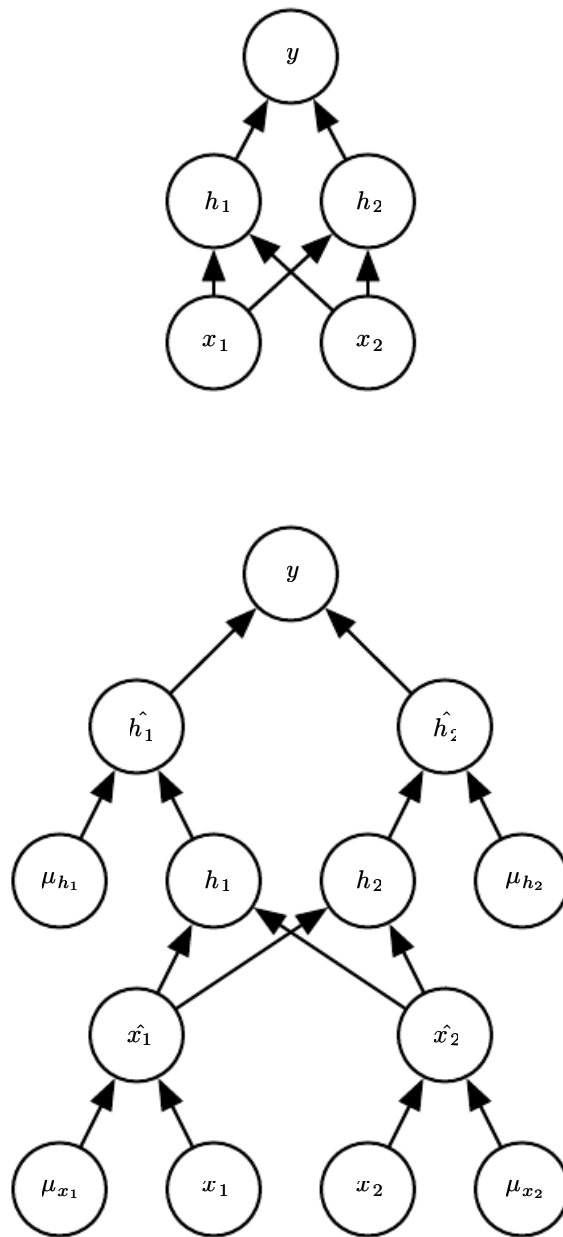


Figure 7.7: An example of forward propagation through a feedforward network using dropout. (*Top*) In this example, we use a feedforward network with two input units, one hidden layer with two hidden units, and one output unit. (*Bottom*) To perform forward propagation with dropout, we randomly sample a vector $\boldsymbol{\mu}$ with one entry for each input or hidden unit in the network. The entries of $\boldsymbol{\mu}$ are binary and are sampled independently from each other. The probability of each entry being 1 is a hyperparameter, usually 0.5 for the hidden layers and 0.8 for the input. Each unit in the network is multiplied by the corresponding mask, and then forward propagation continues through the rest of the network as usual. This is equivalent to randomly selecting one of the sub-networks from figure 7.6 and running forward propagation through it.

Chapter 12

Applications

In this chapter, we describe how to use deep learning to solve applications in computer vision, speech recognition, natural language processing, and other application areas of commercial interest. We begin by discussing the large scale neural network implementations required for most serious AI applications. Next, we review several specific application areas that deep learning has been used to solve. While one goal of deep learning is to design algorithms that are capable of solving a broad variety of tasks, so far some degree of specialization is needed. For example, vision tasks require processing a large number of input features (pixels) per example. Language tasks require modeling a large number of possible values (words in the vocabulary) per input feature.

12.1 Large-Scale Deep Learning

Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior. It truly is important to emphasize the fact that the number of neurons must be *large*. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use. As we saw in section 1.2.3, network sizes have grown exponentially for the past three decades, yet artificial neural networks are only as large as the nervous systems of insects.

Because the size of neural networks is of paramount importance, deep learning

loss functions, the gradient can be computed efficiently (Vincent *et al.*, 2015), but the standard cross-entropy loss applied to a traditional softmax output layer poses many difficulties.

Suppose that $\mathbf{h}$ is the top hidden layer used to predict the output probabilities $\hat{\mathbf{y}}$. If we parametrize the transformation from $\mathbf{h}$ to $\hat{\mathbf{y}}$ with learned weights $\mathbf{W}$ and learned biases $\mathbf{b}$, then the affine-softmax output layer performs the following computations:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \dots, |\mathbb{V}|\}, \quad (12.8)$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}. \quad (12.9)$$

If $\mathbf{h}$ contains n_h elements then the above operation is $O(|\mathbb{V}|n_h)$. With n_h in the thousands and $|\mathbb{V}|$ in the hundreds of thousands, this operation dominates the computation of most neural language models.

12.4.3.1 Use of a Short List

The first neural language models (Bengio *et al.*, 2001, 2003) dealt with the high cost of using a softmax over a large number of output words by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002) and Schwenk (2007) built upon this approach by splitting the vocabulary $\mathbb{V}$ into a **shortlist** $\mathbb{L}$ of most frequent words (handled by the neural net) and a tail $\mathbb{T} = \mathbb{V} \setminus \mathbb{L}$ of more rare words (handled by an n -gram model). To be able to combine the two predictions, the neural net also has to predict the probability that a word appearing after context C belongs to the tail list. This may be achieved by adding an extra sigmoid output unit to provide an estimate of $P(i \in \mathbb{T} \mid C)$. The extra output can then be used to achieve an estimate of the probability distribution over all words in $\mathbb{V}$ as follows:

$$\begin{aligned} P(y = i \mid C) = & 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L}) (1 - P(i \in \mathbb{T} \mid C)) \\ & + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C) \end{aligned} \quad (12.10)$$

where $P(y = i \mid C, i \in \mathbb{L})$ is provided by the neural language model and $P(y = i \mid C, i \in \mathbb{T})$ is provided by the n -gram model. With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.

An obvious disadvantage of the short list approach is that the potential generalization advantage of the neural language models is limited to the most frequent

by training with the squared error criterion

$$\|g(f(\tilde{\mathbf{x}})) - \mathbf{x}\|^2 \quad (14.16)$$

and corruption

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

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

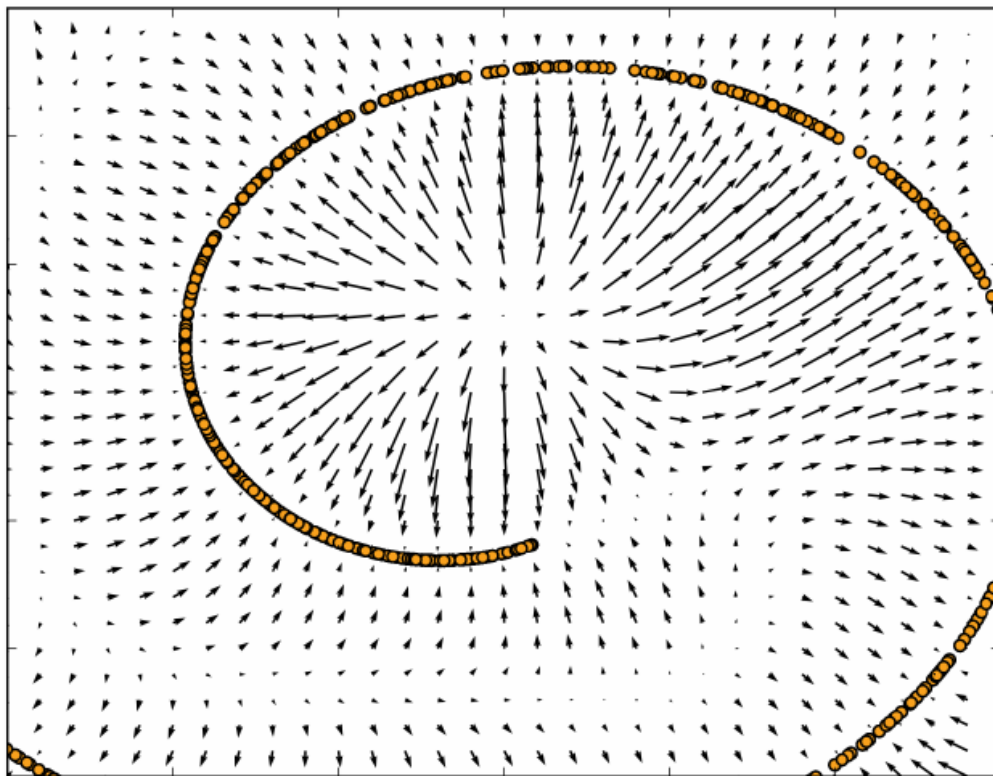


Figure 14.5: Vector field learned by a denoising autoencoder around a 1-D curved manifold near which the data concentrates in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the autoencoder and points towards higher probability according to the implicitly estimated 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 one-dimensional 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, it means that 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 \(2013\)](#).

In general, there is no guarantee that the reconstruction $g(f(\mathbf{x}))$ minus the input $\mathbf{x}$ corresponds to the gradient of any function, let alone to the score. That is

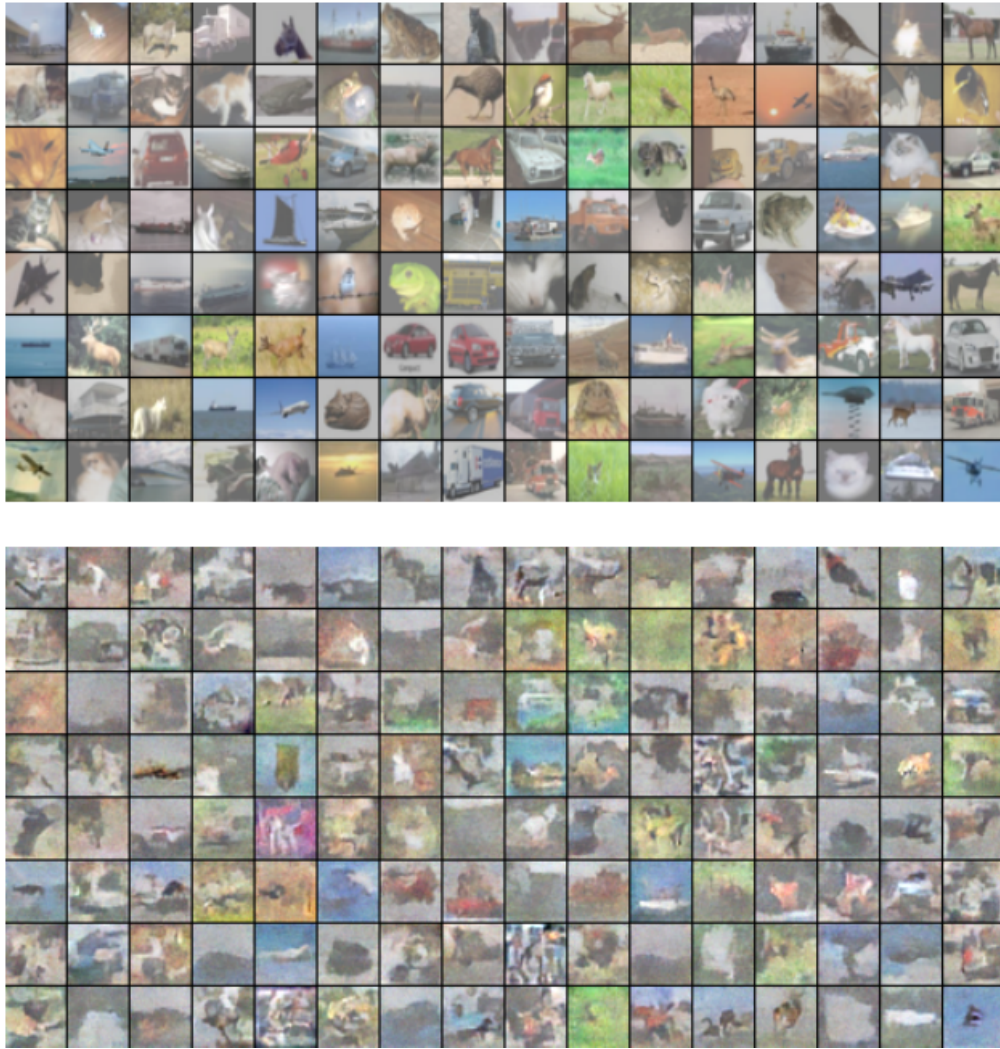


Figure 16.1: Probabilistic modeling of natural images. (*Top*) Example 32×32 pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). (*Bottom*) Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training example that is closest to it in Euclidean space. This comparison allows us to see that the model is truly synthesizing new images, rather than memorizing the training data. Contrast of both sets of images has been adjusted for display. Figure reproduced with permission from Courville *et al.* (2011).

learning problem defines an asymptotically consistent estimator of the original problem.

Specifically, we introduce a second distribution, the **noise distribution** $p_{\text{noise}}(\mathbf{x})$. The noise distribution should be tractable to evaluate and to sample from. We can now construct a model over both $\mathbf{x}$ and a new, binary class variable y . In the new joint model, we specify that

$$p_{\text{joint}}(y = 1) = \frac{1}{2}, \quad (18.29)$$

$$p_{\text{joint}}(\mathbf{x} \mid y = 1) = p_{\text{model}}(\mathbf{x}), \quad (18.30)$$

and

$$p_{\text{joint}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x}). \quad (18.31)$$

In other words, y is a switch variable that determines whether we will generate $\mathbf{x}$ from the model or from the noise distribution.

We can construct a similar joint model of training data. In this case, the switch variable determines whether we draw $\mathbf{x}$ from the **data** or from the noise distribution. Formally, $p_{\text{train}}(y = 1) = \frac{1}{2}$, $p_{\text{train}}(\mathbf{x} \mid y = 1) = p_{\text{data}}(\mathbf{x})$, and $p_{\text{train}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x})$.

We can now just use standard maximum likelihood learning on the **supervised** learning problem of fitting p_{joint} to p_{train} :

$$\boldsymbol{\theta}, c = \arg \max_{\boldsymbol{\theta}, c} \mathbb{E}_{\mathbf{x}, y \sim p_{\text{train}}} \log p_{\text{joint}}(y \mid \mathbf{x}). \quad (18.32)$$

The distribution p_{joint} is essentially a logistic regression model applied to the difference in log probabilities of the model and the noise distribution:

$$p_{\text{joint}}(y = 1 \mid \mathbf{x}) = \frac{p_{\text{model}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})} \quad (18.33)$$

$$= \frac{1}{1 + \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}} \quad (18.34)$$

$$= \frac{1}{1 + \exp\left(\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right)} \quad (18.35)$$

$$= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right) \quad (18.36)$$

$$= \sigma(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})). \quad (18.37)$$

Algorithm 20.1 The variational stochastic maximum likelihood algorithm for training a DBM with two hidden layers.

Set ϵ , the step size, to a small positive number

Set k , the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta} + \epsilon \Delta_{\boldsymbol{\theta}})$ to burn in, starting from samples from $p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta})$.

Initialize three matrices, $\tilde{\mathbf{V}}$, $\tilde{\mathbf{H}}^{(1)}$ and $\tilde{\mathbf{H}}^{(2)}$ each with m rows set to random values (e.g., from Bernoulli distributions, possibly with marginals matched to the model's marginals).

while not converged (learning loop) **do**

Sample a minibatch of m examples from the training data and arrange them as the rows of a design matrix $\mathbf{V}$.

Initialize matrices $\hat{\mathbf{H}}^{(1)}$ and $\hat{\mathbf{H}}^{(2)}$, possibly to the model's marginals.

while not converged (mean field inference loop) **do**

$$\hat{\mathbf{H}}^{(1)} \leftarrow \sigma \left(\mathbf{V} \mathbf{W}^{(1)} + \hat{\mathbf{H}}^{(2)} \mathbf{W}^{(2)\top} \right).$$

$$\hat{\mathbf{H}}^{(2)} \leftarrow \sigma \left(\hat{\mathbf{H}}^{(1)} \mathbf{W}^{(2)} \right).$$

end while

$$\Delta_{\mathbf{W}^{(1)}} \leftarrow \frac{1}{m} \mathbf{V}^\top \hat{\mathbf{H}}^{(1)}$$

$$\Delta_{\mathbf{W}^{(2)}} \leftarrow \frac{1}{m} \hat{\mathbf{H}}^{(1)\top} \hat{\mathbf{H}}^{(2)}$$

for $l = 1$ to k (Gibbs sampling) **do**

Gibbs block 1:

$$\forall i, j, \tilde{V}_{i,j} \text{ sampled from } P(\tilde{V}_{i,j} = 1) = \sigma \left(\mathbf{W}_{j,:}^{(1)} \left(\tilde{\mathbf{H}}_{i,:}^{(1)} \right)^\top \right).$$

$$\forall i, j, \tilde{H}_{i,j}^{(2)} \text{ sampled from } P(\tilde{H}_{i,j}^{(2)} = 1) = \sigma \left(\tilde{\mathbf{H}}_{i,:}^{(1)} \mathbf{W}_{:,j}^{(2)} \right).$$

Gibbs block 2:

$$\forall i, j, \tilde{H}_{i,j}^{(1)} \text{ sampled from } P(\tilde{H}_{i,j}^{(1)} = 1) = \sigma \left(\tilde{\mathbf{V}}_{i,:} \mathbf{W}_{:,j}^{(1)} + \tilde{\mathbf{H}}_{i,:}^{(2)} \mathbf{W}_{j,:}^{(2)\top} \right).$$

end for

$$\Delta_{\mathbf{W}^{(1)}} \leftarrow \Delta_{\mathbf{W}^{(1)}} - \frac{1}{m} \mathbf{V}^\top \tilde{\mathbf{H}}^{(1)}$$

$$\Delta_{\mathbf{W}^{(2)}} \leftarrow \Delta_{\mathbf{W}^{(2)}} - \frac{1}{m} \tilde{\mathbf{H}}^{(1)\top} \tilde{\mathbf{H}}^{(2)}$$

$\mathbf{W}^{(1)} \leftarrow \mathbf{W}^{(1)} + \epsilon \Delta_{\mathbf{W}^{(1)}}$ (this is a cartoon illustration, in practice use a more effective algorithm, such as momentum with a decaying learning rate)

$$\mathbf{W}^{(2)} \leftarrow \mathbf{W}^{(2)} + \epsilon \Delta_{\mathbf{W}^{(2)}}$$

end while

binarization step share a file containing the results of the random binarization, so that there is no difference in results based on different outcomes of the binarization step.

Because being able to generate realistic samples from the data distribution is one of the goals of a generative model, practitioners often evaluate generative models by visually inspecting the samples. In the best case, this is done not by the researchers themselves, but by experimental subjects who do not know the source of the samples (Denton *et al.*, 2015). Unfortunately, it is possible for a very poor probabilistic model to produce very good samples. A common practice to verify if the model only copies some of the training examples is illustrated in figure 16.1. The idea is to show for some of the generated samples their nearest neighbor in the training set, according to Euclidean distance in the space of $\mathbf{x}$. This test is intended to detect the case where the model overfits the training set and just reproduces training instances. It is even possible to simultaneously underfit and overfit yet still produce samples that individually look good. Imagine a generative model trained on images of dogs and cats that simply learns to reproduce the training images of dogs. Such a model has clearly overfit, because it does not produce images that were not in the training set, but it has also underfit, because it assigns no probability to the training images of cats. Yet a human observer would judge each individual image of a dog to be high quality. In this simple example, it would be easy for a human observer who can inspect many samples to determine that the cats are absent. In more realistic settings, a generative model trained on data with tens of thousands of modes may ignore a small number of modes, and a human observer would not easily be able to inspect or remember enough images to detect the missing variation.

Since the visual quality of samples is not a reliable guide, we often also evaluate the log-likelihood that the model assigns to the test data, when this is computationally feasible. Unfortunately, in some cases the likelihood seems not to measure any attribute of the model that we really care about. For example, real-valued models of MNIST can obtain arbitrarily high likelihood by assigning arbitrarily low variance to background pixels that never change. Models and algorithms that detect these constant features can reap unlimited rewards, even though this is not a very useful thing to do. The potential to achieve a cost approaching negative infinity is present for any kind of maximum likelihood problem with real values, but it is especially problematic for generative models of MNIST because so many of the output values are trivial to predict. This strongly suggests a need for developing other ways of evaluating generative models.

Theis *et al.* (2015) review many of the issues involved in evaluating generative