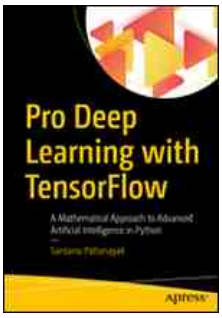


Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

by Santanu Pattanayak

Apress. (c) 2017. Copying Prohibited.

Reprinted for 2362626 2362626, Indiana University

none@books24x7.com

Reprinted with permission as a subscription benefit of **Skillport**,

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 5: Unsupervised Learning with Restricted Boltzmann Machines and Auto-encoders

© Santanu Pattanayak 2017
S. Pattanayak, *Pro Deep Learning with TensorFlow*, https://doi.org/10.1007/978-1-4842-3096-1_5

Overview

Unsupervised learning is a branch of machine learning that tries to find hidden structures within unlabeled data and derive insights from it. Clustering, data dimensionality-reduction techniques, noise reduction, segmentation, anomaly detection, fraud detection, and other rich methods rely on unsupervised learning to drive analytics. Today, with so much data around us, it is impossible to label all data for supervised learning. This makes unsupervised learning all the more important. Restricted Boltzmann machines and auto-encoders are unsupervised methods that are based on artificial neural networks. They have a wide range of uses in data compression and dimensionality reduction, noise reduction from data, anomaly detection, generative modeling, collaborative filtering, and initialization of deep neural networks, among other things. We will go through these topics in detail and then touch upon a couple of unsupervised pre-processing techniques for images, namely PCA (principal component analysis) whitening and ZCA (Mahalanobis) whitening. Also, since restricted Boltzmann machines use sampling techniques during training, I have briefly touched upon Bayesian inference and Markov Chain Monte Carlo sampling for reader's benefit.

Boltzmann Distribution

Restricted Boltzmann machines are energy models based on the Boltzmann Distribution Law of classical physics, where the state of particles of any system is represented by their generalized coordinates and velocities. These generalized coordinates and velocities form the phase space of the particles, and the particles can be in any location in the phase space with specific energy and probability. Let's consider a classical system that contains N gas molecules and let the generalized position and velocity of any particle be represented by $r \in \mathbb{R}^{3\times1}$ and $v \in \mathbb{R}^{3\times1}$ respectively. The location of the particle in the phase space can be represented by (r, v). Every such possible value of (r, v) that the particle can take is called a *configuration* of the particle. Further, all the N particles are identical in the sense that they are equally likely to take up any state. Given such a system at thermodynamic temperature T, the probability of any such configuration is given as follows:

$$P(Data/\theta) = P(x_1x_2x_3x_4x_5x_6/\theta) = \prod_{i=1}^{6} P(x_i/\theta)$$

$E(r, v)$ is the energy of any particle at configuration (r, v), and K is the Boltzmann Constant. Hence, we see that the probability of any configuration in the phase space is proportional to the exponential of the negative of the energy divided by the product of the Boltzmann Constant and the thermodynamic temperature. To convert the relationship into an equality, the probability needs to be normalized by the sum of the probabilities of all the possible configurations. If there are M possible phase-space configurations for the particles, then the probability of any generalized configuration (r, v) can be expressed as

$$P(x = j/\theta) = \theta^{j}(1-\theta)^{(1-j)} \quad \forall j \in \{0,1\}$$

where Z is the partition function given by

$$L(\theta) = \theta^{5}(1-\theta)$$

There can be several values of r and v separately. However, M denotes all the unique combinations of r and v possible, which have been denoted by $(r, v)_i$ in the preceding equation. If r can take up n distinct coordinate values whereas v can take up m distinct velocity values, then the total number of possible configurations $M = n \times m$. In such cases, the partition function can also be expressed as follows:

$$P(\theta) = \frac{\theta(1-\theta)}{6}$$

The thing to note here is that the probability of any configuration is higher when its associated energy is low. For the gas molecules, it's intuitive as well given that high-energy states are always associated with unstable equilibrium and hence are less likely to retain the high-energy configuration for long. The particles in the high-energy configuration will always be in a pursuit to occupy much more stable low-energy states.

If we consider two configurations $s_1 = (r_1, v_1)$ and $s_2 = (r_2, v_2)$, and if the number of gas molecules in these two states are N_1,

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

and N_2 respectively, then the probability ratio of the two states is a function of the energy difference between the two states:

$$P(\theta / D) = \frac{\theta^6 (1-\theta)^2}{252}$$

We will digress a little now and briefly discuss Bayesian inference and Markov Chain Monte Carlo (MCMC) methods since restricted Boltzmann machines use sampling through MCMC techniques, especially Gibbs sampling, and some knowledge of these would go a long way toward helping the readers appreciate the working principles of restricted Boltzmann machines.

Bayesian Inference: Likelihood, Priors, and Posterior Probability Distribution

As discussed in Chapter 1, whenever we get data we build a model by defining a likelihood function over the data conditioned on the model parameters and then try to maximize that likelihood function. Likelihood is nothing but the probability of the seen or observed data given the model parameters:

$$P(B)P(A/B)$$

To get the model defined by its parameters we maximize the likelihood of the seen data:

$$P(A)(B/A) = P(B)P(A/B)$$

Since we are only trying to fit a model based on the observed data, there is a high chance of overfitting and not generalizing to new data if we go for simple likelihood maximization.

If the data size is huge the seen data is likely to represent the population well and so maximizing the likelihood may suffice. On the other hand, if the seen data is small, there is a high chance that it might not represent the overall population well, and thus the model based on likelihood would not generalize well to new data. In that case, having a certain prior belief over the model and constraining the likelihood by that prior belief would lead to better results. Let's say that the prior belief is in the form of our knowing the uncertainty over the model parameters in the form of a probability distribution; i.e., *P(Model)* is known. We can in that case update our likelihood by the prior information to get a distribution over the model given the data. As per Bayes' Theorem of Conditional Probability,

$$P(v,h) = \frac{e^{-E(h,v)}}{Z}$$

P(Model/Data) is called the *posterior distribution* and is generally more informative since it combines one's prior knowledge about the data or model. Since this probability of data is independent of the model, the posterior is directly proportional to the product of the likelihood and the prior:

$$Z = \sum_v \sum_h e^{-E(v,h)}$$

One can build a model by maximizing the posterior probability distribution instead of the likelihood. This method of obtaining the model is called Maximize a Posterior, or MAP. Both likelihood and MAP are point estimates for the models and thus don't cover the whole uncertainty space. Taking the model that maximizes the posterior means taking the mode of the probability distribution of the model. Point estimates given by the maximum likelihood function don't correspond to any mode, since likelihood is not a probability-distribution function. If the probability distribution turns out to be multi-modal, these point estimates are going to perform even more poorly.

A better approach is to take the average of the model over the whole uncertainty space; i.e., to take the mean of the model based on the posterior distribution, as follows:

$$E(v,h) = -b^T v - c^T h - v^T W h$$

To motivate the ideas of likelihood and posterior and how they can be used to derive model parameters, let's get back to the coin problem again.

Suppose we toss a coin six times, out of which heads appears five times. If one is supposed to estimate the probability of heads, what would the estimate be?

Here, the model for us is to estimate the probability of heads θ in a throw of a coin. Each toss of a coin can be treated as an independent Bernoulli trial with the probability of heads being θ. The likelihood of the data, given the model, is given by

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

$$P(v,h)=\frac{e^{b^Tv+c^Th+v^TWh}}{Z}$$

where x_i $\forall i \in \{1,2,3,4,5,6\}$ denotes the event of either heads (*H*) or tails (*T*).

Since the throws of coins are independent, the likelihood can be factorized as follows:

(5.1.1) $$P(Data/\theta)=P(x_1x_2x_3x_4x_5x_6/\theta)=\prod_{i=1}^{6}P(x_i/\theta)$$

Each throw of dice follows the Bernoulli distribution, hence the probability of heads is θ and the probability of tails is $(1-\theta)$, and in general its probability mass function is given by

(5.1.2) $$P(x=j/\theta)=\theta^j(1-\theta)^{(1-j)} \ \forall j\in\{0,1\}$$

where $j = 1$ denotes heads and $j = 0$ denotes tails.

Since there are 5 heads and 1 tails combining (1) and (2), the likelihood *L* as a function of θ can be expressed as follows:

(5.1.3) $$L(\theta)=\theta^5(1-\theta)$$

Maximum likelihood methods treat the $\hat{\theta}$ that minimizes $L(\theta)$ as the model parameter. Hence,

$$e^{c^Th}e^{v^TWh}=\prod_{j=1}^{n}e^{c_jh_j+v^TW[:,j]h_j}$$

If we take the derivative of the computed likelihood in (5.1.3) and set it to zero, we will arrive at the likelihood estimate for θ:

$$\sum_h e^{c^Th}e^{v^TWh}=\sum_h\prod_{j=1}^{n}e^{c_jh_j+v^TW[:,j]h_j}$$

In general, if someone asks us our estimate of θ without our doing a similar maximization of likelihood, we at once answer the probability to be $\frac{5}{6}$ by the basic definition of probability that we learned in high school; i.e.,

$$\begin{aligned}\sum_h e^{c^Th}e^{v^TWh}&=\sum_{h_1=0}^{1}\sum_{h_2=0}^{1}..\sum_{h_n=0}^{1}\prod_{j=1}^{n}e^{c_jh_j+v^TW[:,j]h_j}\\&=\sum_{h_1=0}^{1}\sum_{h_2=0}^{1}..\sum_{h_n=0}^{1}\left(e^{c_1h_1+v^TW[:,1]h_1}\right)\left(e^{c_2h_2+v^TW[:,2]h_2}\right)...\left(e^{c_nh_n+v^TW[:,n]h_n}\right)\end{aligned}$$

In a way, our head is thinking about likelihood and relying on the data seen thus far.

Now, let's suppose had we not seen the data and someone asked us to determine the probability of heads; what would have been a logical estimate?

Well, it depends on any prior belief that we hold about the coin that the probabilities would differ. If we assumed a fair coin, which in general is the most obvious assumption to make given that we have no information about the coin, $\theta=\frac{1}{2}$ would have been a good estimate. However, when we are assuming instead of doing a point estimate of prior for θ, it's better to have a probability distribution over θ with the probability maximum at $\theta=\frac{1}{2}$. The prior probability distribution is a distribution over the model parameter θ.

A Beta distribution with parameters $\alpha = 2$, $\beta = 2$ would be a good prior distribution in this case since it has a maximum probability at $\theta=\frac{1}{2}$ and is symmetrical aound it.

$$P(\theta)=Beta(\alpha=2,\beta=2)=\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}=\frac{\theta(1-\theta)}{B(\alpha,\beta)}$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

For fixed values of α and β, $B(\alpha, \beta)$ is constant and is the normalizing or partition function to this probability distribution. It can be computed as follows:

$$P(h_1 h_2 .. h_n / v) = P(h_1 / v) P(h_2 / v) .. P(h_n / v)$$

Even if one doesn't remember the formula, it can be found out by just integrating $\theta(1 - \theta)$ and taking the reciprocal of the same as the normalizing constant since the integral of the probability distribution should be 1.

(5.1.4) $$P(\theta) = \frac{\theta(1-\theta)}{6}$$

If we combine the likelihood and the prior, we get the posterior probability distribution as follows:

$$P(h_j / v) = \frac{e^{c_j h_j + v^T W[:,j] h_j}}{\sum_{h_j=0}^{1} e^{c_j h_j + v^T W[:,j] h_j}}$$

The proportional sign comes since we have ignored the probability of data. In fact, we can take the 6 out as well and express the posterior as follows:

$$P(h_j = 1 / v) = \frac{e^{c_j + v^T W[:,j]}}{1 + e^{c_j + v^T W[:,j]}}$$

Now, $0 \leq \theta \leq 1$ since θ is a probability. Integrating $\theta^6 (1 - \theta)^2$ in the range of 0 to 1 and taking the reciprocal would give us the normalizing factor of the posterior, which comes out to be 252. Hence, the posterior can be expressed as follows:

(5.1.5) $$P(\theta / D) = \frac{\theta^6 (1-\theta)^2}{252}$$

Now that we have the posterior, there are two ways we can estimate θ. We can maximize the posterior and get a MAP estimate of θ as follows:

$$P(h_j = 0 / v) = \frac{1}{1 + e^{c_j + v^T W[:,j]}}$$
$$P(h_j = 1 / v) = \sigma(c_j + v^T W[:, j])$$

We see the MAP estimate of $\frac{3}{4}$ is more conservative than the likelihood estimate of $\frac{5}{6}$ since it takes the prior into consideration and doesn't blindly believe the data.

Now, let's look at the second approach, the pure Bayesian approach, and take the mean of the posterior distribution to average over all the uncertainties for θ:

$$P(v_i = 1 / h) = \sigma\left(b_i + \sum_{j=1}^{n} h_j w_{ij}\right)$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

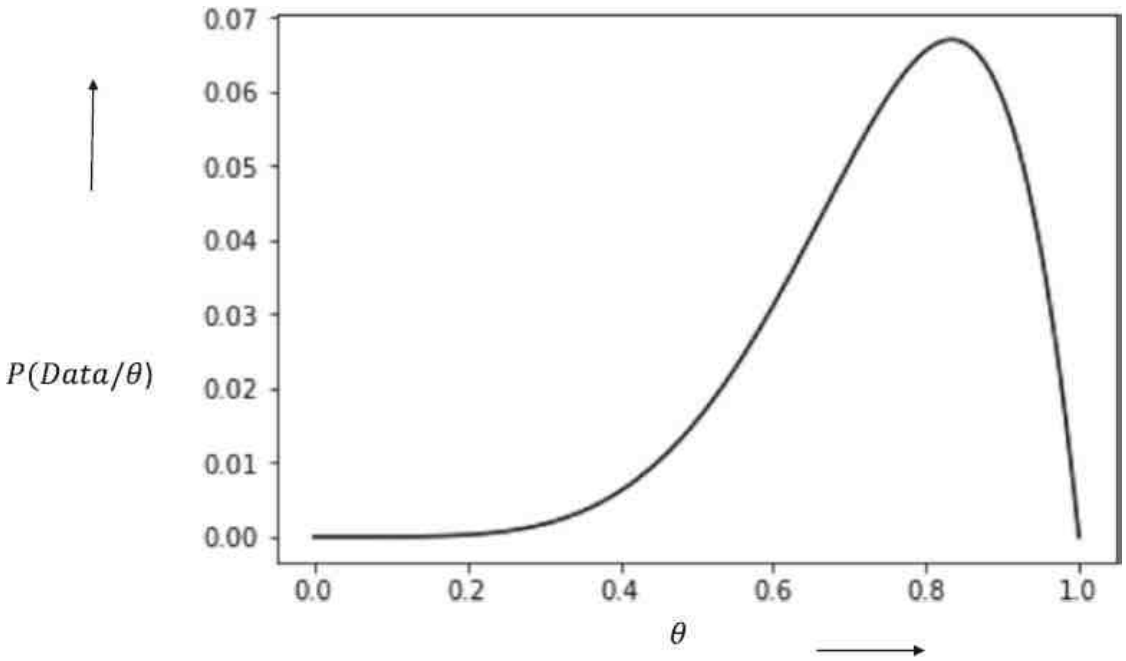


Figure 5-1a: Likelihood function plot

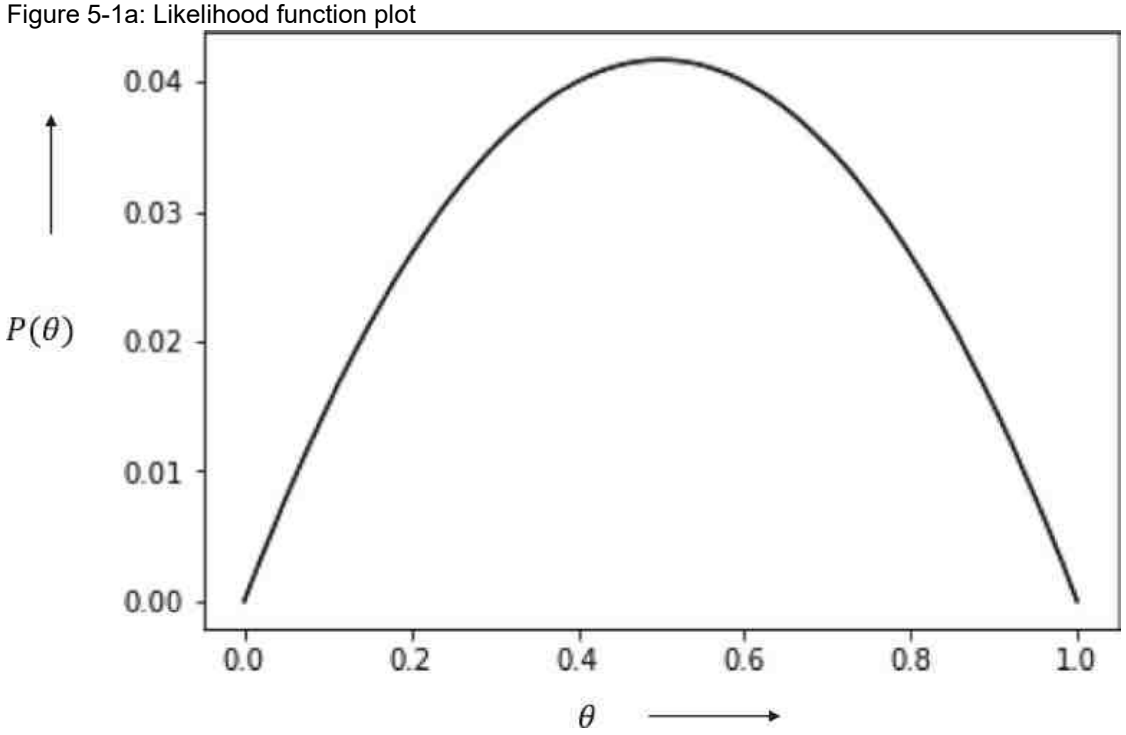


Figure 5-1b: Prior probability distribution

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

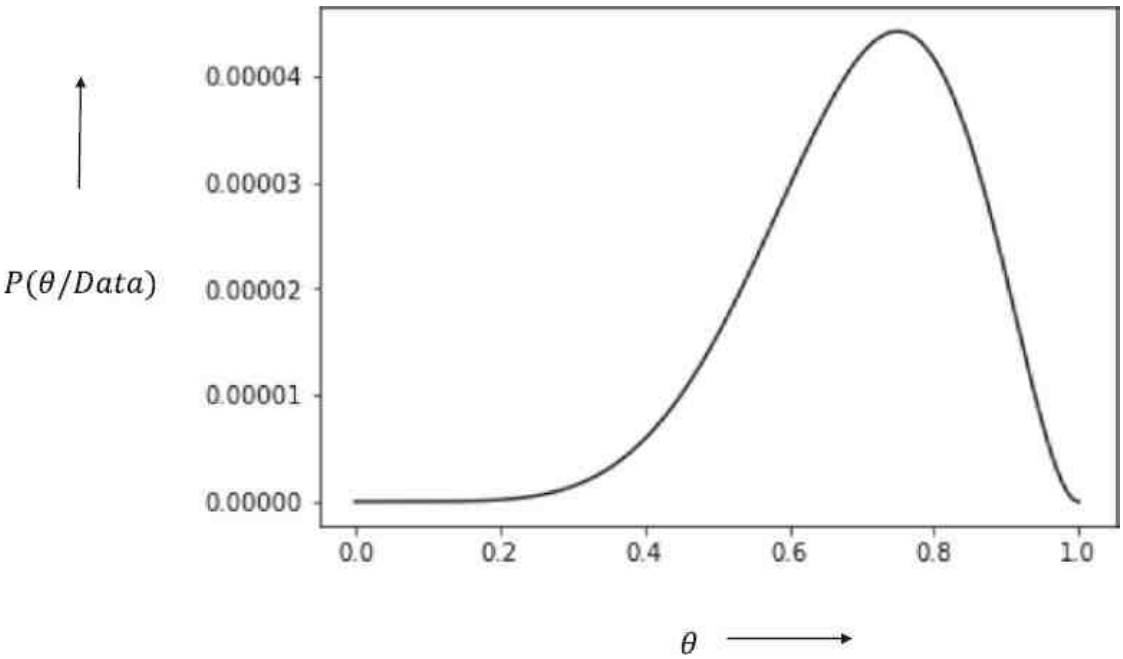


Figure 5-1c: Posterior probability distribution

Plotted in Figure 5-1a through Figure 5-1c are the likelihood function and the prior and posterior probability distributions for the coin problem. One thing to note is the fact that the likelihood function is not a probability density function or a probability mass function, whereas the prior and posteriors are probability mass or density functions.

For complicated distributions, the posterior probability distribution can turn out to be very complex with several parameters and is unlikely to represent known probability distribution forms such as normal, gamma, and so on. Thus, it may become seemingly impossible to compute the integral over the whole uncertainty space of the model in order to compute the mean of the posterior.

Markov Chain Monte Carlo methods of sampling can be used in such cases to sample model parameters, and then their mean is a fair estimate of the mean of the posterior distribution. If we sample n sets of model parameters M_i then

$$L(\theta) = P\left(v^{(1)}/\theta\right)P\left(v^{(2)}/\theta\right)..P\left(v^{(m)}/\theta\right) = \prod_{t=1}^{m} P\left(v^{(t)}/\theta\right)$$

Generally, the mean of the distribution is taken since it minimizes the squared error that is of all c.

$E\left[(y - c)^2\right]$ is minimized when $c = E[y]$. Given that we are trying to represent the probability of the distribution by a single representative such that the squared error over the probability distribution is minimized, mean is the best candidate.

However, one can take the median of the distribution if the distribution is skewed and/or there is more noise in the data in the form of potential outliers. This estimated median can be based on the samples drawn from the posterior.

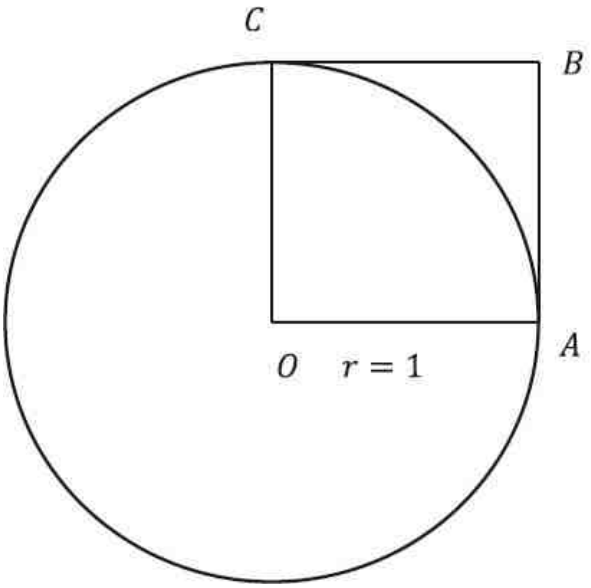
Markov Chain Monte Carlo Methods for Sampling

Markov Chain Monte Carlo methods, or MCMC, are some of the most popular techniques for sampling from complicated posterior probability distributions or in general from any probability distribution for multi-variate data. Before we get to MCMC, let's talk about Monte Carlo sampling methods in general. Monte Carlo sampling methods try to compute the area under a curve based on sampled points.

For example, the area of the transcendental number $Pi(\pi)$ can be computed by sampling points within a square of radius 1 and noting down the number of sampled points within one-fourth of the circle of diameter 2 enclosed within the square. As shown in Figure 5-2, the area of Pi can be computed as follows:

$$C = logL(\theta) = \sum_{t=1}^{m} logP\left(v^{(t)}/\theta\right)$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited



$$\pi = \frac{4 * Area(OAC)}{Area(OABC)}$$

Figure 5-2: Area of Pi

In Listing 5-1, the Monte Carlo method for computing the value of *Pi* is illustrated. As we can see, the value comes out to nearly the value of *Pi*. The accuracy can be improved by sampling more points.

Listing 5-1: Computation of Pi Through Monte Carlo Sampling

```
import numpy as np
number_sample = 100000
inner_area,outer_area = 0,0
for i in range(number_sample):
    x = np.random.uniform(0,1)
    y = np.random.uniform(0,1)
    if (x**2 + y**2) < 1 :
        inner_area += 1
    outer_area += 1

print("The computed value of Pi:",4*(inner_area/float(outer_area)))

--Output--
('The computed value of Pi:', 3.142)
```

The simple Monte Carlo method is highly inefficient if the dimension space is large since the larger the dimensionality is the more prominent the effects of correlation are. Markov Chain Monte Carlo methods are efficient in such scenarios since they spend more time collecting samples from high-probability regions than from lower-probability regions. The normal Monte Carlo method explores the probability space uniformly and hence spends as much time exploring low-probability zones as it does high-probability zones. As we know, the contribution of a low-probability zone is insignificant when computing the expectation of functions through sampling, and hence when an algorithm spends a lot of time in such a zone it leads to significantly higher processing time. The main heuristic behind the Markov Chain Monte Carlo method is to explore the probability space not uniformly but rather to concentrate more on the high-probability zones. In high-dimensional space, because of correlation, most of the space is sparse, with high density found only at specific areas. So, the idea is to spend more time and collect more samples from those high-probability zones and spend as little time as possible exploring low-probability zones.

Markov Chain can be thought of as a stochastic/random process to generate a sequence of random samples evolving over time. The next value of the random variable is only determined by the prior value of the variable. Markov Chain, once it enters a high-probability zone, tries to collect as many points with a high-probability density as possible. It does so by generating the next sample, conditioned on the current sample value, so that points near the current sample are chosen with high probability and points far away are chosen with low probability. This ensures that the Markov Chain collects as many points as possible from a current high-probability zone. However, occasionally a long jump from the current sample is required to explore other potential high-probability zones far from the current zone where the Markov Chain is working.

The Markov Chain concept can be illustrated with the movement of gas molecules in an enclosed container at a steady state. A few parts of the container have a higher density of gas molecules than the other areas, and since the gas molecules are at a steady state, the probabilities of each state (determined by the position of a gas molecule) would remain constant even though there might be gas molecules moving from one position to another.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

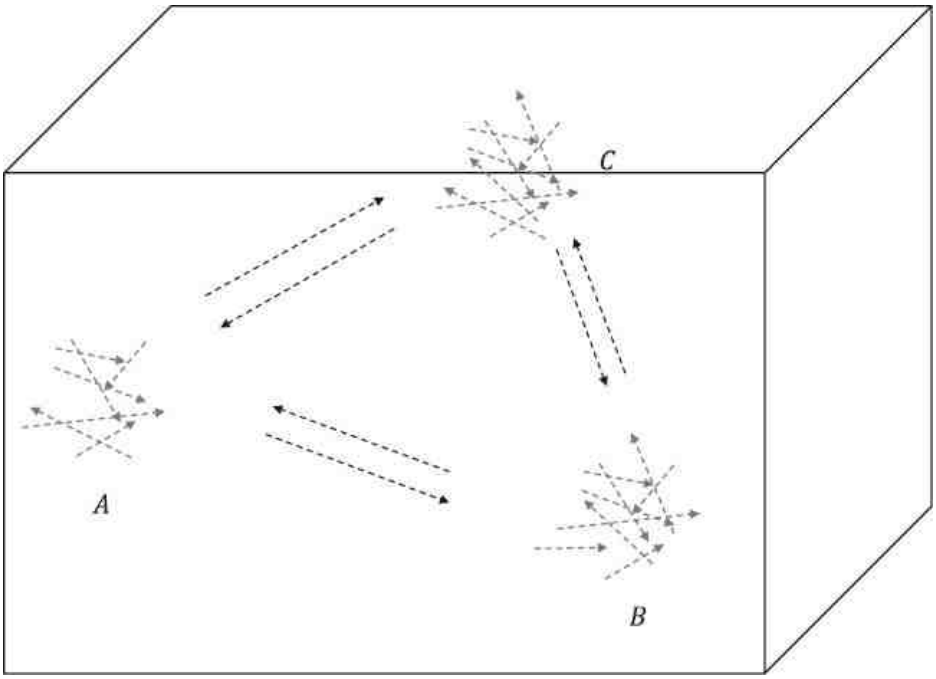


Figure 5-3: Movement of gases in an enclosed container at steady state with only three states: A, B, and C

For simplicity's sake, let us assume there are only three states (position of the gas molecules, in this case) for the gas molecules, as shown in Figure 5-3. Let us denote those states by *A, B*, and *C* and their corresponding probabilities by P_A, P_B, and P_C.

Since the gas molecules are in steady state, if there are gas molecules transitioning to other states, equilibrium needs to be maintained to keep the probability distribution stationary. The simplest assumption to consider is that probability mass going from state *A* to state *B* should come back to *A* from *B;* i.e., pair-wise, the states are in equilibrium.

Let's say *P*(*B*/*A*) determines the transition probability from *A* to *B*. So, probability mass going from *A* to *B* is given by

(5.2.1) $P(A)(B/A)$

Likewise, probability mass coming to *A* from *B* is given by

(5.2.2) $P(B)P(A/B)$

So, in steady state from (5.2.1) and (5.2.2), we have

(5.2.3) $P(A)(B/A)=P(B)P(A/B)$

to maintain the stationarity of the probability distribution. This is called a *detailed balance condition*, and it is a sufficient but not necessary condition for the stationarity of a probability distribution. The gas molecules can be in equilibrium in more complex ways, but since this form of detail balance is mathematically convenient when the possible state space is infinite, this approach has been widely used in Markov Chain Monte Carlo methods to sample the next point based on the current point and has a high acceptance probability. In short, movement of the Markov Chain is expected to behave like gas molecules at steady state spending more time in high probability region than in low probability keeping the detailed balance condition intact.

A few other conditions that need to be satisfied for a good implementation of Markov Chain are listed here:

- **Irreducibility** — A desirable property of the Markov Chain is that we can go from one state to any other state. This is important since in Markov Chain, although we want to keep exploring nearby states of a given state with high probability, at times we might have to take a jump and explore some far neighborhood with the expectation that the new zone might be another high-probability zone.
- **Aperiodicity** — The Markov Chain shouldn't repeat too often, as otherwise it won't be possible to traverse the whole space. Imagine a space with 20 states. If, after exploring five states, the chain repeats, it would not be possible to traverse all 20 states, thus leading to sub-optimal sampling.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Metropolis Algorithm

The Metropolis algorithm is a Markov Chain Monte Carlo method that uses the current accepted state to determine the next state. A sample at time (t +1) is conditionally dependent upon the sample at time t. The proposed state at time (t +1) is drawn from a normal distribution with a mean equal to the current sample at time t with a specified variance. Once drawn, the ratio of the probability is checked between the sample at time (t +1) and time t. If $P(x^{(t+1)})/(Px^{(t)})$ is greater than or equal to 1 then the sample $x^{(t+1)}$ is chosen with a probability of 1; if it is less than 1 then the sample is chosen randomly. Mentioned next are the detailed implementation steps.

- Start with any random sample point $X^{(1)}$.
- Choose the next point $X^{(2)}$ that is conditionally dependent on $X^{(1)}$. You can choose $X^{(2)}$ from a normal distribution with a mean of $X^{(1)}$ and some finite variance, let's say $S^{(2)}$. So, $X^{(2)} \sim Normal\ (X^{(1)}, S^2)$. A key deciding factor for good sampling is to choose the variance S^2 very judicially. The variance shouldn't be too large, since in that case the next sample $X^{(2)}$ has less of a chance of staying near the current sample $X^{(1)}$, in which case a high-probability region might not be explored as much since the next sample is selected far away from the current sample most of the time. At the same time, the variance shouldn't be too small. In such cases, the next samples would almost always stay near the current point and hence the probability of exploring a different high-probability zone far from the current zone would reduce.
- Some special heuristics are used in determining whether to accept $X^{(2)}$ once it has been generated from the preceding step.
    - If the ratio $P(X^{(2)})/P(X^{(1)}) >= 1$ then accept $X^{(2)}$ and keep it as a valid sample point. The accepted sample becomes the $X^{(1)}$ for generating the next sample.
    - If the ratio $P(X^{(2)})/P(X^{(1)}) < 1$, $X^{(2)}$ is accepted if the ratio is greater than a randomly generated number from the uniform distribution between 0 and 1; i.e., U [0, 1].

As we can see, if we move to a higher-probability sample then we accept the new sample, and if we move to a lower-probability sample we sometimes accept and sometimes reject the new sample. The probability of rejection increases if the ratio $P(X^{(2)})/P(X^{(1)})$ is small. Let's say the ratio of $P(X^{(2)})/P(X^{(1)}) = 0.1$. When we generate a random number r_u between 0 and 1 from a uniform distribution, then the probability of $r_u > 0.1$ is 0.9, which in turn implies that the probability of the new sample's getting rejected is 0.9. In general,

$$
\begin{aligned}
C &= \sum_{t=1}^{m} \log P\left(v^{(t)}/\theta\right) \\
&= \sum_{t=1}^{m} \log \sum_{h} P\left(v^{(t)}, h/\theta\right) \\
&= \sum_{t=1}^{m} \log \sum_{h} \frac{e^{-E\left(v^{(t)}, h\right)}}{Z} \\
&= \sum_{t=1}^{m} \log \frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}{Z} \\
&= \sum_{t=1}^{m} \log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - \sum_{t=1}^{m} \log Z \\
&= \sum_{t=1}^{m} \log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - m \log Z
\end{aligned}
$$

where r is the ratio of the probability of the new sample and the old sample.

Let's try to intuit why such heuristics work for Markov Chain Monte Carlo methods. As per detailed balance,

$$C = \sum_{t=1}^{m} log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - m log \sum_{v} \sum_{h} e^{-E(v,h)}$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

We are assuming that the transition probabilities follow normal distribution. We are not checking whether the transition probability framework we have taken is enough to maintain the stationarity of the probability distribution in the form of detailed balance that we wish to adhere to. Let us consider that the ideal transition probabilities between the two states X_1 and X_2 to maintain the stationarity of the distribution is given by $P(X_1/X_2)$ and $P(X_2/X_1)$. Hence, as per detailed balance, the following condition must be satisfied:

$$\nabla_\theta\left(\rho^+\right)=\sum_{t=1}^{m}\frac{\sum_h e^{-E\left(v^{(t)},h\right)}\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)}{\sum_h e^{-E\left(v^{(t)},h\right)}}$$

However, discovering such an ideal transition probability function that ensures stationarity by imposing a detailed balance condition is hard. We start off with a suitable transition probability function, let's say *T(x/y)*, where *y* denotes the current state and *x* denotes the next state to be sampled based on *y*. For the two states X_1 and X_2 the assumed transition probabilities are thus given by $T(X_1/X_2)$ for a move from state X_2 to X_1 and by $T(X_2/X_1)$ for a move from state X_1 to X_2. Since the assumed transition probabilities are different than the ideal transition probabilities required to maintain stationarity through detailed balance, we get the opportunity to accept or reject samples based on how good the next move is. To cover up this opportunity, an acceptance probability for the transition of states is considered such that for a transition of a state from X_1 to X_2

$$\nabla_\theta\left(\rho^+\right)=\sum_{t=1}^{m}\frac{\sum_h \frac{e^{-E\left(v^{(t)},h\right)}}{Z}\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)}{\frac{\sum_h e^{-E\left(v^{(t)},h\right)}}{Z}}$$

where $A(X_2/X_1)$ is the acceptance probability of the move from X_1 to X_2.

As per detailed balance,

$$\begin{aligned}\nabla_\theta\left(\rho^+\right)&=\sum_{t=1}^{m}\frac{\sum_h P\left(v^{(t)},h/\theta\right)\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)}{P\left(v^{(t)}/\theta\right)}\\&=\sum_{t=1}^{m}\sum_h\frac{P\left(v^{(t)},h/\theta\right)}{P\left(v^{(t)}/\theta\right)}\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\\&=\sum_{t=1}^{m}\sum_h P\left(h/v^{(t)},\theta\right)\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\end{aligned}$$

Replacing the ideal transition probability as the product of the assumed transition probability and the acceptance probability we get

$$E\left[f(x)\right]=\sum_x P(x)f(x)=\begin{bmatrix}\sum_{x_1}\sum_{x_2}..\sum_{x_n}P(x_1,x_2,..,x_n)f_1(x_1,x_2,..,x_n)\\\sum_{x_1}\sum_{x_2}..\sum_{x_n}P(x_1,x_2,..,x_n)f_2(x_1,x_2,..,x_n)\end{bmatrix}$$

Rearranging this, we get the acceptance probability ratio as

$$\nabla_\theta\left(\rho^+\right)=\sum_{t=1}^{m}E_{p\left(h/v^{(t)},\theta\right)}\left[\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\right]$$

One simple proposal that satisfies this is given by the Metropolis algorithm as

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

$$\nabla_\theta\left(\rho^-\right)=m\frac{\sum_v\sum_h e^{-E(v,h)}\nabla_\theta\left(-E(v,h)\right)}{\sum_v\sum_h e^{-E(v,h)}}$$
$$=m\frac{\sum_v\sum_h e^{-E(v,h)}\nabla_\theta\left(-E(v,h)\right)}{Z}$$
$$=m\sum_v\sum_h\frac{e^{-E(v,h)}}{Z}\nabla_\theta\left(-E(v,h)\right)$$
$$=m\sum_v\sum_h P(v,h/\theta)\nabla_\theta\left(-E(v,h)\right)$$
$$=mE_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]$$

In the Metropolis algorithm, the assumed transitional probability is generally assumed to be a normal distribution that is symmetric, and hence $T(X_1/X_2) = T(X_2/X_1)$. This simplifies the acceptance probability of the move from X_1 to X_2 as

$$\nabla_\theta(C)=\sum_{t=1}^{m}E_{P\left(h/v^{(t)},\theta\right)}\left[\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\right]-mE_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]$$

If the acceptance probability is 1, then we accept the move with probability 1, while if the acceptance probability is less than 1, let's say r, then we accept the new sample with probability r and reject the sample with probability $(1 – r)$. This rejection of samples with the probability (1 – r) is achieved by comparing the ratio with the randomly generated sample r_u from a uniform distribution between 0 and 1 and rejecting the sample in cases where $r_u > r$. This is because for a uniform distribution probability $P(r_u > r) = 1 – r$, which ensures the desired rejection probability is maintained.

In Listing 5-2 we illustrate the sampling from a bivariate Gaussian distribution through the Metropolis algorithm.

Listing 5-2: Bivariate Gaussian Distribution Through Metropolis Algorithm

```
import numpy as np
import matplotlib.pyplot as plt
#Now let's generate this with one of the Markov Chain Monte Carlo methods called Metropolis
Hastings algorithm
# Our assumed transition probabilities would follow normal distribution X2 ~
N(X1,Covariance= [[0.2 , 0],[0,0.2]])

import time
start_time = time.time()

# Set up constants and initial variable conditions
num_samples=100000
prob_density = 0
## Plan is to sample from a Bivariate Gaussian Distribution with mean (0,0) and covariance of
## 0.7 between the two variables
mean = np.array([0,0])
cov = np.array([[1,0.7],[0.7,1]])
cov1 = np.matrix(cov)
mean1 = np.matrix(mean)
x_list,y_list = [],[]
accepted_samples_count = 0
## Normalizer of the Probability distibution
## This is not actually required since we are taking ratio of probabilities for inference
normalizer = np.sqrt( ((2*np.pi)**2)*np.linalg.det(cov))
## Start wtih initial Point (0,0)
x_initial, y_initial = 0,0
x1,y1 = x_initial, y_initial

for i in xrange(num_samples):
    ## Set up the Conditional Probability distribution, taking the existing point
    ## as the mean and a small variance = 0.2 so that points near the existing point
    ## have a high chance of getting sampled.
    mean_trans = np.array([x1,y1])
    cov_trans = np.array([[0.2,0],[0,0.2]])
    x2,y2 = np.random.multivariate_normal(mean_trans,cov_trans).T
    X = np.array([x2,y2])
    X2 = np.matrix(X)
    X1 = np.matrix(mean_trans)
    ## Compute the probability density of the existing point and the new sampled
    ## point
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
    mahalnobis_dist2 = (X2 - mean1)*np.linalg.inv(cov)*(X2 - mean1).T
    prob_density2 = (1/float(normalizer))*np.exp(-0.5*mahalnobis_dist2)
    mahalnobis_dist1 = (X1 - mean1)*np.linalg.inv(cov)*(X1 - mean1).T
    prob_density1 = (1/float(normalizer))*np.exp(-0.5*mahalnobis_dist1)
    ## This is the heart of the algorithm. Comparing the ratio of probability density
        of the new
    ## point and the existing point(acceptance_ratio) and selecting the new point if it is
        to have more probability
    ## density. If it has less probability it is randomly selected with the probability
        of getting
    ## selected being proportional to the ratio of the acceptance ratio
    acceptance_ratio = prob_density2[0,0] / float(prob_density1[0,0])

    if (acceptance_ratio >= 1) | ((acceptance_ratio < 1) and (acceptance_ratio >= np.random.
    uniform(0,1)) ):
        x_list.append(x2)
        y_list.append(y2)
        x1 = x2
        y1 = y2
        accepted_samples_count += 1

end_time = time.time()

print ('Time taken to sample ' + str(accepted_samples_count) + ' points ==> ' + str(end_time -
start_time) + ' seconds' )
print 'Acceptance ratio ===> ' , accepted_samples_count/float(100000)
## Time to display the samples generated
plt.xlabel('X')
plt.ylabel('Y')
plt.scatter(x_list,y_list,color='black')
print "Mean of the Sampled Points"
print np.mean(x_list),np.mean(y_list)
print "Covariance matrix of the Sampled Points"
print np.cov(x_list,y_list)

-Output-

Time taken to sample 71538 points ==> 30.3350000381 seconds
Acceptance ratio ===>  0.71538
Mean of the Sampled Points
-0.0090486292629 -0.008610932357
Covariance matrix of the Sampled Points
[[ 0.96043199  0.66961286]
 [ 0.66961286  0.94298698]]
```

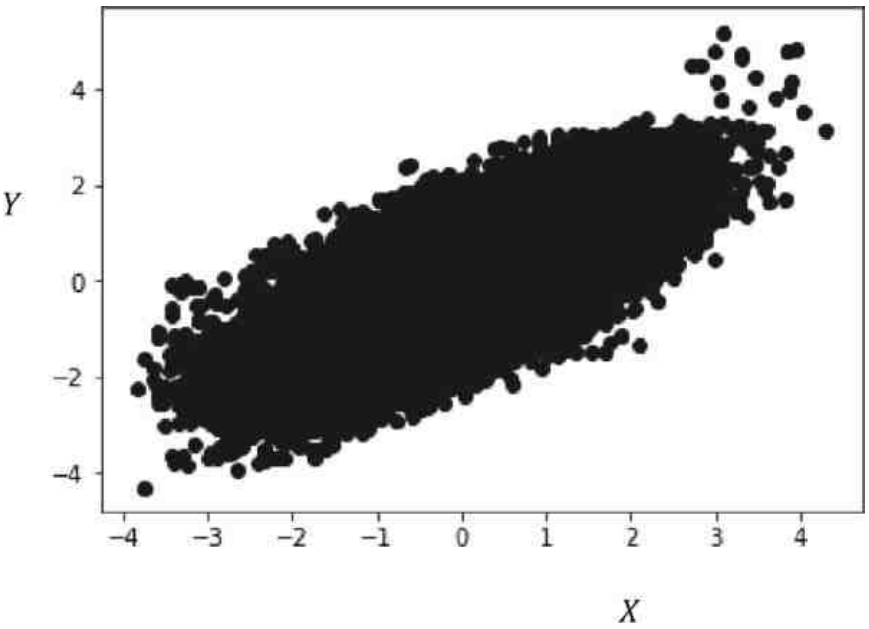


Figure 5-4: Plot of sampled points from multi-variate Gaussian distribution using Metropolis algorithm

We see from the output that the mean and the covariance of the sampled points closely represent the mean and covariance of the bivariate Gaussian distribution from which we are sampling. Also, the scatter plot in Figure 5-4 closely resembles the bivariate Gaussian distribution.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Now that we are aware of the Markov Chain Monte Carlo methods of sampling from probability distribution, we will learn about another MCMC method called Gibbs sampling while examining restricted Boltzmann machines.

Restricted Boltzmann Machines

Restricted Boltzmann machines (RBMs) belong to the unsupervised class of machine-learning algorithms that utilize the Boltzmann Equation of Probability Distribution. Illustrated in Figure 5-5 is a two-layer restricted Boltzmann machine architecture that has a hidden layer and a visible layer. There are weight connections between all the hidden and visible layers' units. However, there are no hidden-to-hidden or visible-to-visible unit connections. The term *restricted* in RBM refers to this constraint on the network. The hidden units of an RBM are conditionally independent from one another given the set of visible units. Similarly, the visible units of an RBM are conditionally independent from one another given the set of hidden units. Restricted Boltzmann machines are most often used as a building block for deep networks rather than as an individual network itself. In terms of probabilistic graphic models, restricted Boltzmann machines can be defined as undirected probabilistic graphic models containing a visible layer and a single hidden layer. Much like PCA, RBMs can be thought of as a way of representing data in one dimension (given by the visible layer *v*) into a different dimension (given by the hidden or latent layer *h*). When the size of the hidden layer is less than the size of the visible layer then RBMs perform a dimensionality reduction of data. RBMs are generally trained on binary data.

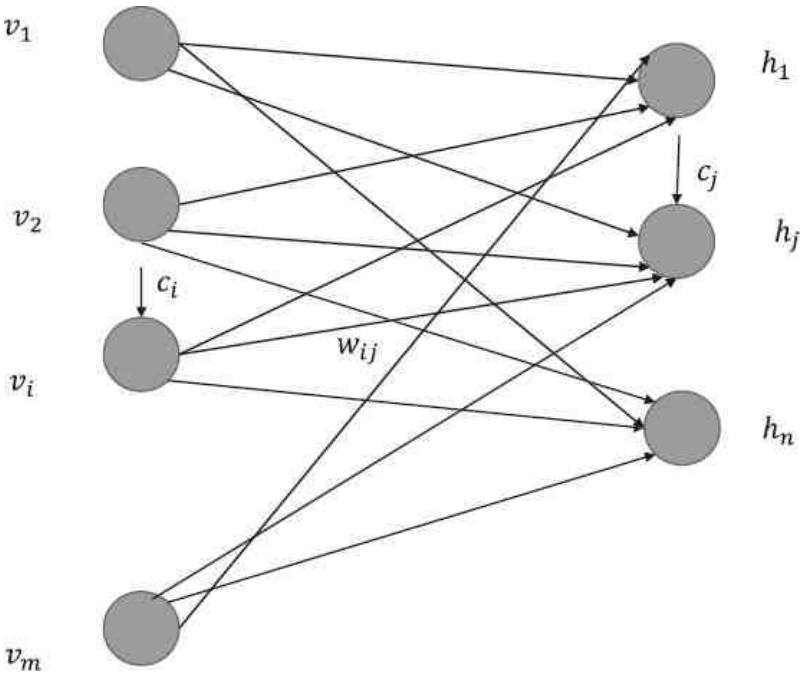


Figure 5-5: Restricted Boltzmann machine visible and hidden layers architecture

Let the visible units of the RBM be represented by vector $v = [v_1 v_2 ... v_m]^T \in \mathbb{R}^{m \times l}$ and the hidden units be represented by $h = [h_1 h_2 ... h_n]^T \in \mathbb{R}^{m \times l}$. Also, let the weight connecting the *ith* visible unit to the *jth* hidden unit be represented by w_{ij} $\forall i \in \{1,2,..m\}, \forall j \in \{1,2,..n\}$. Let the matrix containing the weights w_{ij} be represented by $W \in \mathbb{R}^{m \times n}$.

The energy of the joint probability distribution's having hidden state *h* and visible state *v* is given by

(5.3.1) $$P(v,h) = \frac{e^{-E(h,v)}}{Z}$$

where $E(v, h)$ is the energy of the joint configuration (v, h) and Z is the normalizing factor, commonly known as the partition function. This probability is based on the Boltzmann distribution and assumes the Boltzmann Constant and thermal temperature as 1.

(5.3.2) $$Z = \sum_v \sum_h e^{-E(v,h)}$$

The energy $E(v, h)$ of the joint configuration (v, h) is given by

(5.3.3) $$E(v,h) = -b^T v - c^T h - v^T W h$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

(5.3.4) $$E(v,h)=-\sum_{i=1}^{m}b_i v_i-\sum_{j=1}^{n}c_j h_j-\sum_{j=1}^{n}\sum_{i=1}^{m}v_i w_{ij}h_j$$

The vectors $b \in [b_1 b_2 \ldots b_m]^T \in \mathbb{R}^{m\times 1}$ and c ($[c_1 c_2 \ldots c_n] \in \mathbb{R}^{n\times 1}$ are biases at the visible and hidden units respectively, as we will see later.

In any graphical probabilistic model, the idea is to compute the joint probability distribution over various sets of events. Combining (5.3.1) and (5.3.3), we get

(5.3.5) $$P(v,h)=\frac{e^{b^T v+c^T h+v^T Wh}}{Z}$$

The partition function Z is hard to compute, which makes the computation of $P(v, h)$ hard to compute. For a small set of events, it is possible to compute the partition function. If there are many variables in v and h, the number of possible joint events will be exceedingly large; considering all such combinations becomes hard.

However, the conditional probability distribution $P(h/v)$ is easy to compute and sample from. The following deduction will justify this:

$$\nabla_b\left(-E(v,h)\right)=\nabla_b\left(b^T v+c^T h+v^T Wh\right)=v$$

We can expand the numerator and denominator in terms of the components of the different vectors involved, as follows:

$$\nabla_c\left(-E(v,h)\right)=\nabla_c\left(b^T v+c^T h+v^T Wh\right)=h$$

Since the exponential of a sum is equal to the product of the exponentials, the preceding equation can be written in product form as follows:

(5.3.6) $$e^{c^T h}e^{v^T Wh}=\prod_{j=1}^{n}e^{c_j h_j+v^T W[:,j]h_j}$$

Now, let's look at the denominator, which looks similar to the numerator only with a sum of all possible hidden states h. Using the expression for $e^{c^T h}e^{v^T Wh}$ in (5.3.6), the denominator can be expressed as

(5.3.7) $$\sum_h e^{c^T h}e^{v^T Wh}=\sum_h\prod_{j=1}^{n}e^{c_j h_j+v^T W[:,j]h_j}$$

The sum over vector means the sum over all combinations of its components. Each hidden unit h_i can have a binary state of 0 or 1, and hence $h_j \in \{0,1\}\ \forall j \in (1,2,3,..n\}$. So, the summation over vector h in (5.3.7) can be expanded into multiple summations corresponding to each of its components:

(5.3.8) $$\begin{aligned}\sum_h e^{c^T h}e^{v^T Wh}&=\sum_{h_1=0}^{1}\sum_{h_2=0}^{1}\ldots\sum_{h_n=0}^{1}\prod_{j=1}^{n}e^{c_j h_j+v^T W[:,j]h_j}\\&=\sum_{h_1=0}^{1}\sum_{h_2=0}^{1}\ldots\sum_{h_n=0}^{1}\left(e^{c_1 h_1+v^T W[:,1]h_1}\right)\left(e^{c_2 h_2+v^T W[:,2]h_2}\right)\ldots\left(e^{c_n h_n+v^T W[:,n]h_n}\right)\end{aligned}$$

Now, let's look at a very simple manipulation involving products and sums with two discrete variables a and b:

$$\nabla_W\left(-E(v,h)\right)=\nabla_W\left(b^T v+c^T h+v^T Wh\right)=vh^T$$

So, we see that when we take elements of variables with independent indices the sum of the products of the variables can be expressed as the product of the sum of the variables. Similar to this example, the elements of h (i.e., h_i) in general are involved in the product $\left(e^{c_1 h_1+v^T W[:,1]h_1}\right)\left(e^{c_2 h_2+v^T W[:,2]h_2}\right)\ldots\left(e^{c_n h_n+v^T W[:,n]h_n}\right)$ independently, and hence the expression in (5.3.8) can be simplified as follows:

$$\nabla_b(C)=\sum_{i=1}^{m}E_{p\left(h/v^{(i)},\theta\right)}\left[v^{(i)}\right]-mE_{p(h,v/\theta)}[v]$$

(5.3.9) $$\sum_h e^{c^T h}e^{v^T Wh}=\prod_{j=1}^{n}\sum_{h_j=0}^{1}\left(e^{c_j h_j+v^T W[:,j]h_j}\right)$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Combining the expressions for numerator and denominator from (5.3.6) and (5.3.9), we have

$$\nabla_b(C)=\sum_{t=1}^{m}v^{(t)}-mE_{P(h,v/\theta)}[v]$$

Simplifying this in terms of components of *h* on both sides, we get

(5.3.10)
$$P(h_1h_2..h_n/v)=\prod_{j=1}^{n}\left(\frac{e^{c_jh_j+v^TW[:,j]h_j}}{\sum_{h_j=0}^{1}e^{c_jh_j+v^TW[:,j]h_j}}\right)$$

$$=\left(\frac{e^{c_1h_1+v^TW[:,1]h_1}}{\sum_{h_1=0}^{1}e^{c_1h_1+v^TW[:,1]h_1}}\right)\left(\frac{e^{c_2h_2+v^TW[:,1]h_2}}{\sum_{h_2=0}^{1}e^{c_2h_2+v^TW[:,2]h_2}}\right)\left(\frac{e^{c_nh_n+v^TW[:,n]h_n}}{\sum_{h_n=0}^{1}e^{c_nh_n+v^TW[:,n]h_n}}\right)$$

The joint probability distribution of elements of *h* conditioned on *v* has factored into the product of expressions independent of each other conditioned on *v*. This leads to the fact that the components of *h* (i.e., $h_i\ \forall\ i\in\{1,2,...n\}$)are conditionally independent of each other given *v*. This gives us

(5.3.11) $$P(h_1h_2..h_n/v)=P(h_1/v)P(h_2/v)..P(h_n/v)$$

(5.3.12) $$P(h_j/v)=\frac{e^{c_jh_j+v^TW[:,j]h_j}}{\sum_{h_j=0}^{1}e^{c_jh_j+v^TW[:,j]h_j}}$$

Replacing $h_j = 1$ and $h_j = 0$ in (5.3.12), we get

(5.3.13) $$P(h_j=1/v)=\frac{e^{c_j+v^TW[:,j]}}{1+e^{c_j+v^TW[:,j]}}$$

(5.3.14) $$P(h_j=0/v)=\frac{1}{1+e^{c_j+v^TW[:,j]}}$$

The expressions for (5.3.13) and (5.3.14) illustrate the fact that the hidden units $h_i\ \forall_i \in \{1,2,\ldots n\}$ are independent sigmoid units:

(5.3.15) $$P(h_j=1/v)=\sigma\left(c_j+v^TW[:,j]\right)$$

Expanding the components of *v* and *W*[:,*j*], we can rewrite (5.3.15) as

(5.3.16) $$P(h_j=1/v)=\sigma\left(c_j+\sum_{i=1}^{m}v_iw_{ij}\right)$$

where σ(.) represents the sigmoid function such that

$$\nabla_c(C)=\sum_{t=1}^{m}\hat{h}^{(t)}-mE_{P(h,v/\theta)}[h]$$

Proceeding in a similar fashion, it can be proved that

$$\nabla_W(C)=\sum_{t=1}^{m}E_{p(h/v^{(t)},\theta)}\left[v^{(t)}h^T\right]-mE_{P(h,v/q)}[h]$$

$$=\sum_{t=1}^{m}v^{(t)}\hat{h}^{(t)T}-mE_{P(h,v/\theta)}[h]$$

which means the hidden units are conditionally independent of each other given the visible states. Since RBM is a symmetrical

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

undirected network, like the visible units, the probability of the visible units given the hidden states can be similarly expressed as

(5.3.17) $$P(v_i = 1/h) = \sigma\left(b_i + \sum_{j=1}^{n} h_j w_{ij}\right)$$

From (5.3.16) and (5.3.17) we can clearly see that the visible and hidden units are actually binary sigmoid units with the vectors *b* and *c* providing the biases at the visible and hidden units respectively. This symmetrical and independent conditional dependence of the hidden and visible units can be useful while training the model.

Training a Restricted Boltzmann Machine

We need to train the Boltzmann machine in order to derive the model parameters *b*, *c*, *W*, where *b* and *c* are the bias vectors at the visible and hidden units respectively and *W* is the weight-connections matrix between the visible and hidden layers. For ease of reference, the model parameters can be collectively referred to as

$$\begin{cases} \nabla_b(C) = \sum_{t=1}^{m} v^{(t)} - mE_{P(h,v/\theta)}[v] \\ \nabla_c(C) = \sum_{t=1}^{m} \hat{h}^{(t)} - mE_{P(h,v/\theta)}[h] \\ \nabla_W(C) = \sum_{t=1}^{m} v^{(t)}\hat{h}^{(t)^T} - mE_{P(h,v/\theta)}[h] \end{cases}$$

The model can be trained by maximizing the log likelihood function of the input data points with respect to the model parameters. The input is nothing but the data corresponding to the visible units for each data point. The likelihood function is given by

$$P\left(v_i^{(k)} = 1/h\right) = \frac{e^{\left(b_i^{(k)} + \sum_{j=1}^{n} h_j w_{ij}^{(k)}\right)}}{\sum_{l=1}^{K} e^{\left(b_i^{(l)} + \sum_{j=1}^{n} h_j w_{ij}^{(l)}\right)}}$$

Since the input's data points are independent given the model,

(5.3.18) $$L(\theta) = P\left(v^{(1)}/\theta\right)P(v^{(2)}/\theta)..P\left(v^{(m)}/\theta\right) = \prod_{t=1}^{m} P\left(v^{(t)}/\theta\right)$$

Taking the log on both sides to get the log likelihood expression of the function in (5.3.18), we have

(5.3.19) $$C = logL(\theta) = \sum_{t=1}^{m} logP\left(v^{(t)}/\theta\right)$$

Expanding the probabilities in (5.3.19) by its joint probability form, we get

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

(5.3.20)
$$\begin{aligned} C &= \sum_{t=1}^{m} \log P\left(v^{(t)} / \theta\right) \\ &= \sum_{t=1}^{m} \log \sum_{h} P\left(v^{(t)}, h / \theta\right) \\ &= \sum_{t=1}^{m} \log \sum_{h} \frac{e^{-E\left(v^{(t)}, h\right)}}{Z} \\ &= \sum_{t=1}^{m} \log \frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}{Z} \\ &= \sum_{t=1}^{m} \log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - \sum_{t=1}^{m} \log Z \\ &= \sum_{t=1}^{m} \log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - m \log Z \end{aligned}$$

The partition function Z is not constrained by visible-layer inputs $v^{(t)}$ unlike the first term in (5.3.20). Z is the sum of the negative exponentials of the energies over all possible combinations of v and h and so can be expressed as

$$Z = \sum_{v} \sum_{h} e^{-E(v,h)}$$

Replacing Z with this expression in (5.3.20), we get

(5.3.21)
$$C = \sum_{t=1}^{m} log \sum_{h} e^{-E\left(v^{(t)}, h\right)} - m log \sum_{v} \sum_{h} e^{-E(v,h)}$$

Now, let's take the gradient of the cost function with respect to the combined parameter θ We can think of C as comprising two components, ρ^{+} and ρ^{-}, where

$$E(v,h) = -\sum_{k=1}^{K} \sum_{i=1}^{m} b_i^{(k)} v_i^{(k)} - \sum_{j=1}^{n} c_j h_j - \sum_{k=1}^{K} \sum_{j=1}^{n} \sum_{i=1}^{m} v_i^{(k)} w_{ij}^{(k)} h_j$$

$$P(v,h) \propto e^{-E(v,h)} = e^{\sum_{k=1}^{K} \sum_{i=1}^{m} b_i^{(k)} v_i^{(k)} + \sum_{j=1}^{n} c_j h_j + \sum_{k=1}^{K} \sum_{j=1}^{n} \sum_{i=1}^{m} v_i^{(k)} w_{ij}^{(k)} h_j}$$

Taking the gradient of ρ^{+} with respect to θ, we have

(5.3.22)
$$\nabla_{\theta}\left(\rho^{+}\right) = \sum_{t=1}^{m} \frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)} \nabla_{\theta}\left(-E\left(v^{(t)}, h\right)\right)}{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}$$

$$\frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)} \nabla_{\theta}\left(-E\left(v^{(t)}, h\right)\right)}{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}$$

Now, let's simplify pp by dividing both the numerator and the denominator by Z:

(5.3.23)
$$\nabla_{\theta}\left(\rho^{+}\right) = \sum_{t=1}^{m} \frac{\sum_{h} \frac{e^{-E\left(v^{(t)}, h\right)}}{Z} \nabla_{\theta}\left(-E\left(v^{(t)}, h\right)\right)}{\frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}{Z}}$$

$\frac{e^{-E\left(v^{(t)}, h\right)}}{Z} = P\left(v^{(t)}, h / \theta\right)$ and $\frac{\sum_{h} e^{-E\left(v^{(t)}, h\right)}}{Z} = P\left(v^{(t)} / \theta\right)$ Using these expressions for the probabilities in

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

(5.3.24)
$$\nabla_\theta\left(\rho^+\right)=\sum_{t=1}^{m}\frac{\sum_h P\left(v^{(t)},h/\theta\right)\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)}{P\left(v^{(t)}/\theta\right)}$$
$$=\sum_{t=1}^{m}\sum_h\frac{P\left(v^{(t)},h/\theta\right)}{P\left(v^{(t)}/\theta\right)}\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)$$
$$=\sum_{t=1}^{m}\sum_h P\left(h/v^{(t)},\theta\right)\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)$$

One can remove the θ from the probability notations, such as $P(v^{(t)}, h/\theta)$, $P(v^{(t)}, h/\theta)$, and so forth, for ease of notation if one wishes to, but it is better to keep them since it makes the deductions more complete, which allows for better interpretability of the overall training process.

Let us look at the expectation of functions, which gives us the expression seen in (5.3.24) in a more meaningful form ideal for training purposes. The expectation of $f(x)$, given x, follows probability mass function $P(x)$ nd is given by

$$P\left(hj=1/v\right)=\frac{e^{\left(c_j+\sum_{i=1}^{m}\sum_{k=1}^{K}v_i^{(k)}w_{ij}^{(k)}\right)}}{1+e^{\left(c_j+\sum_{i=1}^{m}\sum_{k=1}^{K}v_i^{(k)}w_{ij}^{(k)}\right)}}$$

If $x = [x_1 x_2 \ldots x_n]^T \in \mathbb{R}^{n\times 1}$ is multi-variate, then the preceding expression holds true and

$$P\left(v_q^{(k)}/V\right)=\sum_h P\left(v_q^{(k)},h/V\right)=\frac{\sum_h P\left(v_q^{(k)},h,V\right)}{P(V)}$$

Similarly, if $f(x)$ is a vector of functions such that $f(x) = [\, f_1(x)\ f_2(x)]^T$, one can use the same expression as for expectation. Here, one would get a vector of expectations, as follows:

(5.3.25)
$$E\left[f(x)\right]=\sum_x P(x)f(x)=\begin{bmatrix}\sum_{x_1}\sum_{x_2}\ldots\sum_{x_n}P\left(x_1,x_2,\ldots,x_n\right)f_1\left(x_1,x_2,\ldots,x_n\right)\\ \sum_{x_1}\sum_{x_2}\ldots\sum_{x_n}P\left(x_1,x_2,\ldots,x_n\right)f_2\left(x_1,x_2,\ldots,x_n\right)\end{bmatrix}$$

To explicitly mention the probability distribution in the expectation notation, one can rewrite the expectation of functions or expectation of vector of functions whose variables x follow probability distribution $P(x)$ as follows:

$$P\left(v_q^{(k)}/V\right)\propto\sum_h P\left(v_q^{(k)},h,V\right)$$
$$\propto\sum_h e^{-E\left(v_q^{(k)},h,V\right)}$$

Since we are working with gradients, which are vectors of different partial derivatives, and each of the partial derivatives is a function of h for given values of θ and v, the expression in (5.3.24) can be expressed in terms of expectation of the gradient $\nabla_\theta(-E(v^{(t)},h))$ with respect to the probability distribution $P(h/v^{(t)}, \theta)$ as

(5.3.26)
$$\nabla_\theta\left(\rho^+\right)=\sum_{t=1}^{m}E_{p\left(h/v^{(t)},\theta\right)}\left[\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\right]$$

Note that the expectation $E_{p\left(h/v^{(t)},\theta\right)}\left[\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\right]$ is a vector of expectations, as has been illustrated in (5.3.25).

Now, let's get to the gradient of $\rho^- = m\log\sum_v\sum_h e^{-E(v,h)}$ with respect to the θ :

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

(5.3.27)
$$\begin{aligned}\nabla_\theta\left(\rho^-\right) &= m\frac{\sum_v\sum_h e^{-E(v,h)}\nabla_\theta\left(-E(v,h)\right)}{\sum_v\sum_h e^{-E(v,h)}}\\ &= m\frac{\sum_v\sum_h e^{-E(v,h)}\nabla_\theta\left(-E(v,h)\right)}{Z}\\ &= m\sum_v\sum_h\frac{e^{-E(v,h)}}{Z}\nabla_\theta\left(-E(v,h)\right)\\ &= m\sum_v\sum_h P(v,h/\theta)\nabla_\theta\left(-E(v,h)\right)\\ &= mE_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]\end{aligned}$$

The expectation in (5.3.27) is over the joint distribution of *h* and *v* whereas the expectation in (5.3.26) is over the *h* given a seen *v*. Combining (5.3.26) and (5.3.27), we get

(5.3.28) $$\nabla_\theta(C)=\sum_{t=1}^{m}E_{P\left(h/v^{(t)},\theta\right)}\left[\nabla_\theta\left(-E\left(v^{(t)},h\right)\right)\right]-mE_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]$$

If we look at the gradient with respect to all the parameters in (5.3.28) it has two terms. The first term is dependent on the seen data $v^{(t)}$, while the second term depends on samples from the model. The first term increases the likelihood of the given observed data while the second term reduces the likelihood of data points from the model.

Now, let's do some simplification of the gradient for each of the parameter sets in θ; i.e. *b, c*, and *W*.

(5.3.29) $$\nabla_b\left(-E(v,h)\right)=\nabla_b\left(b^Tv+c^Th+v^TWh\right)=v$$
(5.3.30) $$\nabla_c\left(-E(v,h)\right)=\nabla_c\left(b^Tv+c^Th+v^TWh\right)=h$$
(5.3.31) $$\nabla_W\left(-E(v,h)\right)=\nabla_W\left(b^Tv+c^Th+v^TWh\right)=vh^T$$

Using (5.3.28) through (5.3.31), the expression for the gradient with respect to each of the parameter sets is given by

(5.3.32) $$\nabla_b(C)=\sum_{t=1}^{m}E_{P\left(h/v^{(t)},\theta\right)}\left[v^{(t)}\right]-mE_{P(h,v/\theta)}[v]$$

Since the probability distribution of the first term is conditioned on $v^{(t)}$, the expectation of $v^{(t)}$ with respect to $P(h/v^{(t)}, \theta)$ is $v^{(t)}$.

(5.3.33) $$\nabla_b(C)=\sum_{t=1}^{m}v^{(t)}-mE_{P(h,v/\theta)}[v]$$
(5.3.34) $$\nabla_c(C)=\sum_{t=1}^{m}E_{P\left(h/v^{(t)},\theta\right)}[h]-mE_{P(h,v/\theta)}[h]$$

The expectation of *h* over the probability distribution $P(h/v^{(t)}, \theta)$ can be easily computed since each of the units of *h* (i.e., h_j) given $v^{(t)}$ is independent. Each of them is a sigmoid unit with two possible outcomes, and their expectation is nothing but the output of the sigmoid units; i.e.,

$$E\left(v_q^{(k)},V,h\right)=-\sum_{k=1}^{K}\sum_{i=1}^{m}b_i^{(k)}v_i^{(k)}-\sum_{j=1}^{n}c_jh_j-\sum_{k=1}^{K}\sum_{j=1}^{n}\sum_{i=1}^{m}v_i^{(k)}w_{ij}^{(k)}h_j-\sum_{j=1}^{n}v_s^{(k)}w_{sj}^{(k)}h_j-v_s^{(k)}b^{(k)}$$

If we replace the expectation with $\hat{h}$ then the expression in (5.3.34) can be written as

(5.3.35) $$\nabla_c(C)=\sum_{t=1}^{m}\hat{h}^{(t)}-mE_{P(h,v/\theta)}[h]$$

Similarly,

(5.3.36) $$\begin{aligned}\nabla_W(C) &= \sum_{t=1}^{m}E_{P\left(h/v^{(t)},\theta\right)}\left[v^{(t)}h^T\right]-mE_{P(h,v/q)}[h]\\ &= \sum_{t=1}^{m}v^{(t)}\hat{h}^{(t)T}-mE_{P(h,v/\theta)}[h]\end{aligned}$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

So, the expressions in (5.3.33), (5.3.35), and (5.3.36) represent the gradients with respect to the three parameter sets. For easy reference:

(5.3.37)
$$\begin{cases} \nabla_b(C) = \sum_{t=1}^{m} v^{(t)} - mE_{P(h,v/\theta)}[v] \\ \nabla_c(C) = \sum_{t=1}^{m} \hat{h}^{(t)} - mE_{P(h,v/\theta)}[h] \\ \nabla_W(C) = \sum_{t=1}^{m} v^{(t)}\hat{h}^{(t)^T} - mE_{p(h,v/\theta)}[h] \end{cases}$$

Based on these gradients, gradient-descent techniques can be invoked to iteratively get the parameter values that maximize the likelihood function. However, there is a little complexity involved to compute the expectations with respect to the joint probability distribution $P(h, v/\theta)$ at each iteration of gradient descent. The joint distribution is hard to compute because of the seemingly large number of combinations for h and v in cases where they are moderate to large dimensionality vectors. Markov Chain Monte Carlo sampling (MCMC) techniques, especially Gibbs Sampling, can be used to sample from the joint distribution and compute the expectations in (5.3.37) for the different parameter sets. However, MCMC techniques take a long time to converge to a stationary distribution, after which they provide good samples. Hence, to invoke MCMC sampling at each iteration of gradient descent would make the learning very slow and impractical.

Gibbs Sampling

Gibbs sampling is a Markov Chain Monte Carlo method that can be used to sample observations from a multi-variate probability distribution. Suppose we want to sample from a multi-variate joint probability distribution $P(x)$ where $x = [x_1 x_2 .. x_n]^T$.

Gibbs sampling generates the next value of a variable x_i conditioned on all the current values of the other variables. Let the t-th sample drawn be represented by $x^{(t)} = \left[x_1^{(t)} x_2^{(t)} .. x_n^{(t)} \right]^T$. To generate the ($t$+1) sample seen next, follow this logic:

- Draw the variable $x_j^{(t+1)}$ by sampling it from a probability distribution conditioned on the rest of the variables. In other words, draw $x_j^{(t+1)}$ from

  $$P\left(x_j^{(t+1)} / x_1^{(t+1)} x_1^{(t+1)} .. x_{j-1}^{(t+1)} x_{j+1}^{(t)} .. x_n^{(t)} \right)$$

  So basically, for sampling x_j conditioned on the rest of the variables, for the j – 1 variables before x_j their values for the (t+1) instance are considered since they have already been sampled, while for the rest of the variables their values at instance t are considered since they are yet to be sampled. This step is repeated for all the variables.

  If each x_j is discrete and can take, let's say, two values 0 and 1, then we need to compute the probability $p_1 = P\left(x_j^{(t+1)} = 1 / x_1^{(t+1)} x_2^{(t+1)} .. x_{j-1}^{(t+1)} x_{j+1}^{(t)} .. x_n^{(t)} \right)$. We can then draw a sample u from a uniform probability distribution between 0 and 1 (i.e., U[0, 1]), and if $p_1 \geq u$ set $x_j^{(t+1)} = 1$, else set $x_j^{(t+1)} = 0$. This kind of random heuristics ensure that the higher the probability $p_1$is, the greater the chances are of $x_j^{(t+1)}$ getting selected as 1. However, it still leaves room for 0 getting selected with very low probability, even if p_1 is relatively large, thus ensuring that the Markov Chain doesn't get stuck in a local region and can explore other potential high-density regions as well. There are the same kind of heuristics that we saw for the Metropolis algorithm as well.

- If one wishes to generate m samples from the joint probability distribution $P(x)$ the preceding step has to be repeated m times.

The conditional distributions for each variable based on the joint probability distribution are to be determined before the sampling can proceed. If one is working on Bayesian networks or restricted Boltzmann machines, there are certain constraints within the variables that help determine these conditional distributions in an efficient manner.

As an example, if one needs to do Gibbs sampling from a bivariate normal distribution with mean [0 0] and covariance matrix

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

$\begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$, then the conditional probability distributions can be computed as follows:

$$P(x_2 / x_1) = P(x_1, x_2) / P(x_1)$$

$$P(x_1 / x_2) = P(x_1, x_2) / P(x_2)$$

If one derives the marginal distributions $P(x_1)$ and $P(x_2)$ as $\int_{x_2} P(x_1, x_2) dx_2$ and $\int_{x_1} P(x_1, x_2) dx_1$, then

$$x_2 / x_1 \sim Normal\left(\rho x_1, 1 - p^2\right)$$

$$x_1 / x_2 \sim Normal\left(\rho x_2, 1 - p^2\right)$$

Block Gibbs Sampling

There are several variants of Gibbs sampling. Block Gibbs sampling is one of them. In Block Gibbs sampling, more than one variable is grouped together, and then the group of variables is sampled together conditioned on the rest of the variables, as opposed to sampling for individual variables separately. For example, in a restricted Boltzmann machine the hidden unit state variables $h = [h_1\ h_1\ ..h_n]^T \in \mathbb{R}^{n \times 1}$ can be sampled together conditioned on the visible unit states $v = [\ v_1\ v_2..v_m]^T \in \mathbb{R}^{m \times 1}$ and vice versa. Hence, for sampling from the joint probability distribution over $P(v, h)$, through Block Gibbs sampling one can sample all the hidden states given the visible unit states through the conditional distribution $P(h/v)$ and sample all the visible unit states given the hidden unit states through the conditional distribution $P(v/h)$. The samples at the (t+1) iteration of Gibbs sampling can be generated as

$$h^{(t+1)} \sim P\left(h / v^{(t)}\right)$$

$$v^{(t+1)} \sim P\left(v / h^{(t+1)}\right)$$

Therefore, $\left(v^{(t+1)}, h^{(t+1)}\right)$ is the combined sample at iteration (t+1).

If we must compute the expectation of a function $f(h, v)$, it can be computed as follows:

$$E\left[f(h,v)\right] \approx \frac{1}{M} \sum_{t=1}^{M} f\left(h^{(t)}, v^{(t)}\right)$$

where M denotes the number of samples generated from the joint probability distribution $P(v, h)$.

Burn-in Period and Generating Samples in Gibbs Sampling

To consider the samples as independent as possible for expectation computation or otherwise based on the joint probability distribution, generally the samples are picked up at an interval of k samples. The larger the value of k the better at removing the auto-correlation among the generated samples. Also, the samples generated at the beginning of the Gibbs sampling are ignored. These ignored samples are said to have been generated in the burn-in period.

The burn-in period uses the Markov Chain to settle down to an equilibrium distribution before we can start drawing samples from it. This is required because we generate the Markov Chain from an arbitrary sample that might be a low-probability zone with respect to the actual distribution, and so we can throw away those unwanted samples. The low-probability samples don't contribute much to the actual expectation, and thus having plenty of them in the sample would obscure the expectation. Once the Markov Chain has run long enough, it will have settled to some high-probability zone, at which point we can start collecting the samples.

Using Gibbs Sampling in Restricted Boltzmann Machines

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Block Gibbs sampling can be used to compute the expectations with respect to the joint probability distribution $P(v, h/\theta)$ as mentioned in the equations in (5.3.37) for computing the gradient with respect to the model parameters *b, c*, and *W*. Here are the equations from (5.3.37) for easy reference:

$$\begin{cases} \nabla_b(C) = \sum_{t=1}^{m} v^{(t)} - mE_{P(h,v/\theta)}[v] \\ \nabla_c(C) = \sum_{t=1}^{m} \hat{h} - mE_{P(h,v/\theta)}[h] \\ \nabla_W(C) = \sum_{t=1}^{m} v^{(t)}\hat{h}^T - \mathrm{mE}_{P(h,v/\theta)}\left[vh^T\right] \end{cases}$$

The expectations $E_{P(h,v/\theta)}[v], E_{P(h,v/\theta)}[h]$, and $E_{P(h,v/\theta)}[vh^T]$ all require sampling from the joint probability distribution $P(v, h/\theta)$. Through Block Gibbs sampling, samples (v, h) can be drawn as follows based on their conditional probabilities, where t denotes the iteration number of Gibbs sampling:

$$h^{(t+1)} \sim P\left(h/v^{(t)},\theta\right)$$

$$v^{(t+1)} \sim P\left(v/h^{(t+1)},\theta\right)$$

What makes sampling even easier is the fact that the hidden units' h_j s are independent given the visible unit states, and vice versa:

$$P(h/v) = P(h_1/v)P(h_2/v)..P(h_n/v) = \prod_{j=1}^{n} P(h_j/v)$$

This allows the individual hidden units' h_j s to be sampled independently in parallel given the values of the visible unit states. The parameter θ has been removed from the preceding notation since θ would remain constant for a step of gradient descent when we perform Gibbs sampling.

Now, each of the hidden unit output states h_j can be either 0 or 1, and its probability of assuming state 1 is given by (5.3.16) as

$$P(h_j = 1/v) = \sigma\left(c_j + \sum_{i=1}^{m} v_i w_{\bar{ij}}\right)$$

This probability can be computed based on the current value of $v = v^{(t)}$ and model parameters $c, W \in \theta$. The computed probability $P\left(h_j = 1/v^{(t)}\right)$ is compared to a random sample u generated from a uniform distribution $U[0, 1]$. If $P\left(h_j = 1/v^{(t)}\right) > u$, then the sampled $h_j = 1$, else $h_j = 0$. All such h_j *s* sampled in this fashion form the combined hidden unit state vector $h^{(t+1)}$.

Similarly, the visible units are independent given the hidden unit states:

$$P(v/h) = P(v_1/h)P(v_2/h)..P(v_n/h) = \prod_{i=1}^{m} P(v_i/h)$$

Each of the visible units can be sampled independently given $h^{(t+1)}$ to get the combined $v^{(t+1)}$ in the same way as for the hidden units. The required sample thus generated in the $(t+1)$ iteration is given by $(v^{(t+1)}, h^{(t+1)})$.

All the expectations $E_{P(h,v/\theta)}[v], E_{P(h,v/\theta)}[h]$, and $E_{P(h,v/\theta)}[vh^T]$ can be computed by taking the averages of the samples generated through Gibbs sampling. Through Gibbs sampling, if we take N samples after considering burn-in periods and auto-correlation as discussed earlier, the required expectation can be computed as follows:

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

$$E_{P(h,v/\theta)}[v] \approx \frac{1}{N}\sum_{i=1}^{N} v^{(i)}$$

$$E_{P(h,v/\theta)}[h] \approx \frac{1}{N}\sum_{i=1}^{N} h^{(i)}$$

$$E_{P(h,v/\theta)}\left[vh^T\right] \approx \frac{1}{N}\sum_{i=1}^{N} v^{(i)}h^{(i)T}$$

However, doing Gibbs sampling for the joint distribution to generate *N* samples in each iteration of gradient descent becomes a tedious task and is often impractical. There is an alternate way of approximating these expectations, called *contrastive divergence*, which we will discuss in the next section.

Contrastive Divergence

Performing Gibbs sampling on the joint probability distribution $P(h, v/\theta)$ at each step of gradient descent becomes challenging since Markov Chain Monte Carlo methods such as Gibbs sampling take a long time to converge, which is required in order to produce unbiased samples. These unbiased samples drawn from the joint probability distribution are used to compute the expectations terms $E_{P(h,v/\theta)}[v]$, $E_{P(h,v/\theta)}[h]$, and $E_{P(h,v/\theta)}[vh^T]$ which are nothing but components of the term $E_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]$ in the combined expression for gradients as deduced in (5.3.28).

$$\nabla_\theta(C) = \sum_{t=1}^{m} E_{P\left(h/v^{(t)},\theta\right)}[\nabla_\theta(-E(v^{(t)},h))] - mE_{P(h,v/\theta)}\left[\nabla_\theta(-E(v,h))\right]$$

The second term in the preceding equation can be rewritten as a summation over the *m* data points, and hence

$$\nabla_\theta(C) = \sum_{t=1}^{m} E_{P\left(h/v^{(t)},\theta\right)}[\nabla_\theta(-E(v^{(t)},h))] - \sum_{t=1}^{m} E_{P(h,v/\theta)}[\nabla_\theta(-E(v,h))]$$

Contrastive divergence approximates the overall expectation $E_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right]$ by a point estimate at a candidate sample $(\bar{v},\bar{h})$ obtained by performing Gibbs sampling for only a couple of iterations.

$$E_{P(h,v/\theta)}\left[\nabla_\theta\left(-E(v,h)\right)\right] \approx \nabla_\theta\left(-E\left(\bar{v},\bar{h}\right)\right)$$

This approximation is done for every data point $v^{(t)}$, and hence the expression for the overall gradient can be rewritten as follows:

$$\nabla_\theta(C) \approx \sum_{t=1}^{m} E_{P(h/v^{(t)},\theta)}[\nabla_\theta(-E(v^{(t)},h))] - \sum_{t=1}^{m}[\nabla_\theta(-E(\bar{v}^{(t)},\bar{h}^{(t)}))]$$

Figure 5-6 below illustrates how Gibbs sampling is performed for each input data point $v^{(t)}$ to get the expectation approximation over the joint probability distribution by a point estimate. The Gibbs sampling starts with $v^{(t)}$ and, based on the conditional probability distribution $P(h/v^{(t)})$, the new hidden state h' is obtained. As discussed earlier, each of the hidden units h_j can be sampled independently and then be combined to form the hidden state vector h'. Then v' is sampled based on the conditional probability distribution $P(v/h')$. This iterative process is generally run for couple of iterations, and the final v and h sampled are taken as the candidate sample $(\bar{v},\bar{h})$.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

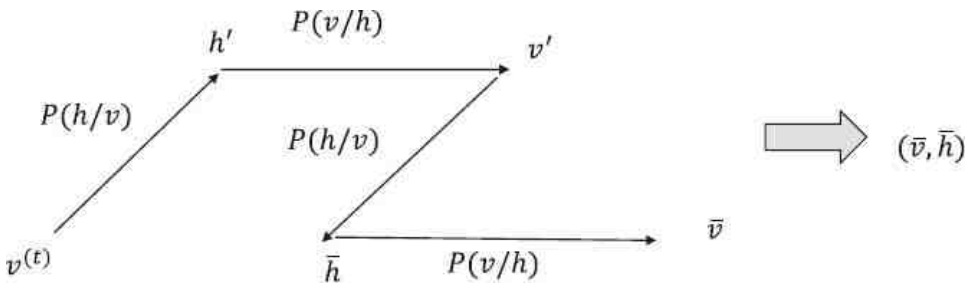


Figure 5-6: Gibbs sampling for two iterations to get one sample for Contrastive Divergence

The contrastive divergence method makes the gradient descent faster since the Gibbs sampling in each step of gradient descent is limited to only a few iterations, mostly one or two per data point.

A Restricted Boltzmann Implementation in TensorFlow

In this section, we will go through the implementation of restricted Boltzmann machines using the MNIST dataset. Here, we try to model the structure of the MNIST images by defining a restricted Boltzmann machine network that consists of the image pixels as the visible units and 500 hidden layers in order to decipher the internal structure of each image. Since the MNIST images are 28 × 28 in dimension, when flattened as a vector we have 784 visible units. We try to capture the hidden structures properly by training the Boltzmann machines. Images that represent the same digit should have similar hidden states, if not the same, when said hidden states are sampled given the visible representations of the input images. When the visible units are sampled, given their hidden structure, the visible unit values when structured in an image form should correspond to the label of the image. The detailed code is illustrated in Listing 5-3a.

Listing 5-3a: Restricted Boltzmann Machine Implementation with MNIST Dataset

```
##Import the Required libraries
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline

## Read the MNIST files
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

## Set up the parameters for training

n_visible      = 784
n_hidden    = 500
display_step = 1
num_epochs = 200
batch_size = 256
lr         = tf.constant(0.001, tf.float32)

## Define the tensorflow variables for weights and biases as well as placeholder for input
x  = tf.placeholder(tf.float32, [None, n_visible], name="x")
W  = tf.Variable(tf.random_normal([n_visible, n_hidden], 0.01), name="W")
b_h = tf.Variable(tf.zeros([1, n_hidden],  tf.float32, name="b_h"))
b_v = tf.Variable(tf.zeros([1, n_visible],  tf.float32, name="b_v"))

## Converts the probability into discrete binary states; i.e., 0 and 1
def sample(probs):
    return tf.floor(probs + tf.random_uniform(tf.shape(probs), 0, 1))

## Gibbs sampling step
def gibbs_step(x_k):
        h_k = sample(tf.sigmoid(tf.matmul(x_k, W) + b_h))
        x_k = sample(tf.sigmoid(tf.matmul(h_k, tf.transpose(W)) + b_v))
        return x_k
## Run multiple Gibbs sampling steps starting from an initial point
def gibbs_sample(k,x_k):
    for i in range(k):
        x_out = gibbs_step(x_k)
# Returns the Gibbs sample after k iterations
    return x_out

# Constrastive Divergence algorithm
# 1. Through Gibbs sampling locate a new visible state x_sample based on the current visible
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
state x
# 2. Based on the new x sample a new h as h_sample
x_s = gibbs_sample(2,x)
h_s = sample(tf.sigmoid(tf.matmul(x_s, W) + b_h))

# Sample hidden states based given visible states
h = sample(tf.sigmoid(tf.matmul(x, W) + bh))
# Sample visible states based given hidden states
x_ = sample(tf.sigmoid(tf.matmul(h, tf.transpose(W)) + b_v))

# The weight updated based on gradient descent
size_batch = tf.cast(tf.shape(x)[0], tf.float32)
W_add  = tf.multiply(lr/size_batch, tf.subtract(tf.matmul(tf.transpose(x), h), tf.matmul(tf.
transpose(x_s), h_s)))
bv_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(x, x_s), 0, True))
bh_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(h, h_s), 0, True))
updt = [W.assign_add(W_add), b_v.assign_add(bv_add), b_h.assign_add(bh_add)]

# TensorFlow graph execution

with tf.Session() as sess:
    # Initialize the variables of the Model
    init = tf.global_variables_initializer()
    sess.run(init)
    total_batch = int(mnist.train.num_examples/batch_size)
    # Start the training
    for epoch in range(num_epochs):
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Run the weight update
            batch_xs = (batch_xs > 0)*1
            _ = sess.run([updt], feed_dict={x:batch_xs})
        # Display the running step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1))

    print("RBM training Completed !")

    ## Generate hidden structure for 1st 20 images in test MNIST

    out = sess.run(h,feed_dict={x:(mnist.test.images[:20]> 0)*1})
    label = mnist.test.labels[:20]

    ## Take the hidden representation of any of the test images; i.e., the 3rd record
    ## The output level of the 3rd record should match the image generated
    plt.figure(1)
    for k in range(20):
        plt.subplot(4, 5, k+1)
        image = (mnist.test.images[k]> 0)*1
        image = np.reshape(image,(28,28))
        plt.imshow(image,cmap='gray')

    plt.figure(2)

    for k in range(20):
        plt.subplot(4, 5, k+1)
        image = sess.run(x_,feed_dict={h:np.reshape(out[k],(-1,n_hidden))})
        image = np.reshape(image,(28,28))
        plt.imshow(image,cmap='gray')
        print(np.argmax(label[k]))

    sess.close()

--Output --
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

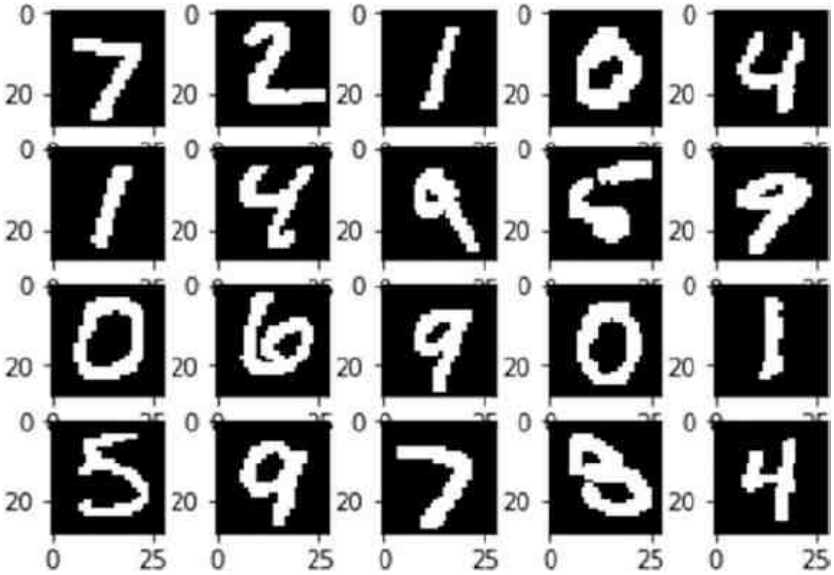


Figure 5-7: Actual test images

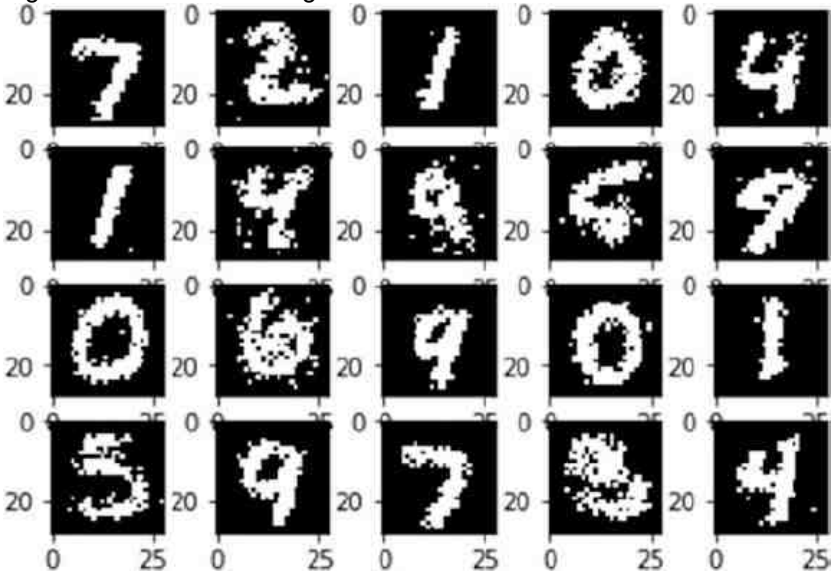


Figure 5-8: Simulated images given the hidden states

We can see from Figure 5-7 and Figure 5-8 that the RBM model did a good job of simulating the input images given their hidden representations. Hence, restricted Boltzmann machines can be used as generative models as well.

Collaborative Filtering Using Restricted Boltzmann Machines

Restricted Boltzmann machines can be used for collaborative filtering in making recommendations. Collaborative filtering is the method of predicting the preference of a user for an item by analyzing the preferences of many users for items. Given a set of items and users along with the ratings the users have provided for a variety of items, the most common method for collaborative filtering is the matrix factorization method, which determines a set of vectors for the items as well as for the users. The rating assigned by a user to an item can then be computed as the dot product of the user vector with the item vector $u^{(k)}$. Hence, the rating can be expressed as

$$r^{(jk)} = u^{(j)T} v^{(k)}$$

where *j* and *k* represent the *jth* user and the *kth* item respectively. Once the vectors for each item and each user have been learned, the expected ratings a user would assign to a product they haven't rated yet can be found out by the same method. Matrix factorization can be thought of as decomposing a big matrix of ratings into user and item vectors.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

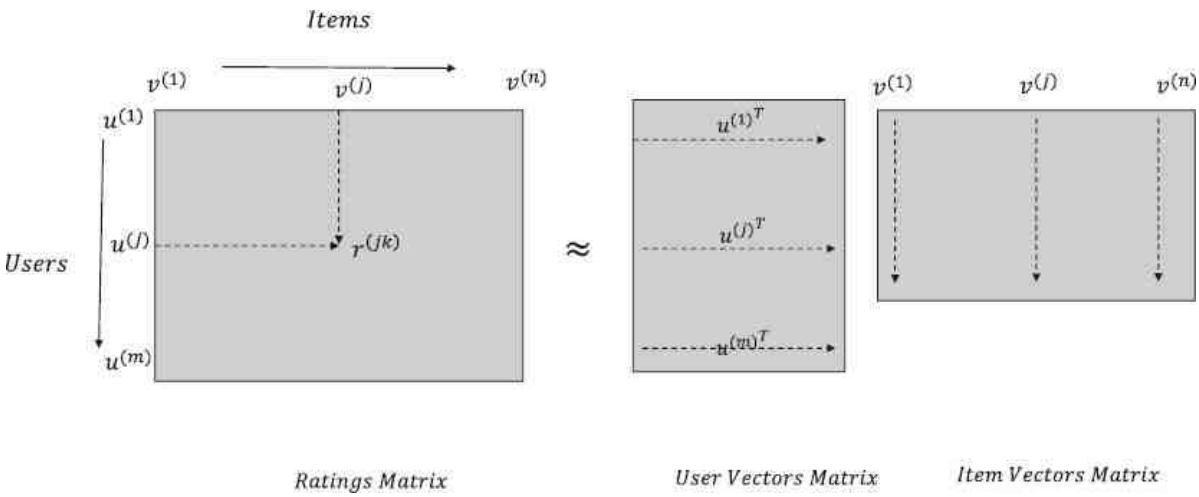


Figure 5-9: Schematic diagram of a matrix factorization method for collaborative filtering

Illustrated in Figure 5-9 is a schematic diagram of a matrix factorization method that decomposes a user item rating matrix into two matrices consisting of user vectors and item vectors. The dimensionality of the user and rating vectors must be equal for their dot product to hold true, which gives an estimate of what rating the user might assign to a particular item. There are several matrix factorization methods, such as singular value decomposition (SVD), non-negative matrix factorization, alternating least squares, and so on. Any of the suitable methods can be used for matrix factorization depending on the use. Generally, SVD requires filling in the missing ratings (where the users have not rated the items) in the matrix, which might be a difficult task, and hence methods such as alternating least squares, which only takes the provided ratings and not the missing values, works well for collaborative filtering.

Now, we will look at a different method of collaborative filtering that uses restricted Boltzmann machines. Restricted Boltzmann machines were used by the winning team in the Netflix Challenge of Collaborative Filtering, and so let's consider the items as movies for this discussion. The visible units for this RBM network would correspond to movie ratings, and instead of being binary, each movie would be a five-way SoftMax unit to account for the five possible ratings from 1 to 5. The number of hidden units can be chosen arbitrarily; we chose *d* here. There would be several missing values for different movies since all the movies would not be rated by all the users. The way to handle them is to train a separate RBM for each user based on only the movies that user has rated. The weights from the movies to the hidden units would be shared by all users. For instance, let's say *User A* and *User B* rate the same movie; they would use the same weights connecting the movie unit to the hidden units. So, all the RBMs would have the same hidden units, and of course their activation of the hidden units may be very different.

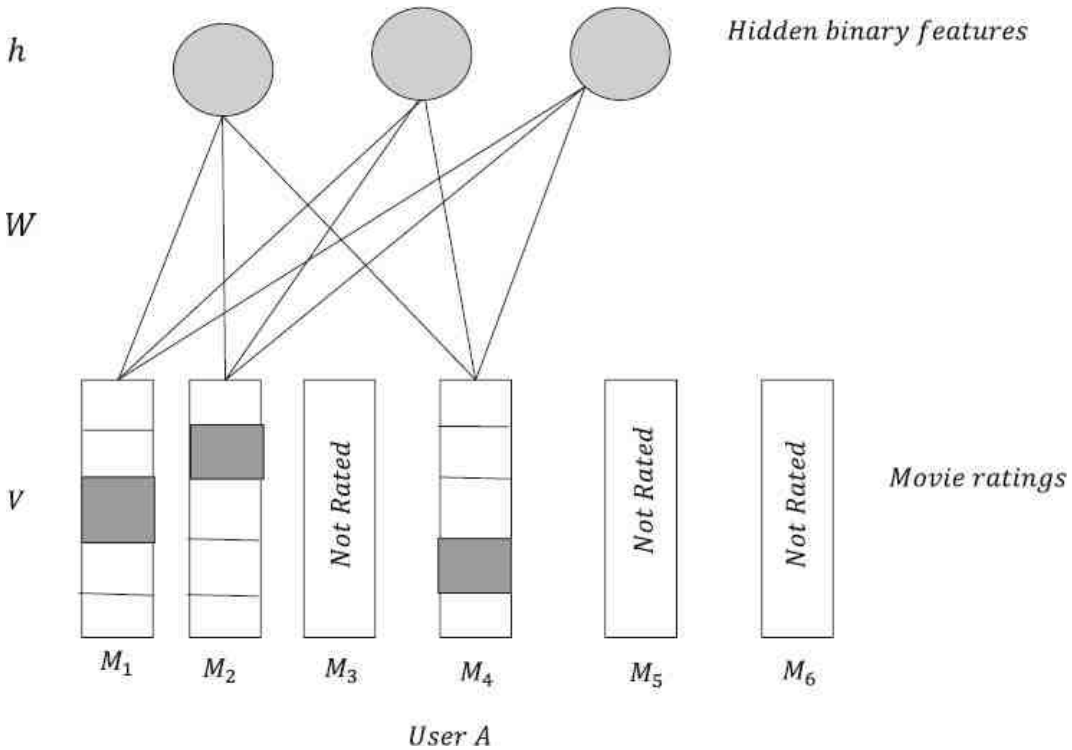


Figure 5-10: Restricted Boltzmann view for User A

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

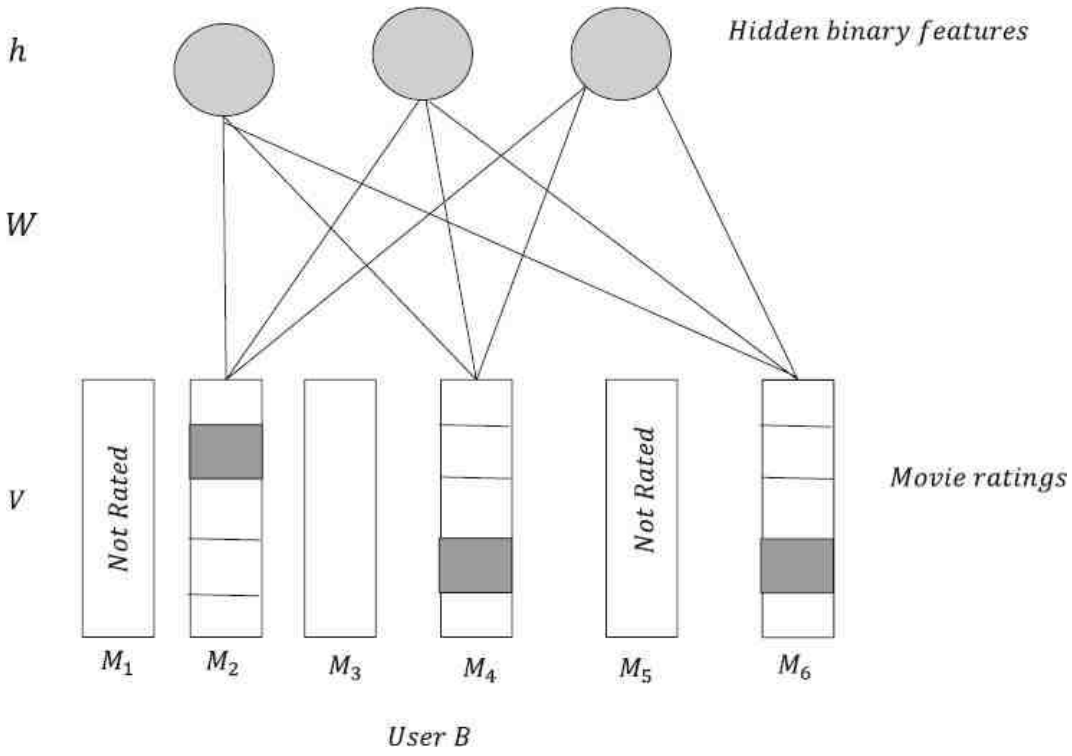


Figure 5-11: Restricted Boltzmann view for User B

As we can see from Figure 5-10 and Figure 5-11, the restricted Boltzmann views for User A and User B are different since they differ in the selection of movies they have rated. However, for the movies they both have rated, the weight connections are the same. This kind of architecture—where each user's RBM is trained separately while the RBMs share weights for same movies—helps overcome the problem of missing ratings and at the same time allows generalized weights for a movie to hidden layer connection for all the users. From each movie to the hidden unit and vice versa there are actually five connections, one for each of the possible ratings for a movie. However, to keep the representation simple, only one combined connection has been shown in the diagrams. Each of the models can be trained separately through gradient descent using contrastive divergence, and the model weights can be averaged across the different RBMs so that all the RBMs share the same weights.

From 5.3.17, we have for the binary visible unit the following:

$$P(v_i = 1/h) = \sigma\left(b_i + \sum_{j=1}^{n} h_j w_{ij}\right)$$

Now that the visible units have *K* possible ratings, the visible units are *K* dimensional vectors with only one index corresponding to the actual rating set to 1, and the rest all are set to zero. So, the new expression of the probability of a rating over *K* possible ratings would be given by a SoftMax function. Also, do note that the *m* in this case is the number of movies a user has watched and would vary for different RBMs for different users. The constant *n* denotes the number of hidden units in the hidden layer for each RBM.

(5.4.1)
$$P\left(v_i^{(k)} = 1/h\right) = \frac{e^{\left(b_i^{(k)} + \sum_{j=1}^{n} h_j w_{ij}^{(k)}\right)}}{\sum_{l=1}^{K} e^{\left(b_i^{(l)} + \sum_{j=1}^{n} h_j w_{ij}^{(l)}\right)}}$$

where $w_{ij}^{(k)}$ is the weight connecting the *kth* rating index for visible unit *i* to the *jth* hidden unit and $b_i^{(k)}$ represents the bias at the visible unit *i* for its *kth* rating.

The energy of a joint configuration *E*(*v, h*) is given by

(5.4.2)
$$E(v,h) = -\sum_{k=1}^{K}\sum_{i=1}^{m} b_i^{(k)} v_i^{(k)} - \sum_{j=1}^{n} c_j h_j - \sum_{k=1}^{K}\sum_{j=1}^{n}\sum_{i=1}^{m} v_i^{(k)} w_{ij}^{(k)} h_j$$

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

So,

(5.4.3) $$P(v,h) \propto e^{-E(v,h)} = e^{\sum_{k=1}^{K}\sum_{i=1}^{m} b_i^{(k)} v_i^{(k)} + \sum_{j=1}^{n} c_j h_j + \sum_{k=1}^{K}\sum_{j=1}^{n}\sum_{i=1}^{m} v_i^{(k)} w_{ij}^{(k)} h_j}$$

The probability of the hidden unit given the input *v* is

(5.4.4) $$P(hj=1/v) = \frac{e^{\left(c_j + \sum_{i=1}^{m}\sum_{k=1}^{K} v_i^{(k)} w_{ij}^{(k)}\right)}}{1+e^{\left(c_j + \sum_{i=1}^{m}\sum_{k=1}^{K} v_i^{(k)} w_{ij}^{(k)}\right)}}$$

Now, the obvious question: How do we predict the rating for a movie a user has not seen? As it turns out, the computation for this is not that involved and can be computed in linear time. The most informative way of making that decision is to condition the probability of the user, provided rating *r* to a movie *q* is conditioned on the movie ratings the user has already provided. Let the movie ratings the user has already provided be denoted by *V*. So, we need to compute the probability $P(v_q^{(k)}/V)$ as follows:

(5.4.5) $$P\left(v_q^{(k)}/V\right) = \sum_h P\left(v_q^{(k)}, h/V\right) = \frac{\sum_h P\left(v_q^{(k)}, h, V\right)}{P(V)}$$

Since *P*(*V*) is fixed for all movie ratings *k*, from (5.4.5) we have

(5.4.6) $$P\left(v_q^{(k)}/V\right) \propto \sum_h P\left(v_q^{(k)}, h, V\right)$$
$$\propto \sum_h e^{-E\left(v_q^{(k)}, h, V\right)}$$

This is a three-way energy configuration and can be computed easily by adding the contribution of $v_q^{(k)}$ in (5.4.2), as shown here:

(5.4.7) $$E\left(v_q^{(k)}, V, h\right) = -\sum_{k=1}^{K}\sum_{i=1}^{m} b_i^{(k)} v_i^{(k)} - \sum_{j=1}^{n} c_j h_j - \sum_{k=1}^{K}\sum_{j=1}^{n}\sum_{i=1}^{m} v_i^{(k)} w_{ij}^{(k)} h_j - \sum_{j=1}^{n} v_s^{(k)} w_{sj}^{(k)} h_j - v_s^{(k)} b^{(k)}$$

Substituting $v_q^{(k)} = 1$ in (5.4.7), one can find the value of $E\left(v_q^{(k)} = 1, V, h\right)$,which is proportional to

$P\left(v_q^{(k)} = 1, V, h\right)$

For all *K* values of rating *K* the preceding quantity $E\left(v_q^{(k)} = 1, V, h\right)$, needs to be computed and then normalized to form probabilities. One can then either take the value of *k* for which the probability is maximum or compute the expected value of *k* from the derived probabilities, as shown here:

$$\hat{k} = \underbrace{argmax}_{k} P\left(v_q^{(k)} = 1/V\right)$$

$$\hat{k} = \sum_{k=1}^{5} k \times P\left(v_q^{(k)} = 1/V\right)$$

The expectation way of deriving the rating turns out to give better predictions than a hard assignment of a rating based on the maximum probability.

Also, one simple way to derive the probability of the rating *k* for a specific unrated movie *q* by a user with rating matrix *V* is to first sample the hidden states *h* given the visible ratings input *V*; i.e., draw $h \sim P(h/V)$. The hidden units are common to all

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

and hence carry information about patterns for all movies. From the sampled hidden units we try to sample the value of $v_q^{(k)}$; i.e., draw $v_q^{(k)} \sim P\left(v_q^{(k)} / h\right)$. This back-to-back sampling, first from $V \rightarrow h$ and then from $h \rightarrow v_q^{(k)}$, is equivalent to sampling $v_q^{(k)} \sim P\left(v_q^{(k)} / V\right)$. I hope this helps in providing an easier interpretation.

Deep Belief Networks (DBNs)

Deep belief networks are based on the restricted Boltzmann machine but, unlike an RBN, a DBN has multiple hidden layers. The weights in each hidden layer *K* are trained by keeping all the weights in the prior (*K* – 1) layers constant. The activities of the hidden units in the (*K* – l) layer are taken as input for the *Kth* layer. At any particular time during training two layers are involved in learning the weight connections between them. The learning algorithm is the same as that of restricted Boltzmann machines. Illustrated in Figure 5-12 is a high-level schematic diagram for a deep belief network.

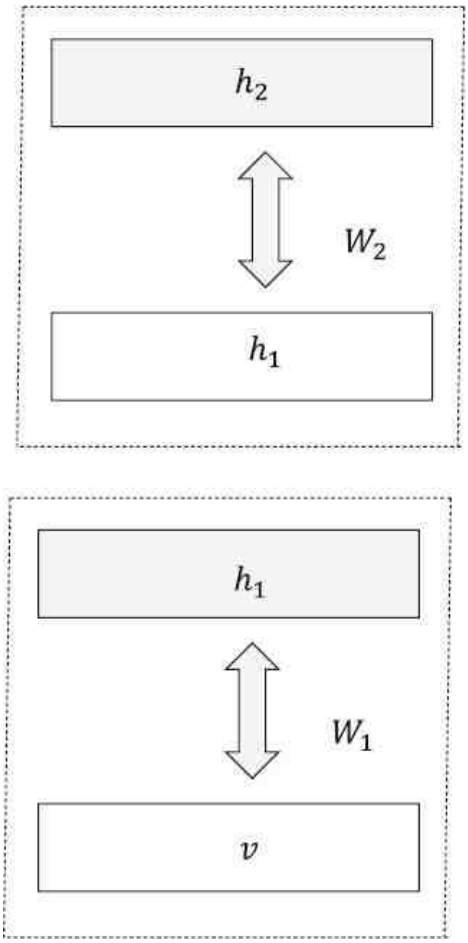


Figure 5-12: Deep belief network using RBMs

Like RBM, in DBN each layer can be trained by gradient descent using contrastive divergence. The DBN learning algorithm is used to learn the initial weights of a deep network being used for supervised learning so that the network has a good set of initial weights to start with. Once the pre-training is done for the deep belief network, we can add an output layer to the DBN based on the supervised problem at hand. Let's say we want to train a model to perform classification on the MNIST dataset. In that case, we would have to append a ten-class SoftMax layer. We can then fine-tune the model by using backpropagation of error. Since the model already has an initial set of weights from unsupervised DBN learning, the model would have a good chance of converging faster when backpropagation is invoked.

Whenever we have sigmoid units in a network, if the network weights are not initialized properly there is a high chance that one might have a vanishing-gradient problem. This is because the output of sigmoid units are linear within a small range, after which the output saturates, leading to near-zero gradients. Since the backpropagation is essentially a chain rule of derivatives, the gradient of the cost function with respect to any weight would have sigmoid gradients from the layers prior to it from a backpropagation order. So, if few of the gradients in the sigmoid layers are operating in the saturated regions and producing gradients close to zero, the latter layers, gradients of the cost function with respect to the weights, would be close to zero, and

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

there is a high chance that the learning would stop. When the weights are not properly initialized, there is a high chance that the network sigmoid units could go into the unsaturated region and lead to near-zero gradients in the sigmoid units. However, when the network weights are initialized by DBN learning, there is less of a chance of the sigmoid units' operating in the saturated zone. This is because the network has learned something about the data while it was pre-training, and there is a smaller chance that the sigmoid units will operate in saturated zones. Such problems with the activation unit saturating are not present for ReLU activation functions since they have a constant gradient of 1 for input values greater than zero.

We now look at an implementation of DBN pre-training of weights followed by the training of a classification network by appending the output layer to the hidden layer of the RBM. In Listing 5-3a we implemented RBM, wherein we learned the weights of the visible-to-hidden connections, assuming all the units are sigmoid. To that RBM we are going to stack the output layer of ten classes for the MNIST dataset and train the classification model using the weights from the visible-to-hidden units learned as the initial weights for the classification network. Of course, we would have a new set of weights corresponding to the connection of the hidden layer to the output layer. See the detailed implementation in Listing 5-3b.

Listing 5-3b: Basic Implementation of DBN

```
##Import the Required libraries
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline

## Read the MNIST files
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

## Set up the parameters for training

n_visible      = 784
n_hidden    = 500
display_step = 1
num_epochs = 200
batch_size = 256
lr         = tf.constant(0.001, tf.float32)
learning_rate_train = tf.constant(0.01, tf.float32)
n_classes = 10
training_iters = 200
## Define the tensorflow variables for weights and biases as well as placeholder for input
x  = tf.placeholder(tf.float32, [None, n_visible], name="x")
y  = tf.placeholder(tf.float32, [None,10], name="y")

W  = tf.Variable(tf.random_normal([n_visible, n_hidden], 0.01), name="W")
b_h = tf.Variable(tf.zeros([1, n_hidden],  tf.float32, name="b_h"))
b_v = tf.Variable(tf.zeros([1, n_visible],  tf.float32, name="b_v"))
W_f = tf.Variable(tf.random_normal([n_hidden,n_classes], 0.01), name="W_f")
b_f = tf.Variable(tf.zeros([1, n_classes],  tf.float32, name="b_f"))
## Converts the probability into discrete binary states i.e. 0 and 1
def sample(probs):
    return tf.floor(probs + tf.random_uniform(tf.shape(probs), 0, 1))

## Gibbs sampling step
def gibbs_step(x_k):
        h_k = sample(tf.sigmoid(tf.matmul(x_k, W) + b_h))
        x_k = sample(tf.sigmoid(tf.matmul(h_k, tf.transpose(W)) + b_v))
        return x_k
## Run multiple Gibbs Sampling steps starting from an initial point
def gibbs_sample(k,x_k):
    for i in range(k):
        x_out = gibbs_step(x_k)
# Returns the gibbs sample after k iterations
    return x_out

# Constrastive Divergence algorithm
# 1. Through Gibbs sampling locate a new visible state x_sample based on the current visible
state x
# 2. Based on the new x sample a new h as h_sample
x_s = gibbs_sample(2,x)
h_s = sample(tf.sigmoid(tf.matmul(x_s, W) + b_h))

# Sample hidden states based given visible states
h = sample(tf.sigmoid(tf.matmul(x, W) + b_h))
# Sample visible states based given hidden states
x_ = sample(tf.sigmoid(tf.matmul(h, tf.transpose(W)) + b_v))
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
# The weight updated based on gradient descent
size_batch = tf.cast(tf.shape(x)[0], tf.float32)
W_add  = tf.multiply(lr/size_batch, tf.subtract(tf.matmul(tf.transpose(x), h), tf.matmul(tf.
transpose(x_s), h_s)))
bv_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(x, x_s), 0, True))
bh_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(h, h_s), 0, True))
updt = [W.assign_add(W_add), b_v.assign_add(bv_add), b_h.assign_add(bh_add)]
#############################################################
## Ops for the Classification Network
#############################################################
h_out = tf.sigmoid(tf.matmul(x, W) + b_h)
logits = tf.matmul(h_out,W_f) + b_f
prob = tf.nn.softmax(logits)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate_train).minimize(cost)
correct_pred = tf.equal(tf.argmax(logits,1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

## Ops for the hidden unit activation

# TensorFlow graph execution

with tf.Session() as sess:
    # Initialize the variables of the Model
    init = tf.global_variables_initializer()
    sess.run(init)

    total_batch = int(mnist.train.num_examples/batch_size)
    # Start the training
    for epoch in range(num_epochs):
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Run the weight update
            batch_xs = (batch_xs > 0)*1
            _ = sess.run([updt], feed_dict={x:batch_xs})

        # Display the running step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1))

    print("RBM training Completed !")

    out = sess.run(h,feed_dict={x:(mnist.test.images[:20]> 0)*1})
    label = mnist.test.labels[:20]

    plt.figure(1)
    for k in range(20):
        plt.subplot(4, 5, k+1)
        image = (mnist.test.images[k]> 0)*1
        image = np.reshape(image,(28,28))
        plt.imshow(image,cmap='gray')

    plt.figure(2)

    for k in range(20):
        plt.subplot(4, 5, k+1)
        image = sess.run(x_,feed_dict={h:np.reshape(out[k],(-1,n_hidden))})
        image = np.reshape(image,(28,28))
        plt.imshow(image,cmap='gray')
        print(np.argmax(label[k]))
    ###################################################
    ### Invoke the Classification Network training now
   ###################################################
    for i in xrange(training_iters):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        if i % 10 == 0:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([cost, accuracy], feed_dict={x: batch_x,
                                                              y: batch_y})
            print "Iter " + str(i) + ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= " + \
                  "{:.5f}".format(acc)
    print "Optimization Finished!"

    # Calculate accuracy for 256 mnist test images
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
    print "Testing Accuracy:", \
        sess.run(accuracy, feed_dict={x: mnist.test.images[:256],
                                      y: mnist.test.labels[:256]})

    sess.close()

--output--

Iter 0, Minibatch Loss= 11.230852, Training Accuracy= 0.06641
Iter 10, Minibatch Loss= 2.809783, Training Accuracy= 0.60938
Iter 20, Minibatch Loss= 1.450730, Training Accuracy= 0.75000
Iter 30, Minibatch Loss= 0.798674, Training Accuracy= 0.83594
Iter 40, Minibatch Loss= 0.755065, Training Accuracy= 0.87891
Iter 50, Minibatch Loss= 0.946870, Training Accuracy= 0.82812
Iter 60, Minibatch Loss= 0.768834, Training Accuracy= 0.89062
Iter 70, Minibatch Loss= 0.445099, Training Accuracy= 0.92188
Iter 80, Minibatch Loss= 0.390940, Training Accuracy= 0.89062
Iter 90, Minibatch Loss= 0.630558, Training Accuracy= 0.90234
Iter 100, Minibatch Loss= 0.633123, Training Accuracy= 0.89844
Iter 110, Minibatch Loss= 0.449092, Training Accuracy= 0.92969
Iter 120, Minibatch Loss= 0.383161, Training Accuracy= 0.91016
Iter 130, Minibatch Loss= 0.362906, Training Accuracy= 0.91406
Iter 140, Minibatch Loss= 0.372900, Training Accuracy= 0.92969
Iter 150, Minibatch Loss= 0.324498, Training Accuracy= 0.91797
Iter 160, Minibatch Loss= 0.349533, Training Accuracy= 0.93750
Iter 170, Minibatch Loss= 0.398226, Training Accuracy= 0.90625
Iter 180, Minibatch Loss= 0.323373, Training Accuracy= 0.93750
Iter 190, Minibatch Loss= 0.555020, Training Accuracy= 0.91797
Optimization Finished!
Testing Accuracy: 0.945312
```

As we can see from the preceding output, with the pre-trained weights from RBM used as initial weights for the classification network we can get good accuracy of around 95 percent on the MNIST test dataset by just running it for 200 batches. This is impressive given that the network does not have any convolutional layers.

Auto-encoders

Auto-encoders are unsupervised artificial neural networks that are used to generate a meaningful internal representation of the input data. The auto-encoder network generally consists of three layers—the input layer, the hidden layer, and the output layer. The input layer and hidden layer combination acts as the encoder while the hidden layer and output layer combination acts as the decoder. The encoder tries to represent the input as a meaningful representation at the hidden layer while the decoder reconstructs the input back into its original dimension at the output layer. Typically, some cost function between the reconstructed input and the original input is minimized as part of the training process.

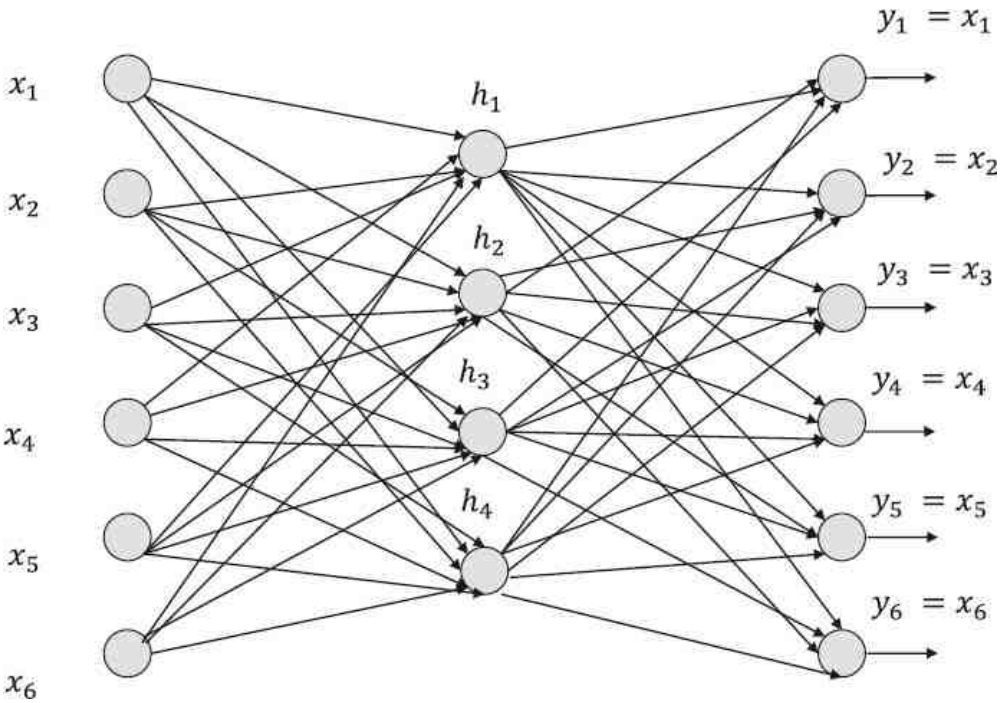


Figure 5-13: Architecture of a basic auto-encoder

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Figure 5-13 represents a basic auto-encoder with one hidden layer and the input and the output layers. The input $x=\left[x_1 x_2\ x_3\ ..\ x_6\right]^T \in \mathbb{R}^{6\times 1}$ while the hidden layer $h=\left[h_1 h_2\ h_3 h_4\right]^T \in \mathbb{R}^{4\times 1}$. The output *y* is chosen to be equal to *x* so that the error between the reconstructed input $\hat{y}$ can be minimized so as to get a meaningful representation of the input in the hidden layer. For generality purposes, let's take

$$x=\left[x_1 x_2\ x_3\ ..\ x_n\right]^T \in \mathbb{R}^{n\times 1}$$

$$h=\left[h_1 h_2\ h_3\ ..\ h_d\right]^T \in \mathbb{R}^{d\times 1}$$

$$y=x=\left[y_1 y_2\ y_3\ ..\ y_n\right]^T \in \mathbb{R}^{n\times 1}$$

Let the weights from *x* to *h* be represented by the weight matrix $W \in \mathbb{R}^{d\times n}$ and the biases at the hidden unit be represented by $b=\left[b_1 b_2\ b_3\ ..\ h_d\right]^T \in \mathbb{R}^{d\times 1}$.

Similarly, let the weights from *h* to *y* be represented by the weight matrix $W' \in \mathbb{R}^{n\times d}$ and the biases at the output units be represented as $b'=\left[\ b_1 b_2\ b_3\ ..\ b_n\right]^T \in \mathbb{R}^{n\times 1}$

The output of the hidden unit can be expressed as

$$h=f_1\left(Wx+b\right)$$

where f_1 is the element-wise activation function at the hidden layer. The activation function can be linear, ReLU, sigmoid, and so forth depending on its use.

Similarly, the output of the output layer can be expressed as

$$\hat{y}=f_2\left(W'h+b'\right)$$

If the input features are of a continuous nature, one can minimize a least square-based cost function as follows to derive the model weights and biases based on the training data:

$$C=\sum_{k=1}^{m}\left\|\hat{y}^{(k)}-y^{(k)}\right\|_2^2=\sum_{k=1}^{m}\left\|\hat{y}^{(k)}-x^{(k)}\right\|_2^2$$

where $\left\|\hat{y}^{(k)}-x^{(k)}\right\|_2^2$ is the Euclidean or the l^2 norm distance between the reconstructed output vector and the original input vector and m is the number of data points on which the model is trained on.

If we represent all the parameters of the model by the vector $\theta=[W;b;W';b']$ then the cost function *C* can be minimized with respect to all the parameters of the model θ to derive the model

$$\hat{\theta}=\underbrace{Arg\ Min}_{\theta} C(\theta)=\underbrace{Arg\ Min}_{\theta}\sum_{k=1}^{m}\left\|\hat{y}^{(k)}-x^{(k)}\right\|_2^2$$

The learning rule of the model as per gradient descent is

$$\theta^{(t+1)}=\theta^{(t)}-\epsilon\nabla_\theta C\left(\theta^{(t)}\right)$$

where ϵ is the learning rate, *t* represents the iteration number, and $\nabla_\theta C(\theta^{(t)})$ is the gradient of the cost function with respect to θ at $\theta=\theta^{(t)}$.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Now, let us consider several cases as follows:

- When the dimension of the hidden layer is less than that of the input layer, i.e., *(d < n)* where *d* is the hidden layer and *n* is the dimension of input layer, then the auto-encoder works as a data-compression network that projects the data from a high-dimension space to a lower-dimension space given by the hidden layer. This is a lossy data-compression technique. It can also be used for noise reduction in the input signal.
- When ($d < n$) and all the activation functions are linear then the network learns to do a linear PCA (principal component analysis).
- When ($d \geq n$) and the activation functions are linear then the network may learn an identity function, which might not be of any use. However, if the cost function is regularized to produce a sparse hidden representation then the network may still learn an interesting representation of the data.
- Complex non-linear representations of input data can be learned by having more hidden layers in the network and by making the activation functions non-linear. A schematic diagram of such a model is represented in Figure 5-14. When taking multiple hidden layers, it is a must that one takes non-linear activation functions to learn non-linear representations of data since several layers of linear activations are equivalent to a single linear activation layer.

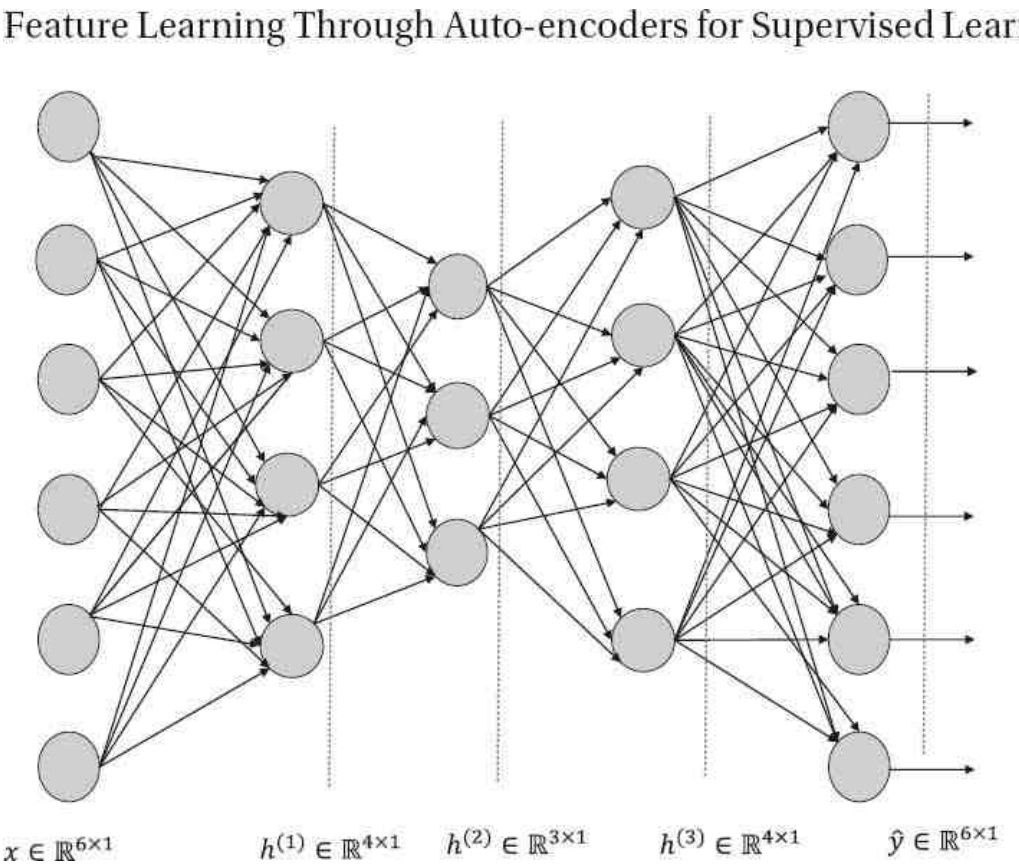


Figure 5-14: Auto-encoder with multiple hidden layers

When we deal with multiple hidden layers, as shown in Figure 5-14, and have a non-linear activation function in the neural units, then the hidden layers learn non-linear relations between the variables of the input data. If we are working on a classification-related problem of two classes where the input data is represented by $x \in \mathbb{R}^{6\times1}$ we can learn interesting non-linear features by training the auto-encoder as in Figure 5-14 and then using the output of the second hidden layer vector $h^{(2)} \in \mathbb{R}^{3\times1}$ feature representation given by $h^{(2)}$ can be used as the input to a classification model, as shown in Figure 5-15. When the hidden layer whose output we are interested in has a dimensionality less than that of the input, it is equivalent to the non-linear version of principal component analysis wherein we are just consuming the important non-linear features and discarding the rest as noise.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

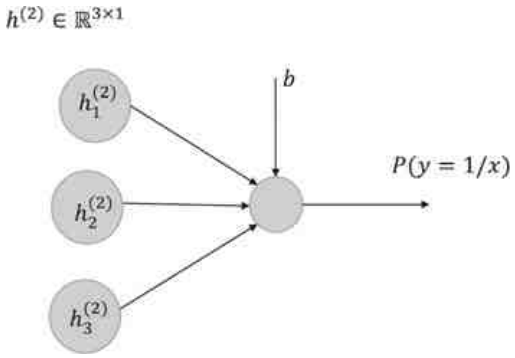


Figure 5-15: Classifier with features learned from auto-encoder

The overall network can be combined into a single network for class-prediction purpose at test time by combining the two networks as shown in Figure 5-16. From the auto-encoder, only the part of the network up to the second hidden layer that is producing output $h^{(2)}$ needs to be considered, and then it needs to be combined with the classification network as in Figure 5-15.

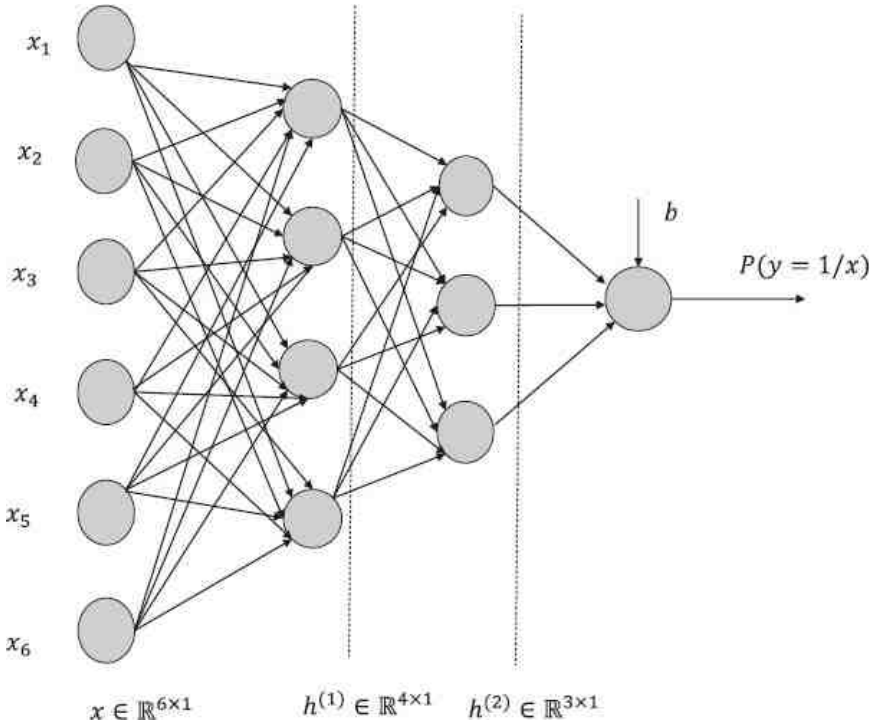


Figure 5-16: Combined classification network for prediction of classes

One might ask the obvious question: Why would linear PCA not suffice for the task the auto-encoder is performing in this example? Linear PCA or Principal Component Analysis only takes care of the capturing the linear relationship between input variables and tries to decompose the input variables into components that are not linearly dependent on each other. These components are called *principal components* and are uncorrelated with each other, unlike the original input variables. However, the input variables are not always going to be related in a linear fashion that leads to linear correlation. Input variables might be correlated in much more complex, non-linear ways, and such non-linear structures within the data can be captured only through non-linear hidden units in the auto-encoders.

Kullback-Leibler (KL) Divergence

The KL divergence measures the disparity or divergence between two random Bernoulli variables. If two random Bernoulli variables X and Y have means of ρ_1 and ρ_2 respectively, then the KL divergence between the variables X and Y is given by

$$KL(\rho_1 \| \rho_2) = \rho_1 \log\left(\frac{\rho_1}{\rho_2}\right) + (1-\rho_1)\log\left(\frac{1-\rho_1}{1-\rho_2}\right)$$

From the preceding expression, we can see that the KL divergence is 0 when $\rho_1 = \rho_2$; i.e., when both distributions are identical.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

When $\rho_1 \neq \rho_2$, the KL divergence increases monotonically with the difference of the means. If ρ_1 is chosen to be 0.2 then the KL divergence versus the ρ_2 plot is as expressed in Figure 5-17.

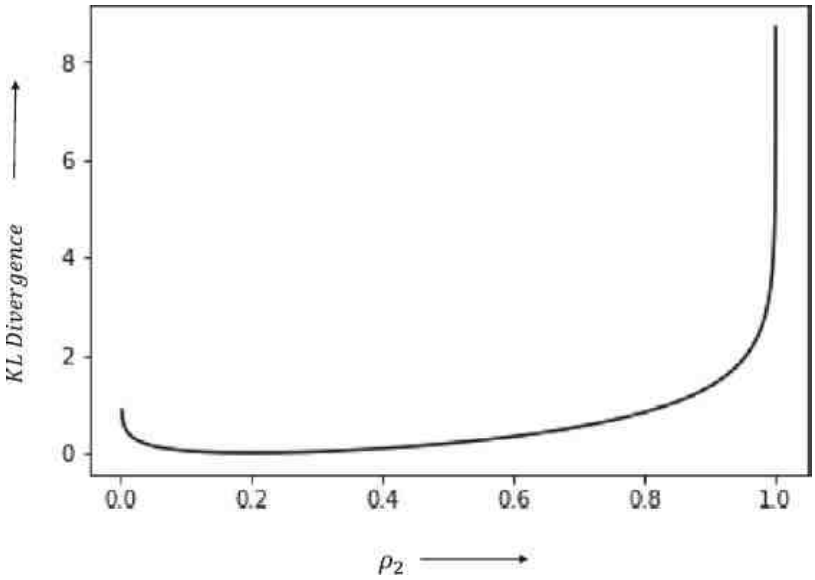


Figure 5-17: KL divergence plot for mean ρ_1 =0.2

As we can see, the KL divergence is at its minimum at $\rho_2 = \rho_1 = 0.2$ and increases monotonically on either side of $\rho_2 = 0.2$. We will be using KL divergence to introduce sparsity in the hidden layer for sparse auto-encoders in the next section.

Sparse Auto-encoders

The purpose of auto-encoders, as we discussed earlier, is to learn interesting hidden structures of the input data or, more specifically, to learn interesting relations among the different variables in the input. The most common way to derive these hidden structures is to make the hidden layer dimension smaller than the input data dimension so that the auto-encoder is forced to learn a compressed representation of the input data. This compressed representation is forced to reconstruct the original data, and hence the compressed representation should have enough information to capture the input sufficiently well. This compressed representation will only be able to capture the input data efficiently if there is redundancy in the data in the form of correlation and other non-linear associations between the input variables. If the input features are relatively independent, then such compression would not be able to represent the original data well. So, for the auto-encoder to give an interesting low-dimensional representation of the input data, the data should have enough structure in it in the form of correlation and other non-linear associations between input variables.

One thing that we touched upon earlier is that when the number of hidden layer units is larger than the dimensionality of the input, there is a high possibility that the auto-encoder will learn identity transform after setting the weights corresponding to the extra hidden layers to zero. In fact, when the number of input and hidden layers are the same, the optimal solution for the weight matrix connecting the input to the hidden layer is the identity matrix. However, even when the number of hidden units is larger than the dimensionality of the input, the auto-encoder can still learn interesting structures within the data, provided some constraints. One such constraint is to restrict the hidden layer output to be sparse so that those activations in the hidden layer units on average are close to zero. We can achieve this sparsity by adding a regularization term to the cost function based on KL divergence. Here, the ρ_1 will be very close to zero, and the average activation in the hidden unit over all training samples would act as the ρ_2 for that hidden unit. Generally, ρ_1 is selected to be very small, to the order of 0.04, and hence if the average activation in each of the hidden units is not close to 0.04 then the cost function would be penalized.

Let $h \in \mathbb{R}^{d \times 1}$ be the hidden layer sigmoid activations of the input $x \in \mathbb{R}^{n \times 1}$ where $d > n$. Further, let the weights connecting the inputs to the hidden layer be given by $W \in \mathbb{R}^{d \times n}$ and the weights connecting the hidden layer to the output layer be given by $W' \in \mathbb{R}^{n \times d}$. If the bias vectors at the hidden and output layers are given by b and b' respectively, then the following relationship holds true:

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

$$h^{(k)} = \sigma(Wx^{(k)} + b)$$

$$\hat{y}^{(k)} = f(W'h^{(k)} + b')$$

where $h^{(k)}$ and $y'^{(k)}$ are the hidden layer output vector and the reconstructed input vector for the *kth* input training data point. The cost function to be minimized with respect to the parameters of the model (i.e., *W, W', b, b'*) is given by

$$C = \sum_{k=1}^{m} \left\| \hat{y}^{(k)} - x^{(k)} \right\|_2^2 + \lambda \sum_{j=1}^{d} KL(\rho \,\|\, \hat{\rho}_j)$$

where $\hat{\rho}_j$ is the average activation in the *jth* unit of the hidden layer over all the training samples and can be represented as follows. Also, $h_j^{(k)}$ represents the hidden layer activation at unit *j* for the *kth* training sample.

$$\hat{\rho}_j = \frac{1}{m} \sum_{k=1}^{m} h_j^{(k)}$$

$$KL\left(\rho \left\| \hat{\rho}_j \right.\right) = \rho \log\left(\frac{\rho}{\hat{\rho}_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_j}\right)$$

Generally, ρ is selected as 0.04 to 0.05 so that the model learns to produce average hidden layer unit activations very close to 0.04, and in the process the model learns sparse representation of the input data in the hidden layer.

Sparse auto-encoders are useful in computer vision to learn low-level features that represent the different kinds of edges at different locations and orientations within the natural images. The hidden layer output gives the weight of each of these low-level features, which can be combined to reconstruct the image. If 10×10 images are processed as 100-dimensional input, and if there are 200 hidden units, then the weights connecting input to hidden units—i.e., *W* or hidden units to the output reconstruction layer *W'*—would comprise 200 images of size 100 (10×10). These images can be displayed to see the nature of the features they represent. Sparse encoding works well when supplemented with PCA whitening, which we will discuss briefly later in this chapter.

Sparse Auto-Encoder Implementation in TensorFlow

In this section, we will implement a sparse auto-encoder that has more hidden units than the input dimensionality. The dataset for this implementation is the MNIST dataset. Sparsity has been introduced in the implemented network through KL divergence. Also, the weights of the encoder and decoder are used for L2 regularization to ensure that in the pursuit of sparsity these weights don't adjust themselves in undesired ways. The auto-encoder and decoder weights represent over-represented basis, and each of these basis tries to learn some low-level feature representations of the images, as discussed earlier. The encoder and the decoder are taken to be the same. These weights that make up the low-level feature images have been displayed to highlight what they represent. The detailed implementation has been outlined in Listing 5-4.

Listing 5-4

```
## Import the required libraries and data

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import time
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# Parameters for training the Network
learning_rate = 0.001
training_epochs = 200
batch_size = 1000
display_step = 1
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
examples_to_show = 10

# Network Parameters
# Hidden units are more than the input dimensionality since the intention
# is to learn sparse representation of hidden units
n_hidden_1 = 32*32
n_input = 784 # MNIST data input (img shape: 28*28)

X = tf.placeholder("float", [None, n_input])

weights = {
    'encoder_h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
}
biases = {
    'encoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'decoder_b1': tf.Variable(tf.random_normal([n_input])),
}

# Building the encoder
def encoder(x):
    # Encoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']),
                                   biases['encoder_b1']))

    return layer_1

# Building the decoder
def decoder(x):
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, tf.transpose(weights['decoder_h1'])),
                                   biases['decoder_b1']))
    return layer_1

## Define the log-based function to be used in computing the KL Divergence

def log_func(x1, x2):
    return tf.multiply(x1, tf.log(tf.div(x1,x2)))

def KL_Div(rho, rho_hat):
    inv_rho = tf.subtract(tf.constant(1.), rho)
    inv_rhohat = tf.subtract(tf.constant(1.), rho_hat)
    log_rho = logfunc(rho,rho_hat) + log_func(inv_rho, inv_rhohat)
    return log_rho

# Model definition
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)
rho_hat = tf.reduce_mean(encoder_op,1)
# Reconstructed output
y_pred = decoder_op
# Targets in the input data.
y_true = X

# Define the TensorFlow Ops for loss and optimizer, minimize the combined error
# Squared Reconstruction error
cost_m = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
# KL Divergence Regularization to introduce sparsity
cost_sparse = 0.001*tf.reduce_sum(KL_Div(0.2,rho_hat))
# L2 Regularization of weights to keep the network stable
cost_reg = 0.0001* (tf.nn.l2_loss(weights['decoder_h1']) + tf.nn.l2_loss(weights
['encoder_h1']))
# Add up the costs
cost = tf.add(cost_reg,tf.add(cost_m,cost_sparse))

optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(cost)

# Initializing the variables
init = tf.global_variables_initializer()

# Launch the Session graph
start_time = time.time()
with tf.Session() as sess:
    sess.run(init)
    total_batch = int(mnist.train.num_examples/batch_size)

    for epoch in range(training_epochs):
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            _, c = sess.run([optimizer, cost], feed_dict={X: batch_xs})
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1),
                  "cost=", "{:.9f}".format(c))

    print("Optimization Finished!")

    # Applying encode and decode over test set
    encode_decode = sess.run(
        y_pred, feed_dict={X: mnist.test.images[:10]})
    # Compare original images with their reconstructions
    f, a = plt.subplots(2, 10, figsize=(10, 2))
    for i in range(10):
        a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
        a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))
    # Store the Decoder and Encoder Weights
    dec = sess.run(weights['decoder_h1'])
    enc = sess.run(weights['encoder_h1'])
end_time = time.time()
print('elapsed time:',end_time - start_time)

-- Output --
```

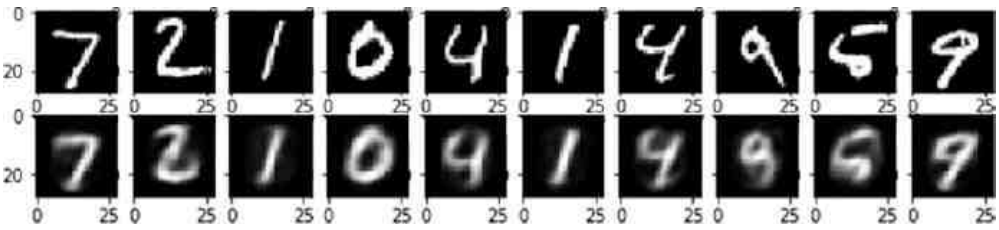


Figure 5-18: Display of the original image followed by the reconstructed image

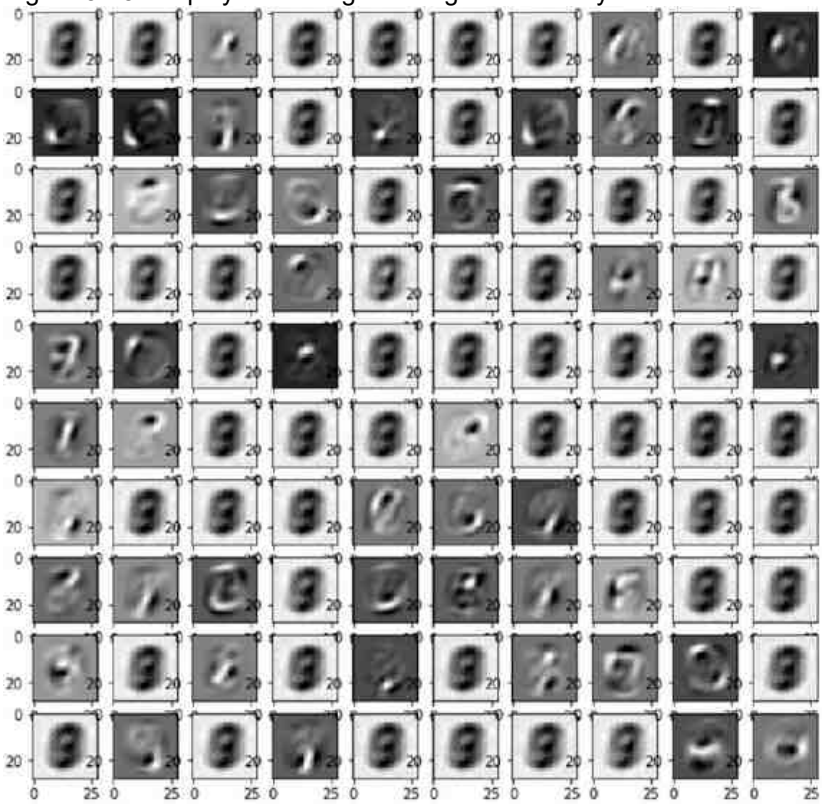


Figure 5-19: Display of a few encoder/decoder weights as images

Figure 5-18 shows the reconstruction of the images by sparse encoders while Figure 5-19 shows the decoder weights in the form of images. The weights corresponding to hidden unit layers are images being displayed in Figure 5-19. This gives you some idea as to the kind of features the sparse encoders are learning. The final image that is reconstructed is the linear combination of these images, with the hidden layer activations acting as the linear weights. Essentially, each of these images is detecting low-level features in the form of hand strokes for the written digits. In terms of linear algebra, these images form a basis for representing the reconstructed images.

Denoising Auto-Encoder

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

Denoising auto-encoders works like a standard auto-encoder, with non-linear activations in the hidden layers, the only difference being that instead of the original input *x*, a noisy version of *x*, say *x*, is fed to the network. The reconstructed image at the output layer is compared with the actual input *x* while computing the error in reconstruction. The idea is that the hidden structured learned from the noisy data is rich enough to reconstruct the original data. Hence, this form of auto-encoder can be used for reducing noise in the data since it learns a robust representation of the data from the hidden layer. For example, if an image has been blurred by some distortion then a denoising auto-encoder can be used to remove the blur. An auto-encoder can be converted into a denoising auto-encoder by just introducing a stochastic noise addition unit.

For images, denoising auto-encoders can have hidden layers as convolutional layers instead of standard neural units. This ensures that the topological structure of the image is not compromised when defining the auto-encoder network.

A Denoising Auto-Encoder Implementation in TensorFlow

In this section, we will work through the implementation of a denoising auto-encoder that learns to remove noise from input images. Two kinds of noise have been introduced to the input images—namely, Gaussian and salt and pepper noise—and the implemented denoising auto-encoder can remove both of these efficiently. The detailed implementation is illustrated in Listing 5-5.

Listing 5-5: Denoising Auto-Encoder Using Convolution and Deconvolution Layers

```
# Import the required library
import tensorflow.contrib.layers as lays
import numpy as np
from skimage import transform
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt

#  Define the Network with Encoder and Decoder
def autoencoder(inputs):
    # encoder
net = lays.conv2d(inputs, 32, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d(net, 16, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d(net, 8, [5, 5], stride=4, padding='SAME')
    # decoder
    net = lays.conv2d_transpose(net, 16, [5, 5], stride=4, padding='SAME')
    net = lays.conv2d_transpose(net, 32, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d_transpose(net, 1, [5, 5], stride=2, padding='SAME', activation_fn=tf.
nn.tanh)
    return net

def resize_batch(imgs):
# Function to resize the image to 32x32 so that the dimensionality can be reduced in
# multiples of 2
    imgs = imgs.reshape((-1, 28, 28, 1))
    resized_imgs = np.zeros((imgs.shape[0], 32, 32, 1))
    for i in range(imgs.shape[0]):
        resized_imgs[i, ..., 0] = transform.resize(imgs[i, ..., 0], (32, 32))
    return resized_imgs

## Function to introduce Gaussian Noise
def noisy(image):
    row,col= image.shape
    mean = 0
    var = 0.1
    sigma = var**0.5
    gauss = np.random.normal(mean,sigma,(row,col))
    gauss = gauss.reshape(row,col)    noisy = image + gauss
    return noisy

## Function to define Salt and Pepper Noise
def s_p(image):
    row,col = image.shape
    s_vs_p = 0.5
    amount = 0.05
    out = np.copy(image)
# Salt mode
    num_salt = np.ceil(amount * image.size * s_vs_p)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.shape]
    out[coords] = 1
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
      # Pepper mode
    num_pepper = np.ceil(amount* image.size * (1. - s_vs_p))
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.shape]
    out[coords] = 0
    return out
# Defining the ops

# input to which the reconstucted signal is compared to
a_e_inputs = tf.placeholder(tf.float32, (None, 32, 32, 1))
# input to the network (MNIST images)
a_e_inputs_noise = tf.placeholder(tf.float32, (None, 32, 32, 1))
a_e_outputs = autoencoder(a_e_inputs_noise)  # create the Auto-encoder network
# calculate the loss and optimize the network
loss = tf.reduce_mean(tf.square(a_e_outputs - a_e_inputs))  # claculate the mean square
error loss
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)

# initialize the network
init = tf.global_variables_initializer()

# Invoking the TensorFlow Graph for Gaussian Noise reduction auto-encoder training and
validation

batch_size = 1000  # Number of samples in each batch
epoch_num = 10     # Number of epochs to train the network
lr = 0.001        # Learning rate

# read MNIST dataset
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# calculate the number of batches per epoch
batch_per_ep = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(epoch_num):
        for batch_num in range(batch_per_ep):
            batch_img, batch_label = mnist.train.next_batch(batch_size)  # read a batch
            batch_img = batch_img.reshape((-1, 28, 28, 1))               # reshape each
sample to an (28, 28) image
            batch_img = resize_batch(batch_img)                          # reshape the
images to (32, 32)
## Introduce noise in the input images
            image_arr = []
            for i in xrange(len(batch_img)):
                img = batch_img[i,:,:,0]
                img = noisy(img)
                image_arr.append(img)
            image_arr = np.array(image_arr)
            image_arr = image_arr.reshape(-1,32,32,1)
            _, c = sess.run([train_op, loss], feed_dict={a_e_inputs_noise:image_arr,a_e_
inputs: batch_img})
            print('Epoch: {} - cost= {:.5f}'.format((ep + 1), c))

    # test the trained network
    batch_img, batch_label = mnist.test.next_batch(50)
    batch_img = resize_batch(batch_img)
    image_arr = []

    for i in xrange(50):
        img = batch_img[i,:,:,0]
        img = noisy(img)
        image_arr.append(img)
    image_arr = np.array(image_arr)
    image_arr = image_arr.reshape(-1,32,32,1)

    reconst_img = sess.run([ae_outputs], feed_dict={ae_inputs_noise: image_arr})[0]

    # plot the reconstructed images and the corresponding Noisy images
    plt.figure(1)
    plt.title('Input Noisy Images')
    for i in range(50):
        plt.subplot(5, 10, i+1)
        plt.imshow(image_arr[i, ..., 0], cmap='gray')

    plt.figure(2)
plt.title('Re-constructed Images')
    for i in range(50):
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
        plt.subplot(5, 10, i+1)
        plt.imshow(reconst_img[i, ..., 0], cmap='gray')
    plt.show()

--Output--
```

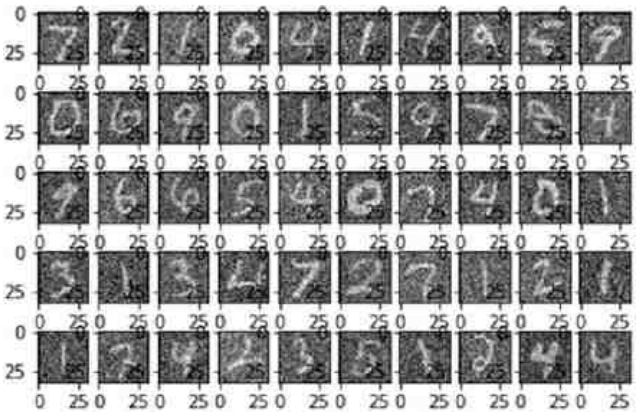


Figure 5-20: Images with Gaussian noise

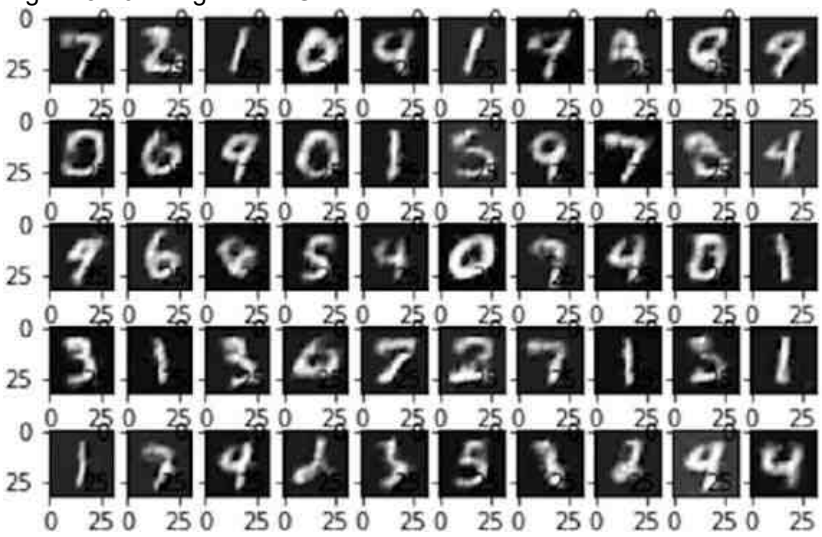


Figure 5-21: Reconstructed Images (without Gaussian Noise) generated by the Denoising Auto Encoder

We can see from Figure 5-20 and Figure 5-21 that the Gaussian noise has been removed by the denoising auto-encoders.

```
# Invoking the TensorFlow Graph for Salt and Pepper Noise reduction auto-encoder training
and validation

batch_size = 1000  # Number of samples in each batch
epoch_num = 10     # Number of epochs to train the network
lr = 0.001        # Learning rate

# read MNIST dataset
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# calculate the number of batches per epoch
batch_per_ep = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(epoch_num):
        for batch_num in range(batch_per_ep):
            batch_img, batch_label = mnist.train.next_batch(batch_size)  # read a batch
            batch_img = batch_img.reshape((-1, 28, 28, 1))               # reshape each
sample to an (28, 28) image
            batch_img = resize_batch(batch_img)                          # reshape the
images to (32, 32)
## Introduce noise in the input images
            image_arr = []
            for i in xrange(len(batch_img)):
                img = batch_img[i,:,:,0]
                img = noisy(img)
```


Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

```
                image_arr.append(img)
            image_arr = np.array(image_arr)
            image_arr = image_arr.reshape(-1,32,32,1)
            _, c = sess.run([train_op, loss], feed_dict={a_e_inputs_noise:image_arr,a_e_
inputs: batch_img})
            print('Epoch: {} - cost= {:.5f}'.format((ep + 1), c))

    # test the trained network
    batch_img, batch_label = mnist.test.next_batch(50)
    batch_img = resize_batch(batch_img)
    image_arr = []

    for i in xrange(50):
        img = batch_img[i,:,:,0]
        img = noisy(img)
        image_arr.append(img)
    image_arr = np.array(image_arr)
    image_arr = image_arr.reshape(-1,32,32,1)

    reconst_img = sess.run([ae_outputs], feed_dict={ae_inputs_noise: image_arr})[0]

    # plot the reconstructed images and the corresponding Noisy images    plt.figure(1)
    plt.title('Input Noisy Images')
    for i in range(50):
        plt.subplot(5, 10, i+1)
        plt.imshow(image_arr[i, ..., 0], cmap='gray')

    plt.figure(2)
 plt.title('Re-constructed Images')
    for i in range(50):
        plt.subplot(5, 10, i+1)
        plt.imshow(reconst_img[i, ..., 0], cmap='gray')
    plt.show()
```

--Output--

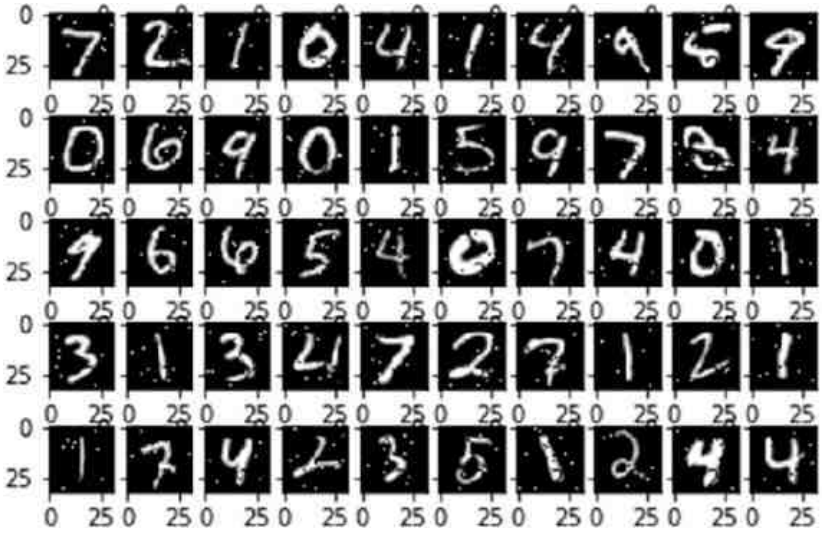


Figure 5-22: Salt and pepper noisy images

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

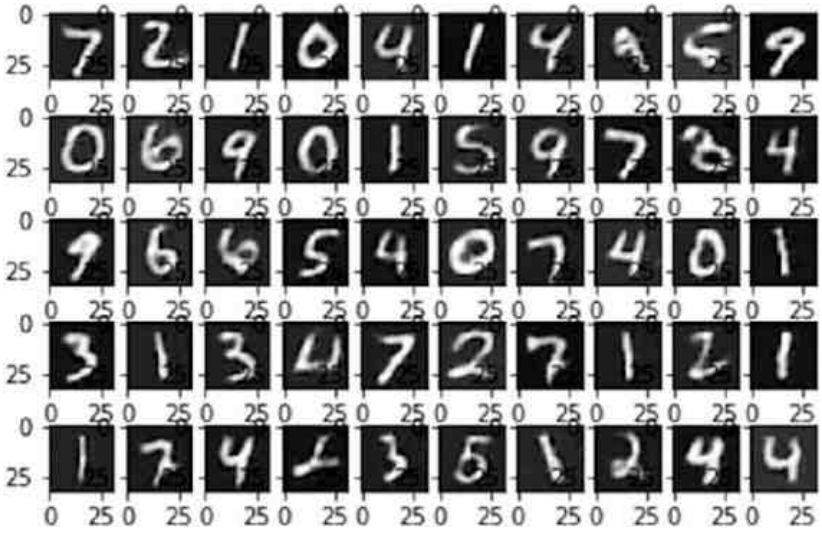


Figure 5-23: Reconstructed images without the salt and pepper noise generated by the denoising auto-encoder

From Figure 5-22 and Figure 5-23 it is evident that the denoising auto-encoder does a good job of removing the salt and pepper noise. Do note that the auto-encoders are trained separately, once for handling Gaussian noise and once for handling salt and pepper noise.

PCA and ZCA Whitening

Generally, images contain pixels whose intensities are highly correlated in any neighborhood of the image, and hence such correlation is highly redundant to a learning algorithm. These dependencies in the form of correlation between nearby pixels is generally of little use to any algorithm. Thus, it makes sense to remove this two-way correlation so that the algorithm puts more emphasis on higher-order correlations. Similarly, the mean intensity of an image might not be of any use to a learning algorithm in cases where the image is a natural image. Therefore, it makes sense to remove the mean intensity of an image. Do note that we are not subtracting the mean per-pixel location, but rather the mean of pixel intensities of each image. This kind of mean normalization is different from the other mean normalization we do in machine learning where we subtract the mean per feature computed over a training set. Coming back to the concept of whitening, the advantages of whitening are two-fold:

- Remove the correlation among features in the data
- Make the variance equal along each feature direction

PCA and ZCA whitening are two techniques generally used to pre-process images before the images are processed through artificial neural networks. These techniques are almost the same, with a subtle difference. The steps involved in PCA whitening are illustrated first, followed by ZCA.

- Remove the mean pixel intensity from each image separately. So, if a 2D image is converted into a vector, one can subtract the mean of the elements in the image vector from itself. If each image is represented by the vector $x^{(t)} \in \mathbb{R}^{n \times 1}$, where i represents the *ith* image in the training set, then the mean normalized image for $x^{(t)}$ is given by

$$x^{(i)} = x^{(i)} - \frac{1}{n}\sum_{j=1}^{n} x_j^{(i)}$$

- Once we have the mean normalized images we can compute the covariance matrix as follows:

$$C = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} x^{(i)T}$$

- Next, we need to decompose the covariance matrix through singular value decomposition (SVD) as follows:

$$C = UDU^T$$

- In general, SVD decomposes as $C = UDV^T$, but since C is a symmetric matrix, $U = V$ in this case. U gives the Eigen vectors of the covariance matrix. The Eigen vectors are aligned in a column-wise fashion in U. The variances of the data along the direction of the Eigen vectors are given by the Eigen values housed along the diagonals of D, while the rest of

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

the entries in *D* are zero since *D* is the covariance matrix for the uncorrelated directions given by the Eigen vectors.

- In PCA whitening, we project the data along the Eigen vectors, or one may say principal components, and then divide the projection value in each direction by the square root of the Eigen value—i.e., the standard deviation along that direction on which the data is projected. So, the PCA whitening transformation is as follows:

$$T = D^{-\frac{1}{2}}U^T$$

- Once this transformation is applied to the data, the transformed data has zero correlation and unit variance along the newly transformed components. The transformed data for original mean-corrected image $x^{(t)}$ is

$$x^{(i)}_{PW} = Tx^{(i)}$$

The problem with PCA whitening is that although it decorrelates the data and makes the new feature variances unity, the features are no longer in the original space but rather are in a transformed rotated space. This makes the structure of objects such as images lose a lot of information in terms of their spatial orientation, because in the transformed feature space each feature is the linear combination of all the features. For algorithms that make use of the spatial structure of the image, such as convolutional neural networks, this is not a good thing. So, we need some way to whiten the data such that the data is decorrelated and of unit variances along its features but the features are still in the original feature space and not in some transformed rotated feature space. The transformation that provides all these relevant properties is called ZCA transform. Mathematically, any orthogonal matrix *R* (the column vectors of which are orgothogonal to each other) when multiplied by the PCA whitening transform *T* produces another whitening transform. If one chooses, *R* = *U*, and the transform

$$Z = UT = UD^{-\frac{1}{2}}U^T$$

is called the ZCA transform. The advantage of ZCA transform is that the image data still resides in the same feature space of pixels, and hence, unlike in PCA whitening, the original pixel doesn't get obscured by the creation of new features. At the same time, the data is whitened—i.e., decorrelated—and of unit variance for each of the features. The unit variance and decorrelated features may help several machine-learning or deep-learning algorithms achieve faster convergence. At the same time, since the features are still in the original space they retain their topological or spatial orientation, and algorithms such as convolutional neural networks that make use of the spatial structure of the image can make use of the information.

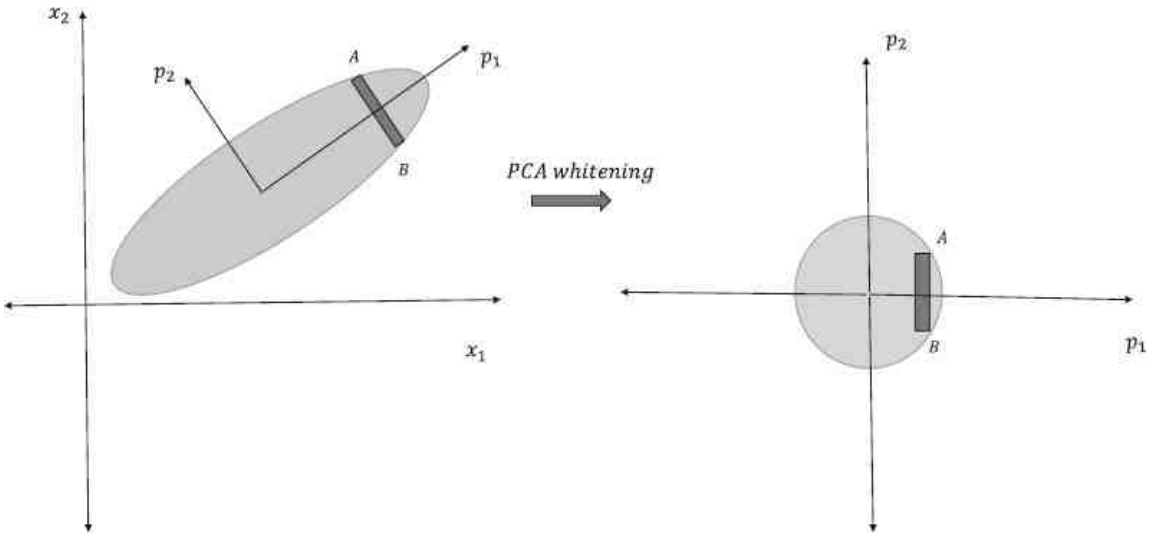


Figure 5-24: PCA whitening illustration

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited

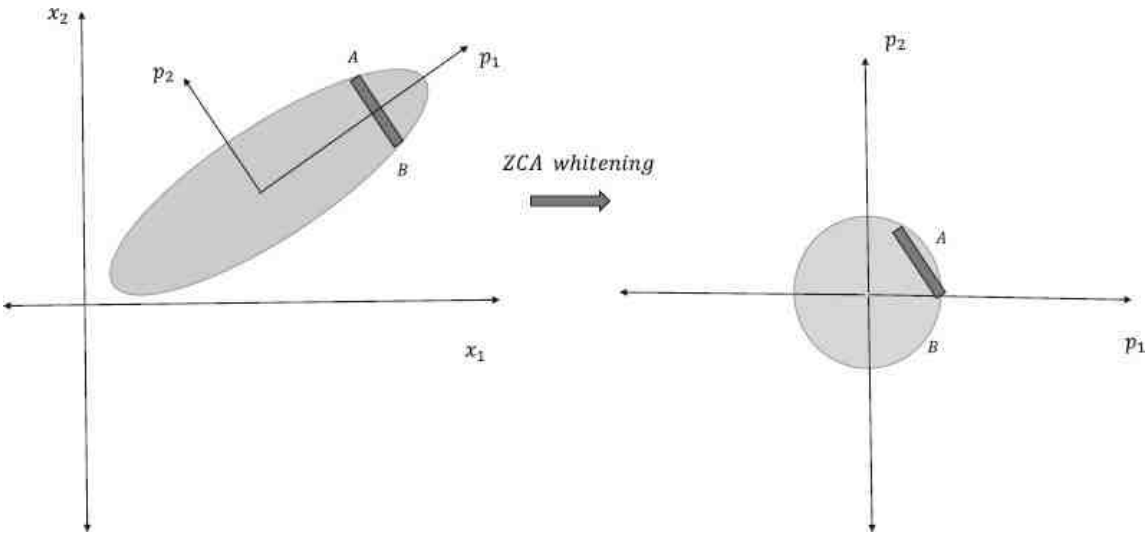


Figure 5-25: ZCA whitening illustration

The key difference between PCA whitening and ZCA whitening is illustrated in Figure 5-24 and Figure 5-25. As we can see, in both case the 2D correlated data is transformed into uncorrelated new data. However, there is a major difference. While in PCA whitening, the new axes have changed from the original axes based on the principal components given by p_1 and p_2, the axes remain same as those of the original with ZCA whitening. The p_1 and p_2 are the Eigen vectors of the co-variance matrix for the data. Also, we see the orientation of the marker *AB* has changed in the new axes for PCA whitening while it remains intact for ZCA whitening. In both cases, the idea is to get rid of the not-so-useful two-way covariances between the input variables so that the model can concentrate on learning about higher-order correlations.

Summary

In this chapter, we went through the most popular unsupervised techniques in deep learning, namely, restricted Boltzmann machines and auto-encoders. Also, we discussed the different applications of using these methods and the training process related to each of these algorithms. Finally, we ended with PCA and ZCA whitening techniques, which are relevant pre-processing techniques used in several supervised deep-learning methods. By the end of this chapter, we had touched upon all the core methodologies in deep learning. Other improvised methods in deep learning can be easily comprehended and implemented given the methods and mathematics touched upon thus far.

In the next and final chapter, we will discuss several improvised deep-learning networks that have gained popularity in recent times, such as generative adversarial networks, R-CNN, and so forth, and touch upon aspects of taking a TensorFlow application into production with ease.

Reprinted for PYH3C/2362626, Indiana University
Apress, Santanu Pattanayak (c) 2017, Copying Prohibited