

Multimodal Deep Learning Library

Jian Jin

Supervised by: Raman Arora

Department of Computer Science, Johns Hopkins University

December 19, 2015

Contents

1	Multimodal Deep Learning Library	4
1.1	Introduction	4
1.2	Content Description	4
2	Network Survey	5
2.1	Neural Network	5
2.2	Markov Random Field	5
2.3	Belief Network	6
2.4	Deep Learning Models	6
3	Restricted Boltzmann Machine	8
3.1	Logic of Restricted Boltzmann Machine	8
3.2	Training of Restricted Boltzmann Machine	10
3.3	Tricks in Restricted Boltzmann Machine Training	11
3.4	Classifier of Restricted Boltzmann Machine	13
3.5	Implementation of RBM	15
3.6	Summary	16
4	Deep Neural Network	17
4.1	Construction of Deep Neural Network	17
4.2	Fine Tuning of Deep Neural Network	18
4.3	Implementation of Deep Neural Network	19
4.4	Summary	20
5	Deep Belief Network	21
5.1	Logic of Deep Belief Network	21
5.2	Training of Deep Belief Network	22
5.3	Classification of Deep Belief Network	23
5.4	Implementation of Deep Belief Network	24
5.5	Summary	25
6	Denoising Autoencoder	26
6.1	Construction of Autoencoder	26
6.2	Fine tuning of Autoencoder	27
6.3	Denoising Autoencoder	29
6.4	Implementation of Denoising Autoencoder	30
6.5	Summary	30
7	Deep Boltzmann Machine	31
7.1	Logic of Deep Boltzmann Machine	31
7.2	Pretraining of Deep Boltzmann Machine	31
7.3	Mean Field Inference	33
7.4	Implementation of Deep Boltzmann Machine	33

8	Multimodal Learning Model	35
8.1	Deep Canonical Correlation Analysis	35
8.1.1	Canonical Correlation Analysis	35
8.1.2	Kernel Canonical Correlation Analysis	36
8.1.3	Deep Correlation Analysis	36
8.2	Modal Prediction and Transformation	37
9	Library Structure	39
9.1	Data Reading	39
9.1.1	MNIST	39
9.1.2	CIFAR	39
9.1.3	XRMB	39
9.1.4	AvLetters	40
9.1.5	Data Processing	40
9.2	Computation and Utilities	41
9.3	Models and Running	42
10	Performance	43
10.1	Restricted Boltzmann Machine	43
10.2	Deep Neural Network	43
10.3	Denoising Autoencoder	43
10.4	Deep Belief Network	44
10.5	Deep Boltzmann Machine	45
10.6	Deep Canonical Correlation Analysis	45
10.7	Modal Prediction	45
	References	46

1 Multimodal Deep Learning Library

1.1 Introduction

This is the document of Multimodal Deep Learning Library, MDL, which is written in C++. It explains principles and implementations with details of Restricted Boltzmann Machine, Deep Neural Network, Deep Belief Network, Denoising Autoencoder, Deep Boltzmann Machine, Deep Canonical Correlation Analysis, and modal prediction model.

MDL uses OpenCV 3.0.0, which is the only dependency of this library. Most of its implementation has been tested in Mac OS. It also provides interface for reading various data set such as MNIST, CIFAR, XRMB, and AVLetters. To read mat file, Matlab must be installed because it uses Matlab/c++ interface provided by Matlab.

There are multiple model options provided. Different gradient descent methods, loss function, annealing methods, and activation functions are given. These options are easy to extend given the structure of MDL. So MDL could be used as a frame for testings in deep learning.

1.2 Content Description

Section 2 goes through common networks. Section 3 to section 7 give descriptions of Restricted Boltzmann Machine, Deep Neural Network, Deep Belief Network, Denoising Autoencoder, Deep Boltzmann Machine, and Deep Canonical Correlation Analysis respectively. Section 8 introduces multimodal learning models. Section 9 gives explanation of the library structure. Section 10 presents performance.

In sections 3 to 7, each section explains principles and implementations of each model. Section 8 introduces two models in Multimodal learning.

2 Network Survey

This section includes a brief description of network models that are mentioned in this documents and a summary of deep learning models of MDL.

2.1 Neural Network

The Neural Network is a directed graph consists of multiple layers of neurons, which is also referred to as units. In general there is no connection between units of the same layer and there are only connections between adjacent layers. The first layer is the input and is referred to as visible layer v . Above the visible layer there are multiple hidden layers $\{h_1, h_2, \dots, h_n\}$. And the output of the last hidden layer forms the output layer o .

In hidden layers, neurons in layer h_i receives input from the previous layer, h_{i-1} or v , and the output of h_i is the input to the next layer, h_{i+1} or o . The value transmitted between layers is called action or emission. Action is computed as:

$$a_i^{(k)} = f\left(\sum_{j=1}^{n_{k-1}} a_j^{(k-1)} w_{ij}^k + b_i^{(k)}\right) = f(z_i^{(k)}) \quad (1)$$

where g is the activation function that enables nonlinear representation of the Neural Network. Without activation function the network could only represent linear combination of the input and its power would be much weaker. A common activation function is sigmoid function. $z_i^{(k)}$ is the total input of the unit i in layer k . It is computed as a weighted sum of the activations of the previous layer. The weight w_{ij}^k and the bias b_i^k is learned in backpropagation.

A process called forward propagation computes actions layer by layer. Learning of neural network uses backpropagation, which is a supervised learning algorithm. It computes error based on the network output and the training label, then uses this error to compute gradient of error with respect to the weights and biases of each layer, and updates model parameters by gradient descent.

2.2 Markov Random Field

A Markov Random Field, also called the Markov Network, is an undirected graphical model in which each node is independent of the other nodes given all the nodes connected to it. It describes the distribution of variables in the graph.

The Markov Random Field uses energy to describe the distribution over the graph

$$P(u) = \frac{1}{Z} e^{-E(u)}, \quad (2)$$

where Z is a partition function defined by

$$Z = \sum_u e^{-E(u)}, \quad (3)$$

E is the energy specified by the model, and u is the set of states of all the nodes such that

$$u = \{v_1, v_2, \dots, v_n\} \quad (4)$$

where v_i is the state of node i .

2.3 Belief Network

The Belief Network, which is also referred to as the Bayesian Network, is an directed acyclic graph for probabilistic reasoning. It exhibits the conditional dependencies of the models by associating each node X with a conditional probability $P(X|Pa(X))$ where $Pa(X)$ denotes the parents of X . Here are two of its conditional independence property:

1. Each node is conditionally independent of its non-descendants given its parents.
2. Each node is conditionally independent of all other nodes given its Markov blanket, which consists of its parents, children, and children's parents.

The inference of Belief Network is to compute the posterior probability distribution

$$P(H|V) = \frac{P(H, V)}{\sum_H P(H, V)} \quad (5)$$

where H is the set of query variables that forms hidden units, and V is the set of evidence variables that forms visible units. Approximate inference involves sampling to compute posterior.

The Sigmoid Belief Network is a type of the Belief Network such that

$$P(X_i = 1|Pa(X_i)) = \sigma\left(\sum_{X_j \in Pa(X_i)} W_{ji}X_j + b_i\right) \quad (6)$$

where W_{ji} is the weight assigned to the edge from X_j to X_i .

2.4 Deep Learning Models

The Restricted Boltzmann Machine is a type of Markov Random Field and is trained in an unsupervised manner. It is the building block for other models and could be used for classification by adding a classifier on top of it.

The Deep Neural Network is a neural network with multiple layers. Each layer is initialized by pretraining a Restricted Boltzmann Machine. Then fine tuning would refine the parameters of the model.

The Deep Belief Network is a hybrid of the Restricted Boltzmann Machine and the Sigmoid Belief Network. It is a generative model, and is not a feedforward neural network or multilayer perceptron even though its training is similar to the Deep Neural

Network.

The Denoising Autoencoder is a type of neural network that has symmetric structure. It could reconstruct the input data and if properly trained could reconstruct corrupted images. However, unlike neural networks, It could be trained in an unsupervised manner.

The Deep Boltzmann Machine is another type of Markov Random Field. It is pre-trained by stacking Restricted Boltzmann Machines with adjusted weights and biases as an approximation to undirected graphs. Its fine tuning uses a method called mean field inference.

The Deep Canonical Correlation Analysis is used for learning multiview or multimodal data by adding a Kernel Canonical Correlation Analysis layer on top of two networks. It finds the projections of the outputs that maximizes their correlation.

3 Restricted Boltzmann Machine

3.1 Logic of Restricted Boltzmann Machine

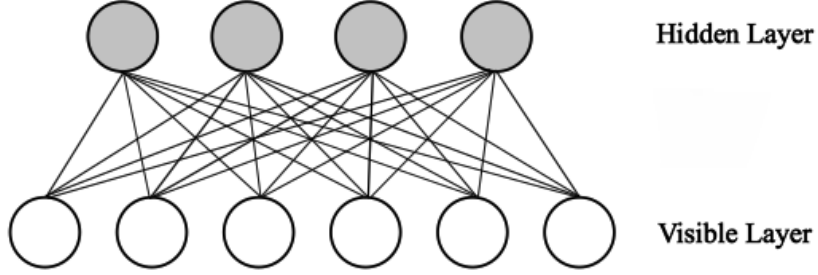


Figure 3.1: Restricted Boltzmann Machine

A Restricted Boltzmann Machine (RBM) is a Markov Random Field consisting of one hidden layer and one visible layer. It is an undirected bipartite graph in which connections are between the hidden layer and the visible layer. Each unit x is a stochastic binary unit such that

$$state(x) = \begin{cases} 1, & p \\ 0, & 1 - p \end{cases} \quad (7)$$

where probability p is defined by the model. Figure 3.1 shows a RBM with four hidden units and six visible units.

As a Markov Random Field, a Restricted Boltzmann Machine defines the distribution over the visible layer v and the hidden layer h as

$$P(v, h) = \frac{1}{Z} e^{-E(v, h)}, \quad (8)$$

where Z is a partition function defined by

$$Z = \sum_{v, h} e^{-E(v, h)}, \quad (9)$$

Its energy $E(v, h)$ is defined by

$$E(v, h) = - \sum_i b_i^v v_i - \sum_j b_j^h h_j - \sum_i \sum_j v_i w_{i,j} h_j \quad (10)$$

where b_i^v is the bias of the i th visible unit and b_j^h is the bias of the j th hidden unit.

Given the conditional independence property of the Markov Random Field, in RBM probability of one layer given the other layer could be factorized as

$$P(h|v) = \prod_j P(h_j|v) \quad (11)$$

$$P(v|h) = \prod_i P(v_i|h). \quad (12)$$

Plug equation 10 in equation 8:

$$P(h|v) = \frac{P(h, v)}{\sum_{h'} P(h', v)} = \frac{\exp(\sum_i b_i^v v_i + \sum_j b_j^h h_j + \sum_i \sum_j v_i w_{i,j} h_j)}{\sum_{h'} \exp(\sum_i b_i^v v_i + \sum_j b_j^{h'} h'_j + \sum_i \sum_j v_i w_{i,j} h'_j)}. \quad (13)$$

And through derivation one could get

$$\frac{\exp(\sum_i b_i^v v_i + \sum_j b_j^h h_j + \sum_i \sum_j v_i w_{i,j} h_j)}{\sum_{h'} \exp(\sum_i b_i^v v_i + \sum_j b_j^{h'} h'_j + \sum_i \sum_j v_i w_{i,j} h'_j)} = \prod_j \frac{\exp(b_j^h + \sum_{i=1}^m W_{i,j} v_i)}{1 + \exp(b_j^h + \sum_{i=1}^m w_{i,j} v_i)}. \quad (14)$$

Combine equation 11, 13, and 14 one could get

$$P(h_j = 1|v) = \sigma \left(b_j^h + \sum_{i=1}^{N_v} W_{i,j} v_i \right) \quad (15)$$

where σ is a sigmoid function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}. \quad (16)$$

Similarly,

$$P(v_i = 1|h) = \sigma \left(b_i^v + \sum_{j=1}^{N_h} W_{i,j} h_j \right). \quad (17)$$

A Restricted Boltzmann Machine maximizes the likelihood $P(x)$ of the input data x , which is

$$P(x) = \sum_h P(x, h) = \sum_h \frac{1}{Z} e^{-E(h, x)} = \frac{1}{Z} e^{-F(x)} \quad (18)$$

where $F(x)$ is called Free Energy such that

$$F(x) = - \sum_{i=1}^{n_v} b_i^v x_i - \sum_{j=1}^{n_h} \log(1 + \exp(b_j^h + \sum_{k=1}^{n_v} W_{k,j} x_k)). \quad (19)$$

In training, maximizing the likelihood of the training data is achieved by minimizing the negative log likelihood of the training data. Because the direct solution is intractable, gradient descent is used, in which weights $\{W_{ij}\}$, biases of hidden units $\{b_j^h\}$, and biases of visible units $\{b_i^v\}$ are updated. The gradient is approximated by an algorithm called Contrastive Divergence. More details are in the training section.

Express the hidden states $\{h_j\}$ in a row vector h , hidden biases $\{b_j^h\}$ in a row vector b^h , weights $\{W_{ij}\}$ in a matrix W indicating the weight from the visible layer to the hidden layer, and σ is element-wise performed, this could be expressed as

$$a^h = \sigma(v * W + b^h). \quad (20)$$

Since these edges are undirected, it is assumed that this mutual transmission has tied weights such that

$$a^v = \sigma(h * W^T + b^v). \quad (21)$$

In training, training data is first used as a visible layer to acquire hidden states.

3.2 Training of Restricted Boltzmann Machine

Training of Restricted Boltzmann Machine is the process of updating the weight and the biases of hidden and visible layers by gradient descent. Instead of Stochastic Gradient Descent, which performs update on each data sample, batch learning is used in RBM so that each update is based on a batch of training data. Using the same training data as input, several epochs are run for updating.

For instance if the input data has 10,000 samples and the number of batches is 200, then there will be 200 updates in each epoch. For each update, gradients will be based on 50 samples. If the number of epochs is 10, there should be a total of 2000 updates in the training process if the gradients computed are not too trivial such that the updates stop before 2000 times.

The gradients of weights are given by Contrastive Divergence as:

$$\nabla W_{ij} = \langle v_i * h_j \rangle_{recon} - \langle v_i * h_j \rangle_{data} \quad (22)$$

where the angle brackets are expectations under the distribution specified by the subscript. The expectations here are approximated by sample mean. So it would be

$$\nabla W_{ij} = \sum_{k=1}^m ((v_i * h_j)_{recon_k} - (v_i * h_j)_{data_k}) / m \quad (23)$$

where m is the size of a batch.

States of the visible layer and hidden layers, which are current values stores in all visible units and hidden units, form a sample in Gibbs sampling, based on which $(v_i * h_j)_{recon_n}$ is computed. Gibbs sampling is a Markov chain Monte Carlo method to obtain observations from a given joint distribution. In RBM, Gibbs sampling works in the following manner:

In Gibbs sampling, k samples of $X = (x_1, \dots, x_n)$ are made from a joint distribution $p(x_1, \dots, x_n)$ by sampling each component variable from a conditional probability. Specifically, in the $(i + 1)$ th sample, $x_j^{(i+1)}$ is sampled from $p(x_j | x_1^{(i+1)}, \dots, x_{j-1}^{(i+1)}, x_{j+1}^{(i)}, \dots, x_n^{(i)})$. Sampling each component variable x_i once forms a sample. To acquire k samples in Gibbs sampling this process needs to be done k times, in which the latest sampled component variables are used to compute conditional probability.

Each unit is a stochastic binary unit. So the state is either 0 or 1. To sample h_j from

$$P(h_j = 1 | v) = \sigma \left(b_j + \sum_{i=1}^m W_{i,j} v_i \right), \quad (24)$$

simply compute $a_j = \sigma(b_j + \sum_{i=1}^m W_{i,j} v_i)$. If a_j is larger than a random sample from uniform distribution, its state is 1, otherwise 0. The probability is a_j that a random

sample u from uniform distribution is smaller than a_j because

$$P(u < a_j) = F_{\text{uniform}}(a_j) = a_j. \quad (25)$$

So we have

$$P(h_j = 1|v) = P(u < a_j). \quad (26)$$

Thus we could sample by testing if $u < a_j$, since it has the same probability from which we want to sample. There are only two states of a unit, so if $u < a_j$ fails the state must be 0. That is why this method of sampling from conditional probability works.

In training of RBM, first use input training data as the visible layer to sample the hidden layer from

$$P(h_j = 1|v) = \sigma \left(b_j + \sum_{i=1}^m W_{i,j} v_i \right) \quad (27)$$

then run a Gibbs sampling to sample the visible layer based on the latest state of the hidden layer, and the hidden layer based on the visible layer just sampled. It has the following sequence:

training data >> hidden layer >> visible layer >> hidden layer

The Contrastive Divergence states that performs one Gibbs sampling is enough for RBM training. The training data and the activation of the first sampled hidden layer are respectively v_{data} and h_{data} . The sampled visible states and the activation of the second sampled hidden layer are respectively v_{recon} and h_{recon} . In this way the gradient of weight could be computed.

The gradient of visible bias is

$$\nabla b_i^v = \langle v_i \rangle_{recon} - \langle v_i \rangle_{data} \quad (28)$$

and the gradient of hidden bias is

$$\nabla b_j^h = \langle h_j \rangle_{recon} - \langle h_j \rangle_{data} \quad (29)$$

3.3 Tricks in Restricted Boltzmann Machine Training

Dropout

Dropout is a method to prevent neural networks from overfitting by randomly blocking emissions from certain neurons, which is similar to adding noise in this process. Specifically, in RBM training a mask is generated and put on the hidden layer.

For instance, the hidden states are

$$h = \{h_1, h_2, \dots, h_n\} \quad (30)$$

and the dropout rate is r . Then a mask m is generated by

$$m_i = \begin{cases} 1, & u_i > r \\ 0, & u_i \leq r \end{cases} \quad (31)$$

where u_i is a sample from uniform distribution, and $i \in \{1, 2, \dots, n\}$. The emission of hidden layer would be

$$\tilde{h} = h * (m) \quad (32)$$

where $*$ denotes element-wise multiplication of two vectors. \tilde{h} , instead of h , is used to calculate visible states in RBM training.

Learning Rate Annealing

There are multiple methods to adapt the learning rate in gradient descent. If the learning rate is trivial, updates may tend to stuck in local minima and waste computation time. If it is too large, the updates may bound around minima and could not go deeper. In annealing, learning rate decay helps ensure the learning rate is not too large.

Suppose α is the learning rate. In exponential decay, the annealed rate is

$$\alpha_a = \alpha * e^{-kt}, \quad (33)$$

where t is the index of the current epoch, k is a customized coefficient. In divide decay, the annealed rate is

$$\alpha_a = \alpha / (1 + kt). \quad (34)$$

A more common method is step decay:

$$\alpha_a = \alpha * 0.5^{\lfloor t/5 \rfloor}. \quad (35)$$

where learning rate is reduced by half every five epochs. There are many other methods and the coefficients in the decay method may be tuned by comparing training results under various settings.

Momentum

With momentum ρ , the update step is

$$\Delta_{t+1}W = \rho * \Delta_tW - r \nabla W. \quad (36)$$

The update is a portion of previous update minus the gradient. The intuition behind it is that if the gradient has the same direction as previous update, the update will become larger. If the the gradient is in the different direction from previous update, the current update will not have the same direction as the gradient and its variance is reduced. In this way, time to converge is reduced.

Momentum is often applied with Annealing so that steps of updates will not be too large. A feasible scheme is to set

$$\rho = \begin{cases} 0.5, & t < 5 \\ 0.9, & t \geq 5 \end{cases} \quad (37)$$

where t is the index of the current epoch.

Weight Decay

In weight decay, a penalty term is added to the gradient as a regularizer. L_1 penalty p_1 is

$$p_1 = k \times \sum_{i,j} |W_{ij}|. \quad (38)$$

L_1 penalty causes many weights to become zero and a few weights to become large. L_2 penalty p_2 is

$$p_2 = k \times \sum_{i,j} W_{ij}^2. \quad (39)$$

L_2 penalty causes weights to become more even and smaller. The coefficient k is customized, sometimes 0.5 would work. Penalty must be, as well as the gradient, multiplied by the learning rate so that annealing will not change the model trained.

3.4 Classifier of Restricted Boltzmann Machine

To build a classifier on RBM is to train a classifier layer with softmax activation function on top of the hidden layer. Softmax activation function takes a vector $a = \{a_1, a_2, \dots, a_q\}$ as input, and outputs a vector $c = \{c_1, c_2, \dots, c_q\}$ with the same dimension, specifically

$$c_i = \frac{e^{a_i}}{\sum_{k=1}^q e^{a_k}}. \quad (40)$$

One of K scheme is here used to present the class distribution. If there are K classes in the training data, then the label of a sample in class i is expressed as a vector of length K with only the i th element as 1, the others as 0. For instance, if the training data has 5 classes, a sample in the forth class has the label

$$\{0, 0, 0, 1, 0\}. \quad (41)$$

For a training set with K classes, there should be K neurons in the classifier layer and the softmax activation emits a vector of length K . The maximum element in this emitted vector is located and its index is the label of this data sample. It could be found from the definition of the softmax activation that the elements of its emitted vector sums to 1 and it is actually the class distribution of prediction. Namely

$$c_i = P\{\text{The sample is in class } i\}. \quad (42)$$

This seems straightforward, however softmax activation takes input vector of length K whereas the trained RBM always has hundreds of units in its hidden layer. So the weight from the hidden layer to the classifier layer must be learned. If there are n_h hidden units in hidden layer and K classes, this weight should be a matrix of dimension $n_h \times K$.

The method of learning this weight is a one-step Backpropagation. Backpropagation is an algorithm in neural networks to compute the derivatives of classification error with respect to weights. Among several loss functions, cross entropy loss is a better choice for classification using softmax, which is justified by papers comparing various combinations of activation function and loss function. If the label is presented in One of K scheme as a vector t , and class distribution of prediction is c , cross entropy loss is

$$L = - \sum_{i=0}^K t_i \log(c_i). \quad (43)$$

Use chain rule to deduct the derivative as

$$\frac{\partial L}{\partial W_{ij}} = \sum_{p=1}^K \frac{\partial L}{\partial c_p} \frac{\partial c_p}{\partial W_{ij}} \quad (44)$$

and

$$\frac{\partial c_p}{\partial W_{ij}} = \sum_{q=1}^K \frac{\partial c_p}{\partial z_q} \frac{\partial z_q}{\partial W_{ij}} \quad (45)$$

where c is the output of softmax activation and z is its input. That is to say,

$$c_i = \frac{\exp(z_i)}{\sum_{p=1}^K \exp(z_p)}. \quad (46)$$

Furthermore

$$\frac{\partial L}{\partial c_p} = -\frac{t_p}{c_p} \quad (47)$$

$$\frac{\partial c_p}{\partial z_q} = \begin{cases} c_p(1 - c_p) & \text{if } p = q \\ -c_p \times c_q & \text{if } p \neq q \end{cases} \quad (48)$$

$$\frac{\partial z_q}{\partial W_{ij}} = \begin{cases} a_i^h & \text{if } q = j \\ 0 & \text{if } q \neq j \end{cases} \quad (49)$$

where a_i^h is the activation of hidden layer. Based on above equations, for combination of cross entropy loss and softmax activation

$$\frac{\partial L}{\partial W_{ij}} = a_i^h (c_j - t_j) \quad (50)$$

Expressed in row vectors it is

$$\nabla W = (a^h)^T \times (c - t) \quad (51)$$

where $(a^h)^T$ is the transpose of row vector a^h . This is the gradient used in batch learning to update the weight from hidden layer to classifier layer. Use the same update method in RBM training to build this classifier.

3.5 Implementation of RBM

My implementation of RBM is in the header file `rbm.hpp`. The RBM class stores information of one hidden layer and one activation layer. It has methods to train a learning or classifier layer. Here is a selected list of methods of class `rbm`:

I. void **dropout**(double i)

-Set dropout rate as input i.

II. void **doubleTrain**(dataInBatch &trainingSet, int numEpoch, int inLayer, ActivationType at = sigmoid_t, LossType lt = MSE, GDType gd = SGD, int numGibbs = 1)

-Train an RBM layer in Deep Boltzmann Machine, which is an undirected graph. This part will be explained in DBM section.

III. void **singleTrainBinary**(dataInBatch &trainingSet, int numEpoch, ActivationType at = sigmoid_t, LossType lt = MSE, GDType gd = SGD, int numGibbs = 1);

-Train an RBM layer with binary units in Deep Belief Networks and RBM.

IV. void **singleTrainLinear**(dataInBatch &trainingSet, int numEpoch, ActivationType at = sigmoid_t, LossType lt = MSE, GDType gd = SGD, int numGibbs = 1);

-Train an RBM layer with linear units.

V. void **singleClassifier**(dataInBatch &modelOut, dataInBatch &labelSet, int numEpoch, GDType gd = SGD);

-Build a classifier layer for RBM.

The model could be tested by running `runRBM.cpp`. It requires only OpenCV. For MNIST dataset, in which each data sample is a 28×28 image and data dimension is 784. First train an training layer consisting of one RBM with one hidden layer and one activation layer:

```
RBM rbm(784, 500, 0);
rbm.dropout(0.2);
rbm.singleTrainBinary(trainingData, 6);
dataInBatch modelOut = rbm.g_activation(trainingData);
```

The hidden layer has 500 units, and the index of this RBM is 0, which is used for multi-layer model. Then set the dropout rate as 0.2 and train it using training data with 6 epochs. After training, stack another RBM layer with softmax activation function:

```
RBM classifier(500, 10, 0);  
classifier.singleClassifier(modelOut, trainingLabel, 6);  
classificationError e = classifyRBM(rbm, classifier, testingData, testingLabel, sigmoid_t);
```

MNIST dataset has 10 classes, so the classifier trains a layer of 10 units using the emission from the training layer just acquired. Training label is used to compute the gradient in one-step backpropagation. The epoch is still 6. The function `classifyRBM` is defined in `rbm.hpp`. Error rate under this setting should be around 8%.

3.6 Summary

RBM is the foundation for several multi-layer models. It is crucial that this component is correctly implemented and fully understood. Some implementations train the classifier the same time as the training layer is built instead of training them in two steps. The advantage in separating these two is easiness in checking problems in the implementation.

4 Deep Neural Network

4.1 Construction of Deep Neural Network

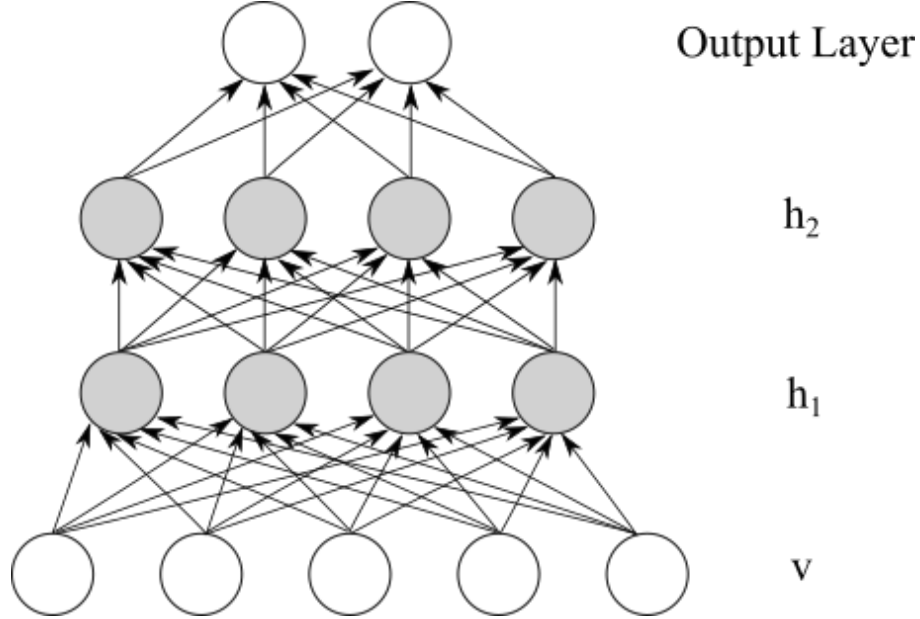


Figure 4.1: Deep Neural Network

A Deep Neural Network (DNN) is a neural network with multiple hidden layers. In neural networks, initialization of weights could greatly effect the training results. By pretraining, in which multiple Restricted Boltzmann Machines are trained to initialize parameters of each DNN layer, such dependence on weight initialization could be reduced. Also, backpropagation is a very time-consuming process. With pretraining, the epochs in bakckpropagation could be reduced to a limited number, which significantly reduces the time to train. Actually a solid performance of the model could be presented by pretraining alone. Figure 4.1 shows a DNN with two hidden layers.

Below shows construction of a Deep Neural Network for classification with pretraining:

I. Set the architecture of the model, specifically the size of the visible layer and the hidden layers, $n_0, n_1, n_2, \dots, n_N$. n_0 is the input dimension and input forms a visible layer with index 0. n_N equals to the number of classes in the label set.

II. Pretrain hidden layers:

for $i = 1$ to N :

1. Train an RBM with the following settings:

$$n_h^{(i)} = n_i \quad (52)$$

$$n_v^{(i)} = n_{i-1} \quad (53)$$

$$d_i = a_{i-1} \quad (54)$$

$$(55)$$

where

$$n_h^{(i)} = \text{Dimension of hidden layer of RBM trained for layer } i, \quad (56)$$

$$n_v^{(i)} = \text{Dimension of visible layer of RBM trained for layer } i, \quad (57)$$

$$d_i = \text{Input of RBM trained for layer } i, \quad (58)$$

$$a_{i-1} = \text{Activations of the } (i-1)\text{th layer of the model.} \quad (59)$$

2. Set

$$W_i = W_{RBM}, \quad (60)$$

$$b_i = b_{RBM}^h, \quad (61)$$

$$a_i = a_{RBM}. \quad (62)$$

That is to say, the weight of this RBM is the weight of the i th hidden layer of the model, the hidden bias of this RBM is the bias of the i th layer of the model, and a_i is the activation of this RBM.
end for.

III. Fine Tuning:

Use backpropagation to refine weights and biases of each layer. One epoch in backpropagation uses the whole training set for update once. Sometimes a dozen of epochs would suffice.

Classification with Deep Neural Network is similar to RBM. The emission of the last layer, namely its softmax activation, is the predicted class distribution. In the last layer, the index of the unit with the maximum probability is the label.

4.2 Fine Tuning of Deep Neural Network

As mentioned in the above section, fine tuning uses backpropagation, which is a common learning algorithm used in neural networks. Unlike one-step backpropagation used in training classifier of RBM, this one is a thorough one going through each layer. Backpropagation algorithm is:

I. Perform a pass through all layers of the network, computing total input of each unit $\{z^{(1)}, \dots, z^{(N)}\}$ and activations $\{a^{(1)}, \dots, a^{(N)}\}$ of each layer. $a^{(i)}$ is the row vector of the activation of layer i .

II. For the last layer, compute $\delta_i^{(N)}$ as

$$\delta_i^{(N)} = \frac{\partial L}{\partial z_i^{(N)}} \quad (63)$$

where L is the classification error. Acquire a row vector $\delta^{(N)}$.

III. For $l = N - 1, \dots, 1$, compute

$$\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)})^T \bullet g'(z^{(l)}) \quad (64)$$

where g is the activation function.

IV. Compute the gradients in each layer. For $l = N, \dots, 1$, compute

$$\nabla_{W^{(l)}} L = (a^{(l-1)})^T \delta^{(l)}, \quad (65)$$

$$\nabla_{b^{(l)}} L = \delta^{(l)}. \quad (66)$$

where $a^{(0)}$ is the input of the model.

V. Update the weights and biases of each layer using gradient descent.

In fine tuning, the same training should be used multiple times to refine the model parameters by repeating the above steps multiple times, or the affect may not be obvious.

4.3 Implementation of Deep Neural Network

The implementation of Deep Neural Network is in the header file `dnn.hpp`. It uses a vector of objects of class `RBMLayer` to store structure information. Here is a selected list of methods of class `dnn`:

I. void **addLayer**(RBMLayer &l)

Add a layer to the current model. This object of class `RBMLayer` should store information of layer size, weight, bias, etc. It could also be modified after added to the model.

II. void **setLayer**(std::vector<size_t> rbmSize)

Object of class `dnn` could automatically initialize random weights and biases of each layer by inputting a vector of layer sizes.

III. void **train**(dataInBatch &trainingData, size_t rbmEpoch, LossType l = MSE, ActivationType a = sigmoid_t)

This method trains all the layers without classifier. The structure of `dnn` should be initialized before calling this method.

IV. void **classifier**(dataInBatch &trainingData, dataInBatch &trainingLabel, size_t rbmEpoch, int preTrainOpt, LossType l = MSE, ActivationType a = sigmoid_t)
Build a Deep Neural Network with a classifier layer. This function contains pretraining option preTrainOpt. If preTrainOpt=1, pretrain each layer by training RBMs, else randomly initialize layer parameters without pretraining.

V. void **fineTuning**(dataInBatch &label, dataInBatch &inputSet, LossType l)
Fine tuning step using backpropagation.

VI. classificationError **classify**(dataInBatch &testingSet, dataInBatch &testinglabel);
Perform Classification. This is a method of class dnn because object of class dnn could store information of multiple layers. Object of class rbm could store information of only one layer so its classification uses a function instead of class method.

4.4 Summary

Construction of the Deep Neural Network is stacking multiple RBMs in the pretraining process and then performing fine tuning. Because the Deep Neural Network is a directed graph and each layer receives data input only from the adjacent layer below it, there is no extra inference step in training this model.

5 Deep Belief Network

5.1 Logic of Deep Belief Network

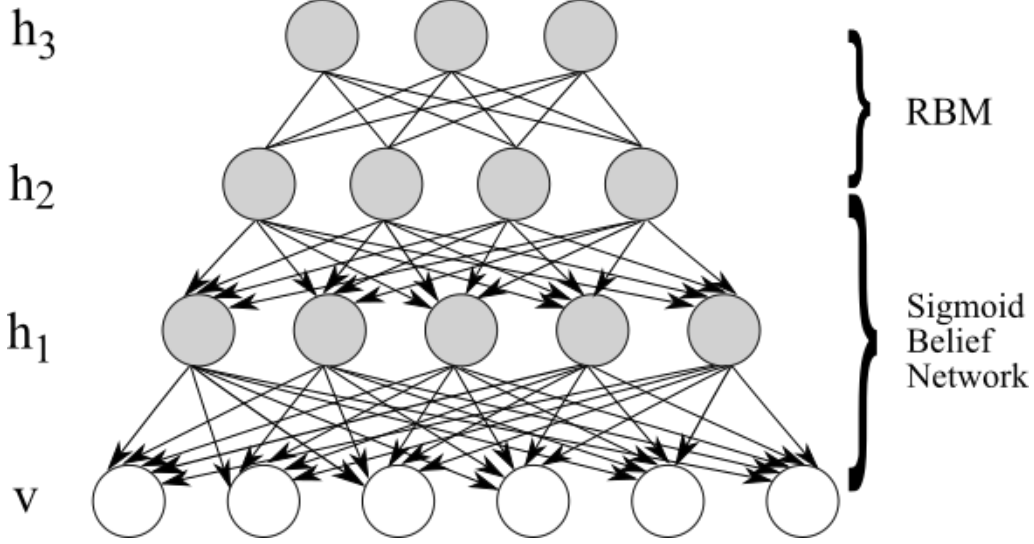


Figure 5.1: Deep Belief Network

A Deep Belief Network (DBN) is a hybrid of a Restricted Boltzmann Machine and a Sigmoid Belief Network. A desirable Deep Belief Network should maximize the likelihood $P(x)$ of the input x . Figure 5.1 shows a DBN.

The distribution over the visible layer (input data) and hidden layers is

$$P(v, h_1, \dots, h_N) = P(v|h_1) \times \left(\prod_{k=1}^{N-2} P(h_k|h_{k+1}) \right) \times P(h_{N-1}, h_N). \quad (67)$$

To prove this, express the distribution using chain rule:

$$P(v, h_1, \dots, h_N) = P(v|h_1, \dots, h_N) \times \left(\prod_{k=1}^{N-2} P(h_k|h_{k+1}, \dots, h_N) \right) \times P(h_{N-1}, h_N). \quad (68)$$

And in the belief network, each node is independent of its ancestors given its parent. So these hold:

$$P(v|h_1, \dots, h_N) = P(v|h_1), \quad (69)$$

$$P(h_k|h_{k+1}, \dots, h_N) = P(h_k|h_{k+1}). \quad (70)$$

So

$$P(v, h_1, \dots, h_N) = P(v|h_1) \times \left(\prod_{k=1}^{N-2} P(h_k|h_{k+1}) \right) \times P(h_{N-1}, h_N), \quad (71)$$

where $P(v|h_1) \times \left(\prod_{k=1}^{N-2} P(h_k|h_{k+1})\right)$ is the distribution over the Sigmoid Belief Network and $P(h_{N-1}, h_N)$ is the distribution over Restricted Boltzmann Machine.

For classification with label set Y , the distribution over the Deep Belief Network is

$$P(v, h_1, \dots, h_N, y) = P(v|h_1) \times \left(\prod_{k=1}^{N-2} P(h_k|h_{k+1})\right) \times P(h_{N-1}, h_N, y). \quad (72)$$

where $P(h_{N-1}, h_N, y)$ could be regarded as the distribution over a RBM which has labels y and the states h_{N-1} as the input.

The training of the Deep Belief Network is to stack pretrained RBMs like in pre-training the Deep Neural Network. However, since this is not a feedforward neural network. A different fine tuning method called Up-Down algorithm is used.

5.2 Training of Deep Belief Network

Training a Deep Belief Network is to find model parameters that maximize the likelihood of the input data. Because of the concavity of the logarithm function, the lower bound of the log likelihood of the input data x could be found:

$$\log P(x) = \log \left(\sum_h Q(h|x) \frac{P(x, h)}{Q(h|x)} \right) \geq \sum_h Q(h|x) \log \frac{P(x, h)}{Q(h|x)} \quad (73)$$

and

$$\sum_h Q(h|x) \log \frac{P(x, h)}{Q(h|x)} = \sum_h Q(h|x) \log(P(h|x) + P(x)) - \sum_h Q(h|x) \log Q(h|x) \quad (74)$$

where $Q(h|x)$ is an approximation to the true probability $P(h|x)$ of the model. If the weight between the first hidden layer and the visible layer and the biases of these two layers are found, it is clear that in the above equation only $P(h)$ is not fixed, and could be computed as

$$P(h) = \sum_{h^{(2)}} P(h, h^{(2)}) \quad (75)$$

which is based on the weight between the first hidden layer and the second hidden layer. This is why greedy layer-wise pretraining works. Furthermore,

$$\sum_h Q(h|x) \log(P(h|x) + P(x)) - \sum_h Q(h|x) \log Q(h|x) = \log P(x) - KL(Q(h|x) || P(h|x)). \quad (76)$$

So the learning is to maximize the lower bound of the log likelihood by minimizing the KL divergence between the true posteriors $P(h|x)$ and the approximations $Q(h|x)$.

This method is called variational learning. Unlike true posteriors, the approximations could be factorized

$$Q(h|x) = \prod_{i=1}^{n_h} Q(h_i|x). \quad (77)$$

The relationships between other hidden layers of the Sigmoid Belief Network are similar to this. And the weights of the pretrained RBMs are regarded as these approximations.

After pretraining is done. Use Up-Down algorithm as fine tuning, which is a combination of the training process of RBM, and an algorithm called Wake-Sleep algorithm, which is for learning of the Sigmoid Belief Network.

5.3 Classification of Deep Belief Network

In training the Restricted Boltzmann Machine, dropout is used to alleviate overfitting. This method reminds us that RBM could have the ability to predict missing values by adding noise in the training process. The Denoising Autoencoder works similarly.

After training the model, given the testing data, it should be able to compute the conditional probability as the prediction distribution. In training, $Q(h_{k+1}|h_k)$, the approximation to $P(h_{k+1}|h_k)$, has been established as the activation of the hidden layer of a pretrained RBM which has h_{k+1} as its hidden layer and h_k as its visible layer. For a DBN that has N hidden layers, by using this approximation the states of the $(N - 1)$ th hidden layer could be computed.

In a DBN that has been trained as a classifier. The prediction units, the $(N - 1)$ th and the N th hidden layer forms a RBM which has prediction units and the $(N - 1)$ th hidden layer as the RBM visible layer and the N th hidden layer as the RBM hidden layer. But only states of the $(N - 1)$ th hidden layer, which is a part of the RBM visible layer, has been directly computed. The derivation of prediction distribution is based on the ability of predicting missing values of RBM. So it is computed by going one step from the $(N - 1)$ th hidden layer to the N th hidden layer, setting values of the missing RBM visible units, which are prediction distribution units, as 0, and then going one more step from the N th hidden layer to the $(N - 1)$ th hidden layer and the distribution units. This transformation is illustrated in Figure 5.2.

If in pretraining the RBM between the N th and the $(N - 1)$ th hidden layer the dropout is used, the last pretrained RBM would be more capable of predicting missing distribution units.

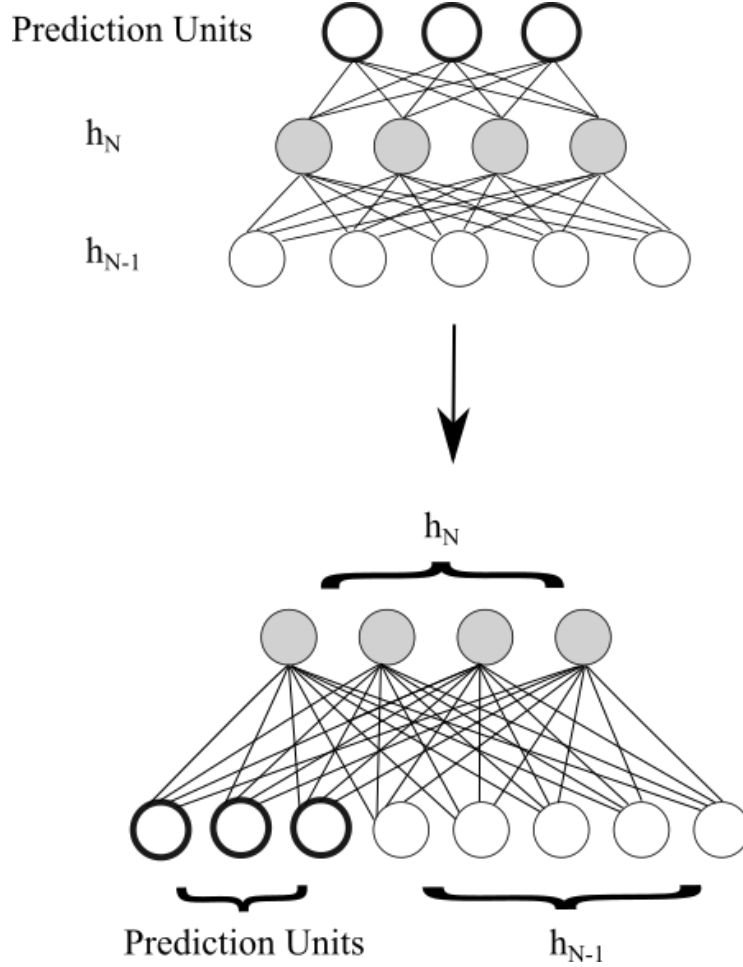


Figure 5.2: Classification of DBN

5.4 Implementation of Deep Belief Network

The implementation of Deep Belief Network is in the header file `dbn.hpp`. It uses a vector of objects of class `RBMLayer` to store structure information. Here is a selected list of methods of class `dnn`:

I. void **addLayer**(`RBMLayer &l`)

Add a layer to the current model. This object of class `RBMLayer` should store information of layer size, weight, bias, etc. It could also be modified after added to the model.

II. void **setLayer**(`std::vector<size_t> rbmSize`)

Object of class `dnn` could automatically initialize random weights and biases of each layer by inputting a vector of layer sizes.

III. void **train**(`dataInBatch &trainingData`, `size_t rbmEpoch`, `LossType l = MSE`, `ActivationType a = sigmoid_t`)

This method trains all the layers without classifier. The structure of dbn should be

initialized before calling this method.

IV. void **trainWithClassifier**(dataInBatch &trainingData, dataInBatch &trainingLabel, size_t rbmEpoch, LossType l = MSE, ActivationType a = sigmoid_t)

This method trains all the layers with classifier. The structure of dbn should be initialized before calling this method.

V. void **fineTuning**(dataInBatch &dataSet, dataInBatch &labelSet, int epoch)

Fine tuning step using Up-Down algorithm. This will call an upDown function.

VI. classificationError **classify**(dataInBatch &testingSet, dataInBatch &testinglabel);

Perform Classification.

5.5 Summary

It is easy to confuse the Deep Belief Network with the Deep Neural Network. Both of them stack pretrained RBMs in the training process. However, because these two models have distinct structures, their classification processes are different. In the Deep Neural Network, forward propagation gives the prediction distribution whereas in the Deep Belief Network the prediction distribution is computed by one more projection from the last hidden layer.

6 Denoising Autoencoder

6.1 Construction of Autoencoder

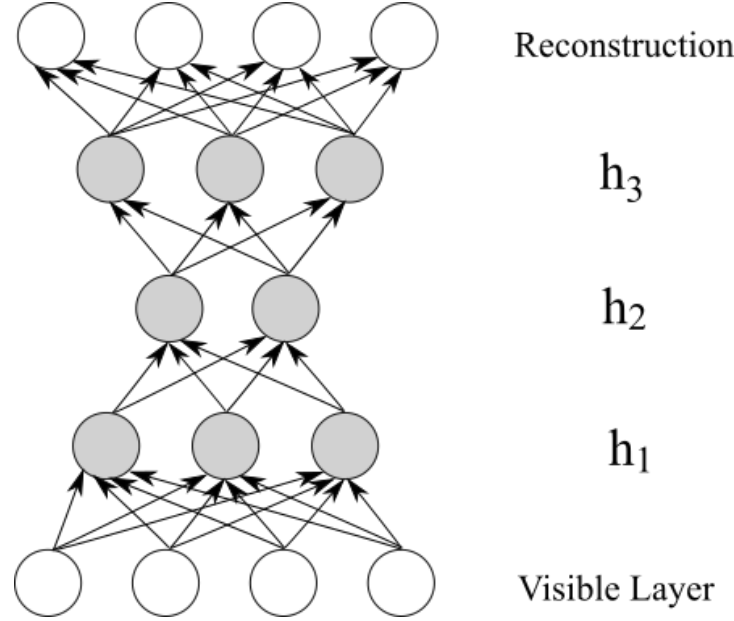


Figure 6.1: Autoencoder

Autoencoder is a type of neural network forming a directed graph. An autoencoder is a stack of layers of which half are fixed by pretraining RBMs and the other half are fixed by its symmetric structure. Its symmetricity states that for an autoencoder with N layers, the dimension of each layer is constrained by

$$n_i = n_{N-i} \quad 1 \leq i \leq N-1 \quad (78)$$

$$n_N = n_{input} \quad (79)$$

where n_i is the dimension of the i th layer, and n_{input} is the input dimension. The output and the input have the same dimension and it is expected that autoencoders could reconstruct the input. So the training of autoencoders is unsupervised learning because input is used as "labels" in fine tuning, and reconstruction errors could be used to access the model. Autoencoders could also be used as a part of supervised learning by adding a classification layer on top of it. Figure 6.1 shows an Autoencoder.

Below is how to construct an autoencoder AE:

I. Set the architecture of the model, specifically the size of each layer, n_1, n_2, \dots, n_k .

II. Fix the parameters of AE:

for $i = 1$ to $N/2$:

1. Train an RBM with the following settings:

$$n_h^{(i)} = n_i \quad (80)$$

$$n_v^{(i)} = \begin{cases} n_{input} & i = 1 \\ n_{i-1} & i > 1 \end{cases} \quad (81)$$

$$d_i = \begin{cases} a_{input} & i = 1 \\ a_{i-1} & i > 1 \end{cases} \quad (82)$$

where

$$n_h^{(i)} = \text{Dimension of hidden layer of RBM trained for layer } i, \quad (83)$$

$$n_v^{(i)} = \text{Dimension of visible layer of RBM trained for layer } i, \quad (84)$$

$$d_i = \text{Input of RBM trained for layer } i, \quad (85)$$

$$a_{i-1} = \text{Activations of the } (i-1)\text{th layer of autoencoder}, \quad (86)$$

$$a_{input} = \text{Input of the autoencoder AE}. \quad (87)$$

2. Set

$$W_i = W_{RBM}, \quad (88)$$

$$b_i = b_{RBM}^h, \quad (89)$$

$$a_i = a_{RBM}. \quad (90)$$

That is to say, the weight of this RBM is the weight of the i th layer of AE, the hidden bias of this RBM is the bias of the i th layer of AE, and a_i is the activation of this RBM.

end for.

for $i = N/2 + 1$ to N :

Set

$$W_i = W_{N-i}^T, \quad (91)$$

$$b_i = b_{N-i}. \quad (92)$$

$$. \quad (93)$$

end for.

III. Fine Tuning:

Use the input of AE as the labels and the output of AE as the predictions in Backpropagation. The Loss function is Mean Square Error.

6.2 Fine tuning of Autoencoder

Fine tuning of Autoencoder uses Backpropagation, which is:

- I. Perform a feedforward pass through all N layers of the autoencoder, computing net-ins $\{z^{(1)}, \dots, z^{(N)}\}$ and activations $\{a^{(1)}, \dots, a^{(N)}\}$ of each layer. $a^{(i)}$ is a row vector representing the activation of layer i .
- II. For the last layer, compute $\delta_i^{(N)}$ as

$$\delta_i^{(N)} = \frac{\partial L}{\partial z_i^{(N)}} \quad (94)$$

where L is the reconstruction error. This step acquire a row vector $\delta^{(N)}$.

- III. For $l = N - 1, \dots, 1$, compute

$$\delta^{(l)} = (\delta^{(l+1)} (W^{(l)})^T) \bullet g'(z^{(l)}) \quad (95)$$

where g is the activation function and here g' is element-wise performed.

- IV. Compute the gradients in each layer. For $l = N, \dots, 2$, compute

$$\nabla_{W^{(l)}} L = (a^{(l-1)})^T \delta^{(l)}, \quad (96)$$

$$\nabla_{b^{(l)}} L = \delta^{(l)}. \quad (97)$$

For $l = 1$, compute

$$\nabla_{W^{(1)}} L = (a^{(in)})^T \delta^{(1)}, \quad (98)$$

$$\nabla_{b^{(1)}} L = \delta^{(1)}. \quad (99)$$

where $a^{(in)}$ is the input data of the autoencoder.

- V. Update the weights and biases of each layer using gradient descent.

The reconstruction error of Autoencodr is

$$L = \frac{1}{2} \sum_{i=1}^{n_N} (a_i^{(N)} - a_i^{(in)})^2 \quad (100)$$

where

$$n_N = \text{Dimension of the Autoencoder output}, \quad (101)$$

$$a_i^{(N)} = \text{Value of the } i\text{th element of Autoencoder output}, \quad (102)$$

$$a_i^{(in)} = \text{Value of the } i\text{th element of Autoencoder input}. \quad (103)$$

Backpropagation of Autoencoder starts from computing $\delta_i^{(N)}$, and then find the δ of each layer below based on the derivative of activation function and δ of the upper layer. Here is how to compute these two values:

For sigmoid activation:

$$g'(t) = \frac{\partial(1 + e^{-t})^{-1}}{\partial t} = \frac{1}{1 + e^{-t}} \frac{e^{-t}}{1 + e^{-t}}. \quad (104)$$

That is to say

$$g'(z_i) = \frac{1}{1 + e^{-z_i}} \left(1 - \frac{1}{1 + e^{-z_i}}\right) = a_i(1 - a_i). \quad (105)$$

For $\delta_i^{(N)}$, use chain rule

$$\delta_i^{(N)} = \frac{\partial L}{\partial z_i^{(N)}} = \sum_{p=1}^{n_N} \frac{\partial L}{\partial a_p^{(N)}} \frac{\partial a_p^{(N)}}{\partial z_i^{(N)}} = \frac{\partial L}{\partial a_i^{(N)}} \frac{\partial a_i^{(N)}}{\partial z_i^{(N)}} = (a_i^{(N)} - a_i^{(in)}) \times a_i^{(N)}(1 - a_i^{(N)}). \quad (106)$$

So with sigmoid activation and square loss, fine tuning of autoencoder could be done based on only activations and weights of each layer. It could save memory by using only activations instead of both activations and net-ins of each layer even though the original formula of backpropagation uses $g'(z)$.

6.3 Denoising Autoencoder

The Denoising Autoencoder reconstructs the input from its corrupted version. So it could predict missing values and it is quite straightforward to observe its performance when the input is images. By putting a denoise mask on the input, the Autoencoders could be transformed to the Denoising Autoencoders. Here is how to make this transformation:

Firstly a denoise rate r is chosen, and the mask is constructed as follows:

$$m_i = \begin{cases} 1, & \text{if } U_i > r \\ 0, & \text{otherwise} \end{cases} \quad 1 \leq i \leq n_N \quad (107)$$

where U_i is the i th sample from uniform distribution, n_N is the size of the last layer, which is also the input dimension of the Autoencoder.

Secondly compute the corrupted input for the Autoencoder

$$a^{(inC)} = a^{(in)} \cdot m = \sum_{i=1}^{n_N} a_i^{(in)} m_i \quad (108)$$

Finally use $a^{(inC)}$ as the training data to compute the reconstruction and still use the uncorrupted input $a^{(in)}$ as labels in fine tuning.

Fine tuning in the Denoising Autoencoder makes more improvement of reconstruction than in the Autoencoder, and is crucial in the Denoising Autoencoder. Sometimes fine tuning in the Autoencoder may not have obvious improvement whereas in the Denoising Autoencoder the improvement is obvious.

6.4 Implementation of Denoising Autoencoder

The implementation of DAE is in the header file `autoencoder.hpp`. Its structure is similar to DBN:

I. void **addLayer**(RBMLayer &l)

Add a layer to current DAE. This object of class RBMLayer should store information of layer size, weight, bias, etc. It could also be modified after added to DAE.

II. void **setLayer**(std::vector<size_t> rbmSize)

Object of class AutoEncoder could automatically initialize random weights and biases of each layer by inputting a vector of layer sizes.

III. void **train**(dataInBatch &trainingData, size_t rbmEpoch, LossType l = MSE, ActivationType a = sigmoid_t)

This method trains all the layers without classifier. The structure of this AutoEncoder object should be initialized before calling this method.

IV. void **reconstruct**(dataInBatch &testingData)

Give the reconstruction of the testingData and stores the result in the model. This method should be called only after the model has been trained.

V. dataInBatch **g_reconstruction**()

Get the reconstruction.

VI. void **fineTuning**(dataInBatch &originSet, dataInBatch &inputSet, LossType l)

Fine tune DAE using backpropagation. Unlike DBN, LossType should be MSE instead of CrossEntropy.

VI. void **denoise**(double dr)

Set the denoise rate as dr. When model is in training, it will detect if denoise rate is set. So if this method is called before training, the Denoising Autoencoder will be trained automatically. Otherwise the Autoencoder will be trained.

6.5 Summary

Construction of the Denoising Autoencoder requires pretraining half of its layers, the other half is set by its symmetric structure. Fine tuning is crucial because it is the only step that uncorrupted data is used and the model knows what it should learn. The performance of Denoising Autoencoder is straightforward to assess because one could observe the reconstructed image of this model.

7 Deep Boltzmann Machine

7.1 Logic of Deep Boltzmann Machine

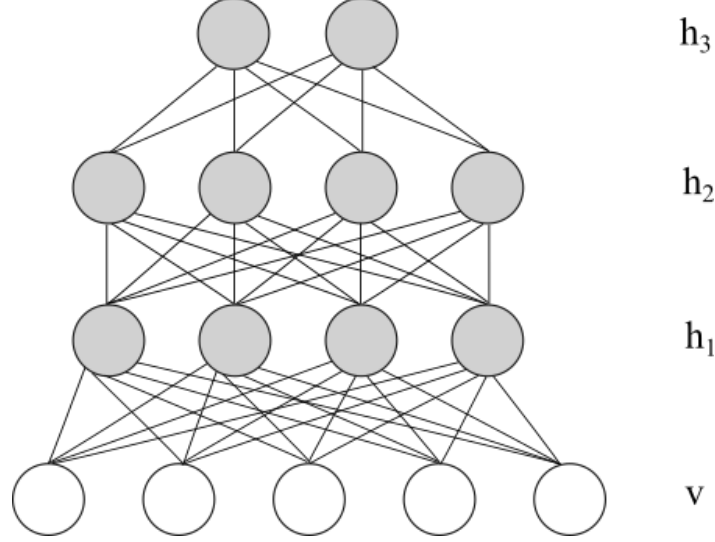


Figure 7.1: Deep Boltzmann Machine

A Deep Boltzmann Machine is a Markov Random Field consisting of multiple layers. Connections exist only between adjacent layers. Intuitively, it could incorporate top-down feedback when computing bottom-up approximations.

The energy function of a Deep Boltzmann Machine with N hidden layers is

$$E(v, h^{(1)}, \dots, h^{(N)}) = -v^T W^{(1)} h^{(1)} - (h^{(1)})^T W^{(2)} h^{(2)} - \dots - (h^{(N-1)})^T W^{(N)} h^{(N)} \quad (109)$$

where $W^{(i)}$ is the weight from the previous layer to the i th hidden layer.

A desirable Deep Boltzmann Machine maximizes the likelihood of the input data. The gradient of its log likelihood is

$$\frac{\partial \log P(v)}{\partial W^{(i)}} = \langle h^{(i-1)} (h^{(i)})^T \rangle_{data} - \langle h^{(i-1)} (h^{(i)})^T \rangle_{model}. \quad (110)$$

The learning uses variational learning as mentioned in the section of Deep Belief Network. More details are in pretraining and inference sections below.

7.2 Pretraining of Deep Boltzmann Machine

Because the Deep Boltzmann Machine is an undirected model, the last hidden layer receives input from only the adjacent layer below it, and the other hidden layers receives input from both direction. The visible layer receives input from only the first hidden layer. So when training Restricted Boltzmann Machines, the weights and

biases need to be adjusted for better approximations. The pretraining process is as below:

I. Set the architecture of the model, specifically the size of each layer, $n_0, n_1, n_2, \dots, n_N$. n_0 is the input dimension and input forms a visible layer with index 0.

II. Pretrain the first hidden layer:

Train an RBM, in which the weight from the visible layer to the hidden layer is $2W_1$ and the weight from the hidden layer to the visible layer is W_1^T . W_1 is the weight of the first hidden layer of DBM.

III. Pretrain intermediate hidden layers:

for $i = 1$ to $N - 1$:

1. Train an RBM with the following settings:

$$n_h^{(i)} = n_i \quad (111)$$

$$n_v^{(i)} = n_{i-1} \quad (112)$$

$$d_i = a_{i-1} \quad (113)$$

$$(114)$$

where

$$n_h^{(i)} = \text{Dimension of hidden layer of RBM trained for layer } i, \quad (115)$$

$$n_v^{(i)} = \text{Dimension of visible layer of RBM trained for layer } i, \quad (116)$$

$$d_i = \text{Input of RBM trained for layer } i, \quad (117)$$

$$a_{i-1} = \text{Activations of the } (i - 1)\text{th layer of autoencoder.} \quad (118)$$

2. Set

$$W_i = W_{RBM}/2, \quad (119)$$

$$b_i = b_{RBM}^h/2, \quad (120)$$

$$a_i = a_{RBM}. \quad (121)$$

The weight of this RBM is double the weight of the i th hidden layer of DBM, the hidden bias of this RBM is double the bias of the i th hidden layer of DBM, and a_i is the activation of this RBM.

end for.

IV. Pretrain the last hidden layer:

Train an RBM, in which the weight from the visible layer to the hidden layer is W_N and the weight from the hidden layer to the visible layer is $2W_N^T$. W_N is the weight of the last hidden layer of DBM.

Algorithm 2 Learning a Deep Boltzmann Machine.

```
1: Given: a training set of  $N$  binary data vectors  $\{\mathbf{v}\}_{n=1}^N$ ,  $M$ 
   (the number of Markov chains), and  $K$  (the number of mean-
   field steps).
2: // Pretraining:
3: Use Algorithm 1 to pretrain parameters
    $\theta_0 = \{W_0^1, W_0^2, W_0^3\}$  of a DBM.
4: Initialize the recognition model  $\theta_0^{rec} = \{R_0^1, R_0^2, R_0^3\}$  to the
   values of  $\theta_0$ .
5: Randomly initialize  $M$  sample particles:
    $\{\tilde{\mathbf{v}}_{0,1}, \tilde{\mathbf{h}}_{0,1}\}, \dots, \{\tilde{\mathbf{v}}_{0,M}, \tilde{\mathbf{h}}_{0,M}\}$ , where  $\tilde{\mathbf{h}} = \{\tilde{\mathbf{h}}^1, \tilde{\mathbf{h}}^2, \tilde{\mathbf{h}}^3\}$ .
6: for  $t = 0$  to  $T$  (number of iterations) do
7:   // Variational Inference:
8:   for each training example  $\mathbf{v}_n$ ,  $n = 1$  to  $N$  do
9:     In a single deterministic bottom-up pass, use the recog-
     nition model (Eqs. 8, 9, 10) to obtain a parameter vector
      $\boldsymbol{\nu}$  of the approximate factorial posterior  $Q^{rec}$ .
10:    Set  $\boldsymbol{\mu} = \boldsymbol{\nu}$  and run the mean-field updates (Eqs. 4, 5, 6)
    for  $K$  steps to obtain the mean-field approximate poste-
    rior  $Q^{MF}$ .
11:    Adjust the recognition parameters by taking a single gra-
    dient step in Eq. 11:
        
$$\theta_{t+1}^{rec} = \theta_t^{rec} + \alpha_t \frac{\partial \text{KL}(Q^{MF} || Q^{rec})}{\partial \theta^{rec}}$$

12:    Set  $\boldsymbol{\mu}_n = \boldsymbol{\mu}$ .
13:   end for
14:   // Stochastic Approximation:
15:   for each sample  $m = 1$  to  $M$  do
16:     Sample  $(\tilde{\mathbf{v}}_{t+1,m}, \tilde{\mathbf{h}}_{t+1,m})$  given  $(\tilde{\mathbf{v}}_{t,m}, \tilde{\mathbf{h}}_{t,m})$  by run-
     ning a Gibbs sampler.
17:   end for
18:   // Parameter Update:
19:   
$$W_{t+1}^1 = W_t^1 + \alpha_t \left( \frac{1}{N} \sum_{n=1}^N \mathbf{v}_n (\boldsymbol{\mu}_n^1)^\top - \frac{1}{M} \sum_{m=1}^M \tilde{\mathbf{v}}_{t+1,m} (\tilde{\mathbf{h}}_{t+1,m}^1)^\top \right)$$

20:   
$$W_{t+1}^2 = W_t^2 + \alpha_t \left( \frac{1}{N} \sum_{n=1}^N \boldsymbol{\mu}_n^1 (\boldsymbol{\mu}_n^2)^\top - \frac{1}{M} \sum_{m=1}^M \tilde{\mathbf{h}}_{t+1,m}^1 (\tilde{\mathbf{h}}_{t+1,m}^2)^\top \right)$$

21:   
$$W_{t+1}^3 = W_t^3 + \alpha_t \left( \frac{1}{N} \sum_{n=1}^N \boldsymbol{\mu}_n^2 (\boldsymbol{\mu}_n^3)^\top - \frac{1}{M} \sum_{m=1}^M \tilde{\mathbf{h}}_{t+1,m}^2 (\tilde{\mathbf{h}}_{t+1,m}^3)^\top \right)$$

22:   Decrease  $\alpha_t$ .
23: end for
```

Figure 7.2: Mean Field Inference of Deep Boltzmann Machine

7.3 Mean Field Inference

The mean field inference of the Deep Boltzmann Machine involves an iterative update of the approximations Q . Figure 7.2 shows the algorithm of mean field inference, which is from [5]. In sampling step, randomly initialize M sample particles. Each particle is a set of states of the model. To facilitate computation, M may be set as the number of batches. This is done by generating a random input, of which each element is a sample of uniform distribution. Then the random input is used to compute activations of each hidden layer. Also use the states of the first hidden layer to compute states of the visible layer. The states of all layers form a sample particle. This process uses the weight acquired in the pretraining step.

7.4 Implementation of Deep Boltzmann Machine

The implementation of DBM is in the header file dbm.hpp.

I. void **addLayer**(RBMLayer &l)

Add a layer to current DBM. This object of class RBMLayer should store information of layer size, weight, bias, etc. It could also be modified after added to DBM.

II. void **setLayer**(std::vector<size_t> rbmSize)

Object of class DBM could automatically initialize random weights and biases of each layer by inputting a vector of layer sizes.

III. void **train**(dataInBatch &Data, dataInBatch &label, size_t rbmEpoch, LossType l = MSE, ActivationType a = sigmoid_t)
This method trains DBM as classifier.

IV. void **fineTuning**(dataInBatch &data, dataInBatch &label)
Fine tune DBM using mean field inference.

V. void **classify**(dataInBatch &data, dataInBatch &label)
Classify the data with DBM. label set is used to compute accuracy.

8 Multimodal Learning Model

8.1 Deep Canonical Correlation Analysis

8.1.1 Canonical Correlation Analysis

Canonical Correlation Analysis is a method to find the relationship between two sets of variables. Specifically, suppose for two sets of variables X and Y , in which each column consists of samples of a variable, there are two projections

$$U = a^T X, \quad (122)$$

$$V = b^T Y. \quad (123)$$

The correlation between U and V is

$$\text{corr}(U, V) = \frac{\text{Cov}(U, V)}{\sqrt{\text{var}(U)\text{var}(V)}}. \quad (124)$$

The goal is to find the projections a and b that maximize this correlation. To ensure a unique solution, two conditions are added to the goal

$$\text{Var}(U) = 1, \quad (125)$$

$$\text{Var}(V) = 1. \quad (126)$$

Centering the data will not change the result but could facilitate computation:

$$x = X - I_X I_X^T X / n_X, \quad (127)$$

$$y = Y - I_Y I_Y^T Y / n_Y. \quad (128)$$

where I_X is an $n_X \times 1$ vector of ones, n_X is the number of rows of X , and n_Y is the number of rows of Y . And the covariance matrix of centered data like x is

$$\text{Cov}(x, x) = \text{Var}(x) = x^T x / n_x. \quad (129)$$

The solution to this problem suggests that a and b are respectively the eigenvectors of the following matrices

$$\text{Cov}(x, x)^{-1} \text{Cov}(x, y) \text{Cov}(y, y)^{-1} \text{Cov}(y, x), \quad (130)$$

$$\text{Cov}(y, y)^{-1} \text{Cov}(y, x) \text{Cov}(x, x)^{-1} \text{Cov}(x, y). \quad (131)$$

The i th eigenvectors of each matrix form the projections weights giving the i th largest correlation, and its corresponding eigenvalue is the square of its correlation value.

In computing, one could add each column of y to x and form a matrix $m = [x \ y]$. Then

$$\text{Var}(m) = \begin{bmatrix} \text{Cov}(x, x) & \text{Cov}(x, y) \\ \text{Cov}(y, x) & \text{Cov}(y, y) \end{bmatrix}. \quad (132)$$

8.1.2 Kernel Canonical Correlation Analysis

Kernel Canonical Correlation Analysis projects the data into a higher-dimensional feature space before linear projections using the mapping

$$\phi: x \rightarrow \phi(x). \quad (133)$$

In computation, kernel $K(x, y)$ is used, which is defined as

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \quad (134)$$

where $\langle a, b \rangle$ indicates inner product of a and b . The solution suggests that projection a is the top eigenvectors of the matrix

$$(K(X, X) + r_X I_X I_X^T)^{-1} K(Y, Y) (K(Y, Y) + r_Y I_Y I_Y^T)^{-1} K(X, X), \quad (135)$$

where r_X and r_Y are regularized terms. And b is given by

$$b = \frac{1}{\lambda} (K(Y, Y) + r_Y I_Y I_Y^T)^{-1} K(X, X) a \quad (136)$$

where λ is the corresponding eigenvalue of a .

8.1.3 Deep Correlation Analysis

In Deep Correlation Analysis (DCCA), A layer of KCCA is added on top of two separately trained Deep Neural Networks, of which each learns the data of one modal. Figure 8.1 shows a example of DCCA. With KCCA on the top, the model could learn the correlation between data of two modals.

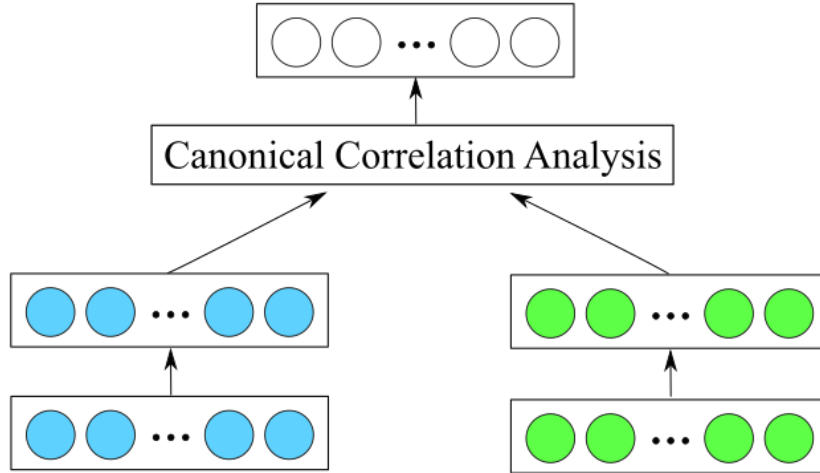


Figure 8.1: Deep Canonical Correlation Analysis

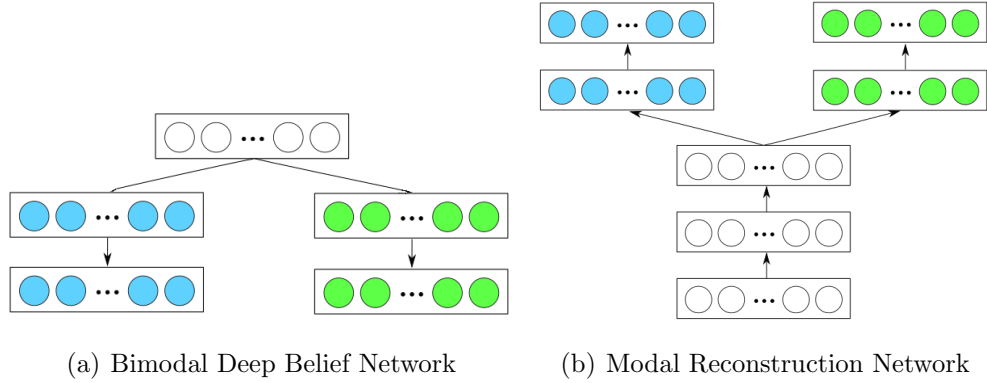


Figure 8.2: Modal Prediction

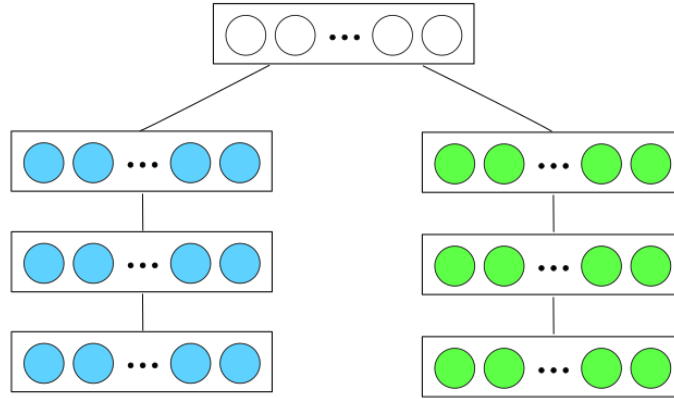


Figure 8.3: MRF Multimodal Learning Model

8.2 Modal Prediction and Transformation

It is possible to make lower-dimension representation of two modals by building a Bimodal Deep Belief Network as in Figure 8.2(a), in which blue nodes represent data from one modal and green nodes represent data from the other modal. In this process the recognition weights which are used in bottom-up computation and the generative weights which are used in top-down computation should be learned. If the model is trained with tied weights, half of the memory space could be saved since transpose of weight matrix would transform recognition weights to generative weights. The weights of this model could be used to reconstruct data of two modals as in Figure 8.2(b).

Another option is to build a Markov Random Field multimodal learning model by combining two Deep Boltzmann Machines. Figure 8.3 shows such a model. This model is constructed by first build two DBMs, each is trained on data of one modal. Then Train an RBM on top of these two DBMs.

Prediction of one data from one modal could be done by first training a Bimodal

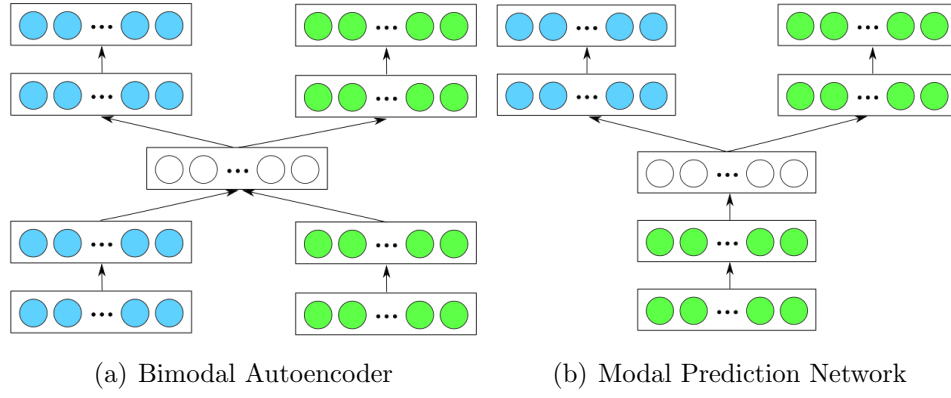


Figure 8.4: Modal Prediction

Autoencoder in Figure 8.4(a) and then use the modal prediction network in Figure(b) to predict data from two modals. It would be necessary to add noise possibly in data corruption when training the Bimodal Autoencoder so that it has more power in reconstruction.

9 Library Structure

9.1 Data Reading

9.1.1 MNIST

MNIST is a selected set of samples from NIST data set. It has one training data set, one training label set, one testing data set, and one testing label set. The training set has 60,000 samples and the testing set has 10,000 samples. Each sample data is a 28×28 grey image which is a handwritten integer between 0 and 9. It has 10 classes, so the label is between 0 (inclusive) and 9 (inclusive).

The data is stored in big-endian format. The content of data should be read as unsigned characters. Header file `readMNIST.hpp` provides functions to read this data set.

I. `cv::Mat imread_mnist_image(const char* path)`

Read data of MNIST. Each row is a sample.

II. `cv::Mat imread_mnist_label(const char* path)`

Read labels of MNIST. Each row is a number indicating class of that sample.

9.1.2 CIFAR

The CIFAR-10 data set consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 5 batches of training images and 1 batch of test images, each consists of 10000 images.

In CIFAR-10 data set, each sample consists of a number indicating its class and the values of the image pixels. Put five training data batches and one testing data batch of CIFAR-10 in the same folder named "data". The following function in the header file `readCIFAR.hpp` reads them to four OpenCV matrices:

```
void imread_cifar(Mat &trainingData, Mat &trainingLabel, Mat &testingData, Mat &testingLabel)
```

Each row of the read OpenCV matrices consists of the label of the data of a sample.

9.1.3 XRMB

As stated in its explanation, Wisconsin X-ray Microbeam Database (XRMB) data used in "Deep Canonical Correlation Analysis" is processed by Galen Andrew, Raman Arora, etc. The dimension of the MFCC data is 273 and the dimension of the XRMB data is 112. There are label files of fold 0 that provides the phone labels. The data is stored in double-precision format.

The following function in the header file `readXRMB.hpp` reads them:

```
cv::Mat imread_XRMB_data(const char* path, int inputDim)
```

```
indexLabel imread_XRMB_label(const char* path)
```

When reading MFCC files, `inputDim = 273`. When reading XRMB files, `inputDim = 112`. The label is between 0 (inclusive) and 39 (inclusive). The data is stored sample by sample.

9.1.4 AvLetters

AvLetters is the data set recording audio data and video data of different people uttering letters. The dimension of the audio data is 23 and the dimension of the video data is 60×80 . The data is stored in single-precision big-endian format. Each file is the data of a person uttering a certain letter. For instance, the file `A1_Anya.mfcc` contains the audio data of the person named Anya uttering letter "A".

The following function in the header file `readAvLetters.hpp` reads audio data:

```
cv::Mat imread_avletters_mfcc(const char* path)
```

The output is an OpenCV matrix, of which each row contains data of a sample. The original video data is in MATLAB file format. The header file `readMat.hpp` contains the function

```
cv::Mat matRead(const char* fileName, const char* variableName, const char* saveName)
```

to read the mat file and at the same time save it as binary file named as the argument "saveName". This header file uses the MATLAB/c++ interface provided by MATLAB and requires an environment setting, which is contained as comments in the header file. There are some problems running the libraries in this interface together with OpenCV. So use this function to transform all MATLAB files to binary files before training models and then read the transformed binary files would be better. The header file `readDat.hpp` provide the function to read the transformed binary files:

```
cv::Mat readBinary(const char* file, int rowSize, int colSize)
```

The output is an OpenCV matrix, of which each row contains data of a sample.

9.1.5 Data Processing

Header file `processData.hpp` stores functions processing data.

data **oneOfK**(indexLabel l, int labelDim)
Transfer index label to one of k expression.

dataInBatch **corruptImage**(dataInBatch input, double denoiseRate)
Give corrupted data in batches.

std::vector<dataInBatch> **dataProcess**(dataCollection& reading, int numBatch)
Transfer reading of a data set into data batches.

dataCollection **shuffleByRow**(dataCollection& m)
Shuffle the read data

cv::Mat **denoiseMask**(size_t rowSize, size_t colSize, double rate)
Generate a mask to corrupt data

9.2 Computation and Utilities

activation.hpp includes multiple activation functions, such as sigmoid, tanh, relu, leaky_relu, softmax. Each activation function is paired with a function that computes its derivatives to facilitate computation in backpropagation.

cca.hpp includes functions computing CCA and KCCA.

upDown.hpp includes fine tuning method used for DBN.

gd.hpp includes functions for adaptive gradient descent and stochastic gradient descent, and a function to anneal the learning rate in which three types of annealing methods are provided.

inference.hpp includes the mean field inference implementation used by DBM.

upDown.hpp includes the up-down fine tuning algorithm implementation used by DBN.

kernel.hpp includes multiple kernel functions used by KCCA.

loss.hpp includes computation of loss functions. MSE, absolute loss, cross entropy, and binary loss are provided together with the functions to compute their derivatives.

matrix.hpp includes some OpenCV matrix manipulation functions.

loadData.hpp contains function to test data loading by visualization.

visualization.hpp contains function of visualization.

9.3 Models and Running

Table 1 shows the header files and files contains main function to test each model. Besides these, modalPrediction.cpp contains implementation of a modal prediction model. And modalDBM.cpp contains the implementation of a multimodal learning model based on Deep Boltzmann Machines.

Model	RBM	DNN	DBN	DAE/AE	DBM	DCCA
Header(.hpp)	rbm	dnn	dbn	autoencoder	dbm	dcca
Main(.cpp)	runRBM	runDNN	runDBN	runDAE	runDBM	runDCCA

Table 1: Header files and main functions of models

10 Performance

10.1 Restricted Boltzmann Machine

The main function to run Restricted Boltzmann Machine is in `runRBM.cpp`. It uses RBM for classification of MNIST data set. On MNIST, use 60,000 samples for training and 10,000 samples for testing. With hidden layer of size 500. The classification error rate is 0.0707 (Accuracy 92.93%). Multiple deep learning libraries give similar results.

10.2 Deep Neural Network

The main function to run Deep Neural Network is in `runDNN.cpp`. Usually it takes a large number of epochs in backpropagation in training. However this number is greatly reduced by pretraining.

For testing, a DNN with hidden layers of size 500, 300, 200 respectively is constructed. On MNIST data set, pretraining alone would give out the error rate of 0.093 (Accuracy 90.7%). In backpropagation, each epoch goes through once the whole training data for update. One epoch in backpropagation gives out classification error rate of 0.0858 (Accuracy 91.42%). Ten epochs in backpropagation gives out classification error rate of 0.0288 (Accuracy 97.12%). Learning rate is 0.01 in fine tuning. The result is similar to the performance of running DNN on MNIST with MEDAL.

10.3 Denoising Autoencoder

The main function to run Denoising Autoencoder is in `runDAE.cpp`. For testing, a Denoising Autoencoder with hidden layers of size 500, 300, 500 is constructed. Figure 10.1 shows the Denoising Autoencoder reconstruction of the corrupted testing set using model trained by training set, in which upper six lines are reconstructions and the lower six lines are uncorrupted input. As a comparison, Figure 10.2 shows the Autoencoder reconstruction of the uncorrupted testing set. Fine tuning in the Denoising Autoencoder gives more improvement of performance than in the Autoencoder.

Each epoch goes through once the whole training data for update. On MNIST, run 10 epochs in fine tuning, the reconstruction error computed by MSE in the Denoising Autoencoder are: Average error without fine tuning 6686.69. Average error after fine tuning 3256.34. The reconstruction error computed by MSE in the Autoencoder are: Average error without fine tuning 4463.24. Average error after fine tuning 3182.69.

So after sufficient fine tuning, the reconstruction of Denoising Autoencoder is similar to the reconstruction from the uncorrupted images. The reconstruction visualization is comparable to published result.



Figure 10.1: Reconstruction of DAE and AE on MNIST

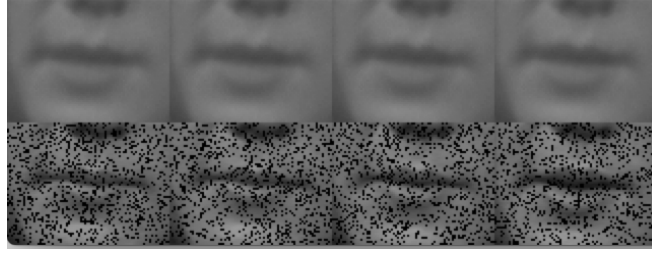


Figure 10.2: Denoising Avletter audio data

Another test is made with AvLetters data set, which is in `runDAE_letter.cpp`. Figure 10.2 shows denosing avletter audio data. It uses the audio data of a person pronouncing "A", "B", "C", "D", "E", "F", "G" to train the model and reconstructs the data of person pronouncing "H" and "I".

10.4 Deep Belief Network

The main function to run the Deep Belief Network is in `runDBN.cpp`. For testing, a Deep Belief Network with hidden layers of sizes 500, 300 is constructed. The classification error rate on MNIST without fine tuning is 0.0883 (Accuracy 91.17%). With the Up-down fine tuning method the classification error rate could be reduced to 0.0695 (Accuracy 93.05%). [6] indicates the best performance of DBN on MNIST could achieve the error rate of 1.25%. There are multiple parameters in the training process could affect the result, such as learning rate and tricks in gradient descent.

This possibly affect the result.

10.5 Deep Boltzmann Machine

The main function to run the Deep Boltzmann Machine is in `runDBM.cpp`. For testing, a Deep Boltzmann Machine with hidden layers of sizes 500, 500 is constructed. The error rate on MNIST without fine tuning is 0.0937 (Accuracy 90.63%). Mean Field inference improves the accuracy to 93.47%. Again multiple parameters could affect the result. [5] indicates that the accuracy on MNIST could achieve test error of 0.95%. Its source code uses Conjugate Gradient optimization, which is not implemented in this library. This possibly causes the difference.

10.6 Deep Canonical Correlation Analysis

The main function to run DCCA is in `runDCCA.cpp`. The testing data set is CIFAR-10. Each image is divided as the left and the right halves as data from two views. Then two networks trained on two views are built. Each view of data is of dimension 512. Each network gives the output of dimension 20. The DCCA could quickly find the solution in which the sum of the top 20 correlations is 10.8756. Without network below CCA layer, the solution could not be computed in a reasonable time.

A test on AvLetters was also done. Unfortunately the correlation given is close to 0. This suggests that there is very weak linear dependence between data of two modals. So it should be another form of dependence existing.

10.7 Modal Prediction

Modal prediction implementation is in the `modal_prediction.cpp`. It trains the modal using audio and video data of uttering letter from "A" to "G" and then uses the audio data of letter "H" and "I" to predict the video data of "H" and "I". The reconstruction error is 10.46%.

References

- [1] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
- [2] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.
- [3] David R Hardoon, Sandor Szedmak, and John Shawe-Taylor. Canonical correlation analysis: An overview with application to learning methods. *Neural computation*, 16(12):2639–2664, 2004.
- [4] Ruslan Salakhutdinov and Geoffrey E Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455, 2009.
- [5] Ruslan Salakhutdinov and Hugo Larochelle. Efficient learning of deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 693–700, 2010.
- [6] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- [7] Galen Andrew, Raman Arora, Jeff Bilmes, and Karen Livescu. Deep canonical correlation analysis. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1247–1255, 2013.
- [8] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.
- [9] Nicolas Le Roux and Yoshua Bengio. Representational power of restricted boltzmann machines and deep belief networks. *Neural computation*, 20(6):1631–1649, 2008.
- [10] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.
- [11] University of Montreal LISA lab. Deep learning tutorial. <http://deeplearning.net/tutorial/deeplearning.pdf>. Accessed: 2015-11-18.
- [12] Lawrence K Saul, Tommi Jaakkola, and Michael I Jordan. Mean field theory for sigmoid belief networks. *Journal of artificial intelligence research*, 4(1):61–76, 1996.
- [13] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

- [14] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543. ACM, 2008.
- [15] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.