Java Deep Learning Library-DLcty

Tianyi Chen, PhD@Johns Hopkins University, tchen59@jhu.edu

1 A new Java Deep Learning Library-DLcty

1.1 Abstract

Java as a popular programming language, somehow due to the lack of powerful mathematical package support , like numpy in python, doesn't have a well-known deep learning library like theano, caffe. To fix it up, the author create a this Java Deep learning library named it as DLcty. The documentation is attached in this writeup with some numerical results. This library provides the following interfaces of deep learning:

1. Binary case Restricted Boltazmann Machine(RBM).
2. Binary case Deep Belief Network (DBN)
3. Multi-layer Perceptron(MLP), with/without binary DBN pre-train.

1.2 Preliminaray

1.To allocate methods built in DLcty, DLcty.jar file should be attached as external library in Java project.
2. Add external jar file: math3 from Apache.

1.3 LoadData

All of Input data should be converted as RealMatrix instances from math3 library.

```
DataStream.load(String fileX, String Y)
```

DataStream.load method built in DLcty can load data which are in the same format as Demo data.
For example:

```
String inputX="german_numer01_X.out";
String inputY="german_numer01_Y.out";

Map<String, RealMatrix> infoMap=DataStream.loadData(inputX, inputY);

RealMatrix X=infoMap.get("X");
RealMatrix Y=infoMap.get("Y");
```

where X is the Data matrix, the default size of X is m by n, m is number of samples, n is number of features, Y is a column matrix saving labels.

1.4 Restricted Boltazmann Machine(RBM)

The theoritical knowledge of RBM can be found in the lecture note I scribed. In DLcty, I implement contrasive divergence to train binary RBM, while the persistent CD is expected to implement in this well-organized library easily.
To run RBM on input data, the following steps should be done:

Step 1: create an instance of RBM class.

```
int num_features=X.getColumnDimension();
int num_samples=X.getRowDimension();
//RBM(int num_visible, int num_hidden)
RBM rbm=new RBM(num_features, num_features/2);
```

Step 2: set parameters, including Contrasive-Divergence K, learning rate α, training epoch, and batch size. For examples:

```
int K=50;
double alpha=0.1;
rbm.setK(K);
rbm.setAlpha(alpha);
int training_epoches=100;
int batch_size=111;
```

Step 3: train RBM:

```
// train RBM
for(int t=0;t<training_epoches;t++){
        Random randomGenerator = new Random();

        // select a batch of data randomly
        RealMatrix inputData=otherTools.stochasticSubmatrix(X, batch_size, randomGenerator);

        // update parameters of rbm1 with SGD solver
        rbm.updateParams(inputData);

        // calculate reconstruction loss value
        double loss=rbm.CrossEntropy(inputData);
        System.out.println("loss value:"+loss);

}
```

To obtain the weights of edges of RBM, and bias of hidden layer, visible layer. Run the following command:

```
// get weights of edges
RealMatrix weights=rbm.getW();

// get bias of hidden layer
RealMatrix hbias=rbm.getHbias();

// get bias of visible layer
RealMatrix vbias=rbm.getVbias();
```

The output on console should be like

```
training epoch:0 loss value:-0.7586267301050253
training epoch:1 loss value:-0.42361143804903734
training epoch:2 loss value:-0.5062772086049139
training epoch:3 loss value:-0.39472787021550987
training epoch:4 loss value:-0.3386319033418799
training epoch:5 loss value:-0.2885894840596366
training epoch:6 loss value:-0.1784915414427655
training epoch:7 loss value:-0.2563098524576439
training epoch:8 loss value:-0.20920941463328502
training epoch:9 loss value:-0.19087708883369667
training epoch:10 loss value:-0.19953132725629807
```

where the loss value is reconstruction loss using cross-entropy loss.
Using german-numer-binary dataset, Figure.1 shows observe even K=1 can let loss value descent fast

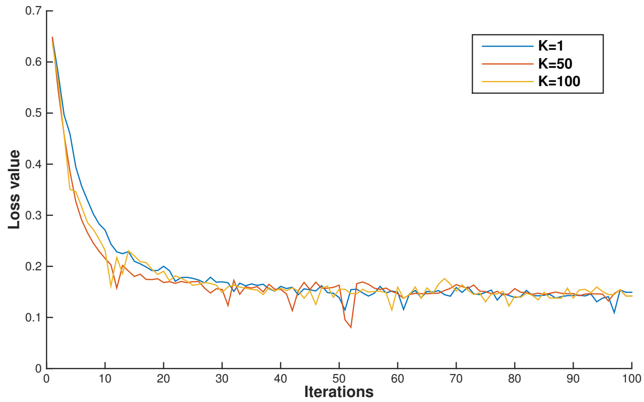

Figure 1: Loss value versus iteration under different K of Contrasive Divergence

and sufficiently.

1.5 DBN

Deep belief network is stacked by multiple RBMs. To train DBN, we should adopt the following strategy:

1. Initialize multiple RBM layers.
2. Train the first RBM layer.
3. Forward sample hidden layer of the first RBM layer as the visible layer of the second RBM layer.
4. Fix the parameters of first RBM layer, train the second RBM layer.
5. Repeat the above steps until all RBM layers have been trained.

To run DBN on input data, we need to do the following steps:

Step 1: Set hidden layer size.

```
// three hidden layers, the first hidden layer contains 10 neurons, the second hidden layer
// contains 100 neurons, the third one has 50 units
int[] hiddensizes={10,100,50};
```

Step 2: Set various parameters, including batch size, CD-k, pretrain epoch, learning rate α.

```
int batch_size=111;
int K=10;
int pre_training_epoches=10;
double alpha=0.1;
```

Step 3: create an instance of DBN class

```
//DBN(int inputSize, int[] hiddenSizes, int outputSize, int K, int pretraining_epoch, double alpha)
DBN dbn=new DBN(num_features,hiddensizes,2,K,pre_training_epoches,alpha);
```

Step 4: train DBN

```
dbn.pretraining(X, batch_size);
```

The output on console should be like:

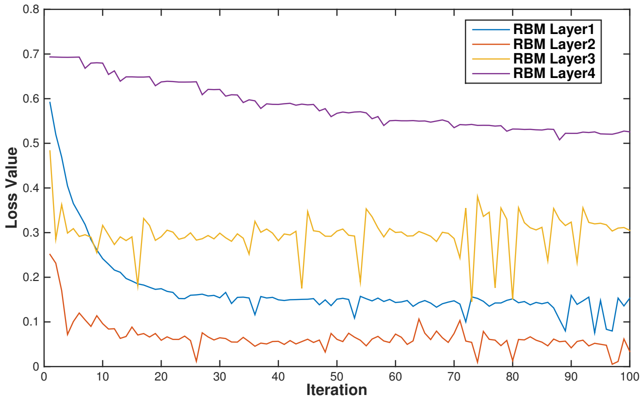

Figure 2: Loss value versus iteration of different RBM layers

```
RBM layer 0:
pretraining_epoch:0 loss value:-0.881350017834405
pretraining_epoch:1 loss value:-0.8390507134244802
pretraining_epoch:2 loss value:-0.7762359475699605
pretraining_epoch:3 loss value:-0.7302838685660593
pretraining_epoch:4 loss value:-0.654893559361064
pretraining_epoch:5 loss value:-0.5922925681225922
pretraining_epoch:6 loss value:-0.7269764976888777
pretraining_epoch:7 loss value:-0.49953019658958897
pretraining_epoch:8 loss value:-0.44569765990116317
pretraining_epoch:9 loss value:-0.3966535711223775
RBM layer 1:
pretraining_epoch:0 loss value:-0.6499980083516476
pretraining_epoch:1 loss value:-0.35198477897381214
pretraining_epoch:2 loss value:-0.2386644401683641
pretraining_epoch:3 loss value:-0.16297411827225888
pretraining_epoch:4 loss value:-0.1795433610478966
pretraining_epoch:5 loss value:-0.1761257791313093
pretraining_epoch:6 loss value:-0.15042153063761374
pretraining_epoch:7 loss value:-0.14418155386811432
pretraining_epoch:8 loss value:-0.13350506524899258
pretraining_epoch:9 loss value:-0.13071603138811735
RBM layer 2:
pretraining_epoch:0 loss value:-0.7596952344886028
pretraining_epoch:1 loss value:-0.6522887226764831
pretraining_epoch:2 loss value:-0.6025875190341131
pretraining_epoch:3 loss value:-0.5659029584113164
pretraining_epoch:4 loss value:-0.5373838809163487
pretraining_epoch:5 loss value:-0.527928251170187
pretraining_epoch:6 loss value:-0.5386322026886582
pretraining_epoch:7 loss value:-0.5364908634840797
pretraining_epoch:8 loss value:-0.343428541703665
pretraining_epoch:9 loss value:-0.5728706249450987
RBM layer 3:
pretraining_epoch:0 loss value:-0.7681064220297842
pretraining_epoch:1 loss value:-0.6657686762639301
pretraining_epoch:2 loss value:-0.6537710337535471
pretraining_epoch:3 loss value:-0.6921091214259893
pretraining_epoch:4 loss value:-0.6890022501853523
pretraining_epoch:5 loss value:-0.6845761178581661
pretraining_epoch:6 loss value:-0.6831882728400412
pretraining_epoch:7 loss value:-0.6750384463330193
pretraining_epoch:8 loss value:-0.6749699540065863
pretraining_epoch:9 loss value:-0.6699778637401456
```

To obtain $i-th$ RBM layer, we can

```
RBM rbm=dbn.getRbmlayers().get(i);
```

Then we can obtain the weights, bias of RBM layer as the RBM instance in last part. Figure.2 shows the loss value of each RBM layer with SGD(stochastic gradient descent) decrease.

1.6 MLP

Multi-Layer Perceptron is a widely used neural network. Here, I just skip the theoritical part of MLP since it can be found in the lecture note or other deep learning documentations.

In DLcty, an interface can support DBN pretrain. Under DBN pretrain, we will pretrain DBN at first, then use the same layer size of DBN as the network size of MLP, and use weights, bias trained from DBN as the initial weights, bias of MLP. Then train MLP.

To run MLP, we should do the following steps
Step 1: create MLP instance.

```
MLP mlp=new MLP(X,num_features,hiddensizes,outputneurons,needPretrain,lr,batch_size,pretrainingepoch,alpha,K);
```

1. $needPretrain$ is a boolean value. $needPretrain = true$ will allocate DBN to pretrain, $needPretrain = false$ will initialize the weights and bias of MLP randomly from Gaussian distribution.
2. lr is the learning rate of training MLP
3. $alpha$ is the learning rate of DBN pretrain
4. K is the CD-K in DBN pretrain.

For example,

```
int num_features=X.getColumnDimension();
int num_samples=X.getRowDimension();
int training_epoches=1000;
int pretrainingepoch=10;
int batch_size=23;

double lr=0.3; // learning rate of MLP
double alpha=0.1; // learning rate of DBN pretraining
int K=10;

int[] hiddensizes={28,14,14,14}; // hidden layer sizes.
MLP mlp=new MLP(X,num_features,hiddensizes,2,false,lr,batch_size,pretrainingepoch,alpha,K);
```

Step 2: train MLP

```
mlp.train(X, Y, batch_size, training_epoches);
```

Step 3: test MLP

```
RealMatrix test_Y=mlp.convertLabelToVector(testY,2);
mlp.predict(testX, test_Y);
```

To get the weights of each layer in MLP, $getWeights()$ will return a HashMap with layer index as key, and weights as corresponding value.

```
Map weightsMap=mlp.getWeights();
```

To get the bias of each layer in MLP, $getBias()$ will return a HashMap with layer index as key, and bias as corresponding value.

```
Map biasMap=mlp.getBias();
```

It will output the accuracy of predicting, e.g. Figure.3.