deep neural network programming exercises

Generated by Doxygen 1.8.13

Contents

Chapter 1

Neural network exercises

Learning neural networks by exercises.

Prerequisties

Libraries:

- icc,mkl

Installing

sudo apt-get install g++ g++-multilib build-essential
install icc,mkl
make test_mlr test_ffnn

Datasets

datasets/train.csv, datasets/test.csv

the training/validation data sets uses the MNIST datasets in csv format:
`https://www.kaggle.com/c/digit-recognizer/data`

Running the test

multi-classes logistic regression

./test_mlr

```
Cost of train/validation at epoch    0 :  0.45934263 0.34229076
Cost of train/validation at epoch   10 :  0.25960219 0.28029829
Cost of train/validation at epoch   20 :  0.24500903 0.27580458
Cost of train/validation at epoch   30 :  0.23828110 0.27294943
validation set accuracy:0.925119
Enter image id (0<=id<8400):~
```

```
0
----------------------------



             *********
           ***********
           ************
           ************
                    ***
                    ***
                    ***
                   ****
                  *****
                 *****
                *****
      **************
    ***************
   ****************
   *****************
   ******************
   **********     ****
   ********       *****
    ****           ****
                    ***



 ----------------------------
Predicted: 2
Actual: 2
```

feed-forward neural network with two hidden-layers

Initialize the hidden layer dimensions with {256,128}
activation types with {"ReLU","sigmoid"}
keep probabilities in dropout regularization with {0.9,0.6,0.8}:
./test_ffnn -n 100

```
Cost of train/validation at epoch    0 :  0.88942140 0.34440878
Cost of train/validation at epoch   10 :  0.16244245 0.10217448
Cost of train/validation at epoch   20 :  0.10956403 0.08221784
Cost of train/validation at epoch   30 :  0.09188860 0.06962664
Cost of train/validation at epoch   40 :  0.07479116 0.06810240
Cost of train/validation at epoch   50 :  0.06701186 0.06320439
Cost of train/validation at epoch   60 :  0.06487222 0.06216515
Cost of train/validation at epoch   70 :  0.05761564 0.06120869
Cost of train/validation at epoch   80 :  0.05512367 0.05802767
Cost of train/validation at epoch   90 :  0.05044307 0.05985889
Cost of train/validation at epoch  100 :  0.05259707 0.05880265
validation set accuracy:0.982143
```

The accuracy is much better than the logistic regression

License

MIT

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 dnn Class Reference

Public Member Functions

- dnn (int n_f, int n_c)
- dnn (int n_f, int n_c, int n_h, const vector< int > &dims, const vector< string > &act_types)
- dnn (int n_f, int n_c, int n_h, const vector< int > &dims, const vector< string > &act_types, const vector< float > &k_ps)
- ∼dnn ()

  destructor, clean the memory space
- void train_and_dev (const vector< float > &X_train, const vector< int > &Y_train, const vector< float > &X_dev, const vector< int > &Y_dev, const int &n_train, const int &n_dev, const int num_epochs, float learning_rate, float lambda, int batch_size, bool print_cost)
- void predict (const vector< float > &X, vector< int > &Y_prediction, const int &n_sample)
- float predict_accuracy (const vector< float > &_X, const vector< int > &Y, vector< int > &Y_prediction, const int &n_sample)
- void initialize_weights ()
- void shuffle (float *X, float *Y, int n_sample)
- void batch (const float *X, const float *Y, float *X_batch, float *Y_batch, int batch_size, int batch_id)
- void batch (const float *X, float *X_batch, int batch_size, int batch_id)
- float max (const float *x, int range, int &index_max)
- void sigmoid_activate (const int &l, const int &n_sample)
- void ReLU_activate (const int &l, const int &n_sample)
- void sigmoid_backward_activate (const int &l, const int &n_sample)
- void ReLU_backward_activate (const int &l, const int &n_sample)
- void get_softmax (const int &n_sample)
- void forward_activated_propagate (const int &l, const int &n_sample, const bool &eval)
- void backward_propagate (const int &l, const int &n_sample)
- void multi_layers_forward (const int &n_sample, const bool &eval)

  multi layers forward propagate and activation
- void multi_layers_backward (const float *Y, const int &n_sample)

  multi layers backward propagate to get gradients
- void weights_update (const float &learning_rate)
- void initialize_layer_caches (const int &n_sample, const bool &is_bp)

  allocate memory space for layer caches A,dZ
- void clear_layer_caches ()

clear layer caches A,dZ
- void initialize_dropout_masks ()

  allocate memory space for dropout masks DropM
- void clear_dropout_masks ()

  clear DropM
- void set_dropout_masks ()
- float cost_function (const float ∗Y, const int &n_sample)

3.1.1 Constructor & Destructor Documentation

3.1.1.1 dnn() [1/3]

```
dnn::dnn (
        int n_f,
        int n_c )
```

constructor without hidden layers, perform logistic regression

Parameters

n_f	No. of features in X
n_c	No. of classes in Y

3.1.1.2 dnn() [2/3]

```
dnn::dnn (
        int n_f,
        int n_c,
        int n_h,
        const vector< int > & dims,
        const vector< string > & act_types )
```

constructor with hidden layer dimensions and activation types specified

Parameters

n_f	No. of features in X
n_c	No. of classes in Y
n_h	No. of hidden layers
dims	integer vector containing hidden layer dimension
act_types	string vector containing activation types for the hidden layers

3.1.1.3 dnn() [3/3]

```
dnn::dnn (
            int n_f,
            int n_c,
            int n_h,
            const vector< int > & dims,
            const vector< string > & act_types,
            const vector< float > & k_ps )
```

constructor with hidden layer dimensions, activation types and dropout keep_probs specified

Parameters

n_f	No. of features in X
n_c	No. of classes in Y
n_h	No. of hidden layers
dims	integer vector containing hidden layer dimension
act_types	string vector containing activation types for the hidden layers
k_ps	keep probabilities for dropout in the hidden layers

3.1.2 Member Function Documentation

3.1.2.1 backward_propagate()

```
void dnn::backward_propagate (
            const int & l,
            const int & n_sample )
```

backward propagate for each layer
if J is the total mean cost, denote all
$\partial J/\partial A \rightarrow dA, \partial J/\partial Z \rightarrow dZ,$
$\partial J/\partial W \rightarrow dW, \partial J/\partial b \rightarrow db,$
$\partial A/\partial Z \rightarrow dF$
and denote layer_dims[l]->n_[l], $*$ for dot product, .$*$ for element-wise product
input: dZ[l],A[l-1],W[l]

update:
db[l](n[l])=sum(dZ[l](n_sample,n[l]),axis=0)
dW[l](n[l],n[l-1])=dZ[l](n_sample,n[l]).T$*$A[l-1](n_sample,n[l-1])
dF=activation_backward(A[l-1](n_sample,n[l-1])
dZ[l-1](n_sample,n[l])=(dZ[l](n_sample,n[l])$*$W[l](n[l],n[l-1])).$*$dF(n_sample,n[l-1])

output: dZ[l-1],dW[l],db[l]

Parameters

l	layer index
n_sample	No. of samples in the datasets

Returns

dZ[l-1],dW[l],db[l] updated

3.1.2.2 batch() [1/2]

```
void dnn::batch (
            const float * X,
            const float * Y,
            float * X_batch,
            float * Y_batch,
            int batch_size,
            int batch_id )
```

Obtain a batch datasets from the full datasets (used for training/developing)

Parameters

X	pointer to data X
Y	pointer to data Y
X_batch	pointer to datasets batched from X
Y_batch	pointer to datasets batched from Y
batch_size	batch size
batch_id	No. of batches extracted, used as an offset

Returns

batched dataset stored in X_batch,Y_batch

3.1.2.3 batch() [2/2]

```
void dnn::batch (
            const float * X,
            float * X_batch,
            int batch_size,
            int batch_id )
```

Obtain a batch datasets from the full datasets (used for predicting)

Parameters

X	pointer to data X
X_batch	pointer to datasets batched from X
batch_size	batch size
batch_id	No. of batches extracted, used as an offset

Returns

batched dataset stored in X_batch

3.1.2.4 cost_function()

```
float dnn::cost_function (
            const float * Y,
            const int & n_sample )
```

Calculate the mean cost using the cross-entropy loss
input: A[n_layers-1], Y

update:
J=-Y.∗log(A[n_layers-1])
cost=sum(J)/n_sample

output:

Parameters

Y	pointer to the datasets Y
n_sample	No. of samples in the datasets the mean cost

3.1.2.5 forward_activated_propagate()

```
void dnn::forward_activated_propagate (
            const int & l,
            const int & n_sample,
            const bool & eval )
```

Forward propagate and activation for each layer
input: A[l-1],W[l],b[l],DropM[l-1]

update:
if dropout==true: A[l-1]=A[l-1].∗DropM[l-1]

A[l]=activation_function(W[l].T∗A[l-1]+b[l])

output: A[l]

Parameters

l	layer index
n_sample	No. of samples in the datasets
eval	if eval==true, dropout is not used

Returns

A[l] updated

3.1.2.6 get_softmax()

```
void dnn::get_softmax (
            const int & n_sample )
```

Calculate the softmax of the given(by default the final) layer
l=n_layers-1
input: Z[l] (stored in A[l])

update:
A[l][i]= exp(Z[l][i])/(exp(Z[l][i]))

output: A[l]

Parameters

n_sample	No. of samples in the datasets

Returns

A[l] stored with the softmax neurons

3.1.2.7 initialize_weights()

```
void dnn::initialize_weights ( )
```

Allocate memory space for W,dW,b,db,
initialize weights W with random
normal distributions and b with zeros

3.1.2.8 max()

```
float dnn::max (
            const float * x,
            int range,
            int & index_max )
```

Get the maximum value and index from the array

Parameters

x	pointer to the array
range	range of array
index_max	reference pointer to argmax of the array

Returns

the maximum value,argmax of the array store in index_max

3.1.2.9 predict()

```
void dnn::predict (
            const vector< float > & X,
            vector< int > & Y_prediction,
            const int & n_sample )
```

Perform prediction for the given unlabeled datasets

Parameters

X	datasets X
Y_prediction	output integer vector containing the predicted labels
n_sample	No. of samples in the datasets

Returns

the predicted labels stored in Y_prediction

3.1.2.10 predict_accuracy()

```
float dnn::predict_accuracy (
            const vector< float > & _X,
            const vector< int > & Y,
            vector< int > & Y_prediction,
            const int & n_sample )
```

Predict and calculate the prediction accuracy for the given labeled datasets

Parameters

X	datasets X
Y	datasets Y (labels)
Y_prediction	output integer vector containing the predicted labels
n_sample	No. of samples in the datasets

Returns

accuracy , and the predicted labels stored in Y_prediction

3.1.2.11 ReLU_activate()

```
void dnn::ReLU_activate (
            const int & l,
            const int & n_sample )
```

Perform ReLU activation for the given layer
input: Z[l] (stored in A[l])

update:
A[l]= Z[l], if Z[l]>0
0 , otherwise

output: A[l]

Parameters

l	layer index
n_sample	No. of samples in the datasets

Returns

A[l] stored with activated neurons

3.1.2.12 ReLU_backward_activate()

```
void dnn::ReLU_backward_activate (
            const int & l,
            const int & n_sample )
```

Perform ReLU backward gradients calculation
input: A[l]

update:
dF= 1, if A[l]$>$0
0, otherwise
dZ[l]=dF.$*$dZ[l]

output:dZ[l]

Parameters

l	layer index
n_sample	No. of samples in the batch

Returns

dZ[l] updated

3.1.2.13 set_dropout_masks()

```
void dnn::set_dropout_masks ( )
```

Initialize dropout masks DropM
if No. of hidden layers $>$0 and dropout==true
assign 1/0 according to keep probabilities keep_probs

3.1.2.14 shuffle()

```
void dnn::shuffle (
            float * X,
            float * Y,
            int n_sample )
```

Shuffle the datasets

Parameters

X	data X
Y	data Y
n_sample	range (or No. of samples) in X,Y to be shuffled

Returns

X,Y being shuffled

3.1.2.15 sigmoid_activate()

```
void dnn::sigmoid_activate (
            const int & l,
            const int & n_sample )
```

Perform sigmoid activation for the given layer
input: Z[l] (stored in A[l])

update:
A[l]=1/(1+exp(-Z[l])

output: A[l]

Parameters

l	layer index
n_sample	No. of samples in the datasets

Returns

A[l] stored with activated neurons

3.1.2.16 sigmoid_backward_activate()

```
void dnn::sigmoid_backward_activate (
            const int & l,
            const int & n_sample )
```

Perform sigmoid backward gradients calculation
input: A[l]

update:
dF= A_[l]∗(1-A_[l]) =A_[l]-A_[l]∗A_[l]
dZ[l]=dF.∗dZ[l]

output:dZ[l]

Parameters

l	layer index
n_sample	No. of samples in the batch

Returns

dZ[l] updated

3.1.2.17 train_and_dev()

```
void dnn::train_and_dev (
            const vector< float > & X_train,
            const vector< int > & Y_train,
            const vector< float > & X_dev,
            const vector< int > & Y_dev,
            const int & n_train,
            const int & n_dev,
            const int num_epochs = 500,
            float learning_rate = 0.01,
            float lambda = 0,
            int batch_size = 128,
            bool print_cost = false )
```

Perform stochastic batch gradient training and evaluation using the validation(developing) data sets

Parameters

X_train	training datasets X
Y_train	training datasets Y
X_dev	validation datasets X
Y_dev	validation datasets Y
n_train	No. of training samples
n_dev	No. of validataion samples
num_epochs	No. of epochs to train
learning_rate	learning rate of of gradients updating
lambda	L2-regularization factor
batch_size	batch size in the stochastic batch gradient training
print_cost	print the training/validation cost every 50 epochs if print_cost==true

Returns

weights and bias W,b updated in the object

3.1.2.18 weights_update()

```
void dnn::weights_update (
            const float & learning_rate )
```

update all the weights W and bias b

for each l from 1 to n_layers-1
W[l]:=W[l]-learning_rate∗dW[l]
b[l]:=b[l]-learning_rate∗db[l]

Parameters

learning_rate	learning rate

Returns

W,b updated

The documentation for this class was generated from the following files:

- dnn.h
- dnn.cpp

Index