Malware Analysis using Data Mining Techniques on GPGPU

Advance Data Mining Project

(Project report)

Submitted for partial fulfillment of the requirement of CS G520 Advance Data Mining

By

Mayank Chaudhari

2016H1030014G

Under supervision of

Dr. S. K Sahay

Assistant professor

Mr. Hemant Rathore

Department of Computer Science and Information Systems

August – December 2017

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE,

PILANI, K. K. BIRLA GOA CAMPUS

CONTENTS

CHAPTER 1

ABSTRACT

Malware variants are continuously evolving in terms of severity level, sophistication and malicious activity so they remain a severe problem for corporations, governments agencies and individuals. This project aims to study the existing GPU based classification approaches in various domains and develop an algorithm to classify Win32 executables as malware or benign using Deep Neural network on CPU and GPU based on features extracted from malwares and benign. Being a static approach, the executable files need not to run either on CPU, GPU or any other virtual environment during detection phase. In this approach we try to group executables into malware and benign based on opcode frequencies during the training phase. The trained model is then used to detect the existing and new malwares. This approach is able to achieve better accuracy then the Naive Bayes approach used in our previous work. Furthermore, the execution time needed for classification using GPU is also good considering its complex nature thus making it a possible candidate for replacing our previous work on malware classification using Naive Byes on GPU.

CHAPTER 2

INTRODUCTION

The extensive use of information technology and reliance on computers has made the information a valuable asset. Computer malwares are a major threat a to today's networked environment. Malwares can be designed to steal, destroy or modify the information. The effect is leakage of sensitive information which can further be used for other malicious tasks affecting human lives in various forms. The most commonly used method for malware detection is signature based detection, that uses specific features of an executable to classify it as malware or benign. In our approach I use the opcode frequency of executables as signature for creating our model. This approach is not effective for previously unknown/self-modifying malwares, due to the same reason there is need to work on a more generalized approach for malware detection. The approaches to tackle this problem can be put under static malware analysis and dynamic malware analysis. Static analysis analyzes the structural properties of the malwares without executing the program under inspection (PUI), on the other hand dynamic analysis monitors the behavior of PUI by running under a sandbox.

For decades' researchers have been working to find more efficient techniques for detection of malware. With the advancement of technology many new techniques such as data-mining and machine learning are also being used in addition to the existing traditional techniques. The drawback of these techniques is that they are compute intensive and slow so this work is focused on developing a fast and efficient solution for malware detection. In this project static malware detection approach is used for detection of

malwares. The primary objective of this project is to find and study an existing GPU based solution for classification problem and study its feasibility for solving the malware classification problem. In case if no such algorithm exists then try to develop our own GPU based solution by modifying the existing serial implementations of the classification algorithms. The algorithms which being considered in this study are SVM, Decision trees, Random Forests, and Neural networks.

Report Layout

Chapter 3, discuss Literature survey. Chapter 4, discuss methodology. In Chapter 5, we present the Experimental analysis and result with comparison with our previous work and the reference papers. Chapter 6, discuss the Conclusion and Future Directions.

CHAPTER 3

Literature Survey

A lot of work has been done in the field of malware analysis using datamining techniques. All the related work is focused on solving the malware detection using CPU only. During this project I explored various implementations of sequential and parallel datamining algorithms for solving different classification problems. Following are some of the previous attempts and standard implementations considered during this project work.

Some SVM based Classification implementations: -

LibSVM - Published in Fan et al. (2005). The LibSVM is a popular CPU-only implementation that we use as a reference. It is available at https://www.csie.ntu.edu. tw/~cjlin/libsvm. The LibSVM stores data in a sparse format. However, the dense variant is available also.

GPU-LibSVM - published in Athanasopoulos and Dimou (2011) and available at http://mklab.iti.gr/project/GPU-LIBSVM. It is a modification of the dense variant of the LibSVM, where only the computation of the kernel matrix elements in only cross-validation mode is ported to GPU.

GPUSVM- Published in Catanzaro et al. (2008) and available at https://code. google.com/p/gpusvm is a more advanced CUDA implementation of a sequential minimal optimization (SMO).

cuSVM- Published in Carpenter (2009) available at http://patternsonascreen. net/cuSVM.html is practically just a CUDA reimplementation of the LibSVM algorithm.

MultiSVM- Published in Herrero-Lopez et al. (2010) available at https://code. google.com/p/multisvm is the first GPU SVM implementation that allows a multiclass classification in the one-vs-all manner besides the two-class problems. A crosstask kernel caching technique is used to significantly reduce total amount of computations needed

gtSVM- Published in Cotter et al. (2011) available at http://ttic.uchicago.edu/ ~cotter/projects/gtsvm uses sparse data format. It

does not use SMO but it uses a larger working set of size 16. A clustering algorithm is used to regularize sparsity patterns in data and permits better memory access. The size of the clusters can be selected from two options: large or small that means 256 or 16 samples, respectively. In the results Section 3 we marked the large clusters and the small clusters variants as" 'gtSVM LC"' and" 'gtSVM SC"', respectively.

WUSVM - Published in Tyree et al. (2014) is available at https://github.com/ wusvm. A sparse primal SVM variant of the training algorithm is implemented in WUSVM. However, the algorithm contains random shuffling of the training data by default that brings a stochastic component that produces models with variable performance in variable training times.

Majority of the SVM implementations focus on parallel implementation of the training process and no further support is being provided as the new scikit learn library proves to be more efficient in terms of the resource utilization and not much is being gained even after using the parallel architecture of the GPU.

List random Forest based Classifiers reviewed: -

CudaRF – It is task parallel implementation of Random Forest classifier using Nvidia GPU and CUDA (Compute Unified Device Architecture) framework. In CudaRF, both learning and classification phases of Random Forest are parallelized. With this, one CUDA thread is used to build one tree in the forest. The speedup achieved for CudaRF is in the range 7.7 – 9.2 is achieved for 128 trees in the forest

CudaTree- CudaTree is an implementation of Leo Breiman's Random Forests adapted to run on the GPU. A random forest is an ensemble of randomized decision trees which vote together to predict new labels. CudaTree parallelizes the construction of each individual tree in the ensemble.

CHAPTER 4

METHODOLOGY

This section discusses about the different components of proposed malware classification project. First of all, we collected the class labelled dataset of malwares and benign and did some preprocessing over it. After preprocessing different models were constructed and tested to get better accuracy and then compared the performance of those models with existing work.

Preprocessing: - The dataset contained some attributes related with origin and corresponding executable which were not needed during model construction so they were removed to decrease the size of input vector to the algorithm. After removing the unwanted attributes, the data was normalized using min-max normalization in order to reduce any biasing effect due to bigger weights. The study is composed of malwares and benign with sizes up to 500 kb so any opcodes above that range was discarded as malwares samples are under 500 kb but the benign samples consisted some files greater than 500 kb also. We also divided the data in the ratio of 9:1 for training and testing.

Neural Network

For classification we used a deep feed forward neural network consisting of three layers where the first three 1809 node layers out of which one dense layers consist of PReLU activation function followed by a dense layer consisting of sigmoid function as third layer being prediction layer.

Design: First of all we choose a deep neural network rather than a shallow but wide neural network because of the developed understanding that deep architectures can be more efficient (in terms of number of fitting parameters) than shallow network. This is necessary for the scope of this project because the no of malware and benign samples with us are very less compared to the executables present in real life so our sampling is limited. The goal was to increase the expressiveness of the neural net, while maintaining the less complex network that can be quickly trained on CPU and GPU.

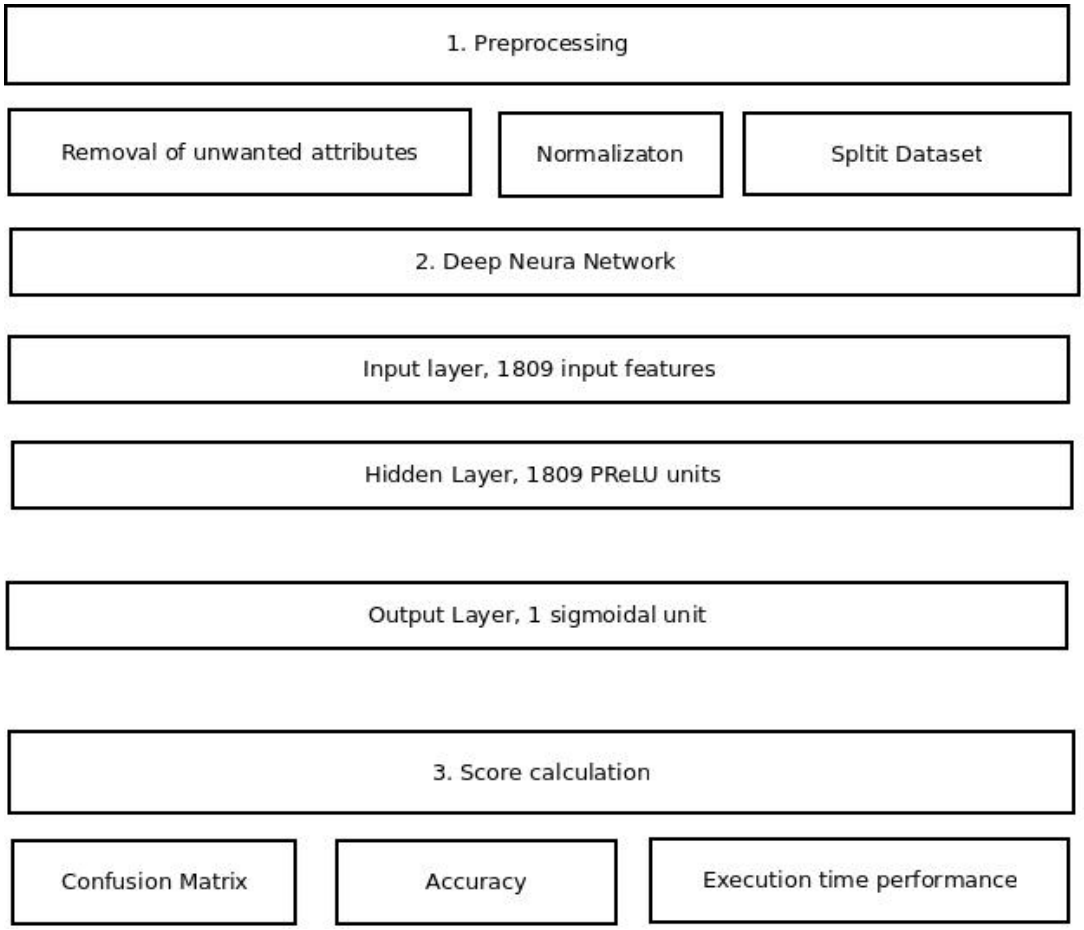

figure 1: Deign of framework

Speeding Up Learning: Rectified linear units(ReLU) have been shown to significantly speedup network training over traditional sigmoid activation functions, such as tanh, by avoiding significant decay in gradient descent convergence rate after an initial set of iterations. This slowdown is due to saturating non-linearities in sigmoid functions at their edges. Using ReLU activation functions can also lead to bad performance when the input values are below 0, and PReLU activator functions are made to dynamically adjust in order to avoid this issue. Thus, yielding significant improved convergence rate.

Initialization of weights, before training, can significantly impact the convergence of the back-propagation algorithm. The goal of a good initialization is to avoid multiplicative impact of weight aggregation from multiple layers during back-propagation. In our approach we use the Gaussian distribution that is normalized based on the size of the input and output of the layers.

CHAPTER 5

EXPERIMENTAL ANALYSIS AND RESULTS

The experiment was performed on both CPU and GPU to record execution time performance and compared with the parallel implementation of parallel Naive Bayes approach for execution time performance calculation and accuracy.

The experimental setup consists of Nvidia 1050ti GPU with 768 cuda cores, 1290 MHz clock frequency and Processor clock of 1392 MHz, 4 GB graphics memory with memory clock of 7 Gbps and Intel Core i7 7700HQ CPU with clock speed of 2.8 GHz and 8GB of RAM. The software uses Keras v2.0.9 deep learning library to implement the neural network model described above using TensorFlow and CUDA as backend. The preprocessing is done by using python.

Results and Analysis:

We carried out test using single, two and three hidden layers and found that there was no significant gain in performance by increasing the complexity of the network. The increasing size of network affects the execution time performance of the classification algorithm. Using only sigmoid layer the training and testing time taken was small and obtained a training accuracy of 94% while we were able to achieve testing accuracy of 91% over a sample of 13068 test cases. But it was having a variable and high false positive rate.

The neural net was also tested one PRelU layer followed by sigmoid layer and two PreLU layers followed by one sigmoid layer without dropout

but during these setups it seemed that the neural network is over-fitting itself so we used cross-validation with a data split of 57:33 and used remaining 10 for testing over our models and got following results.

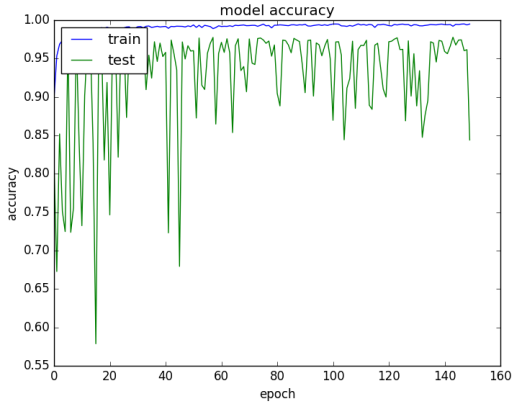

figure 2: 4-layer network with dropout showing train test accuracy

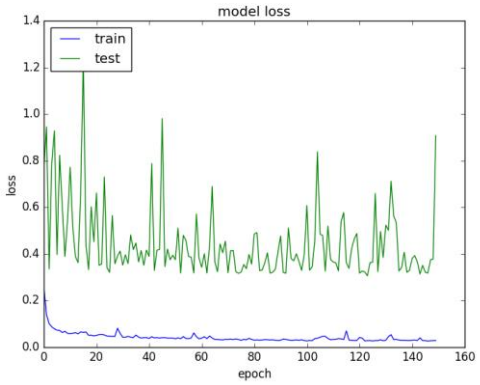

figure 3: 4-layer network with dropout showing train test loss variation

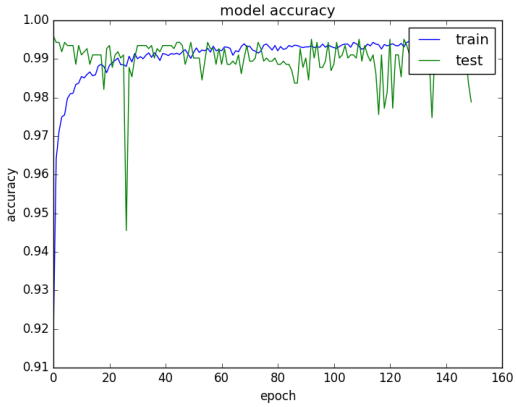

figure 4: 3-layer network with dropout showing train and test accuracy

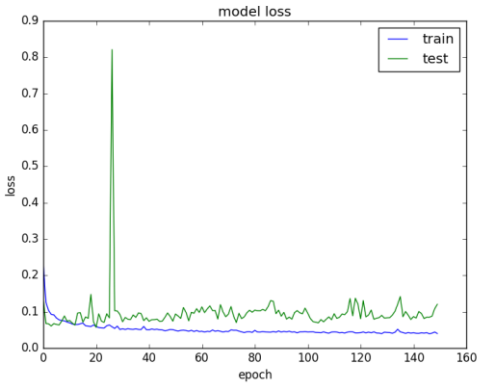

figure 5: 3-layer network with dropout showing train-test loss variation

In our second experiment we changed the split of data for validation and the ratio was of 80:10 and test set of 10% and got following results with 3-layer

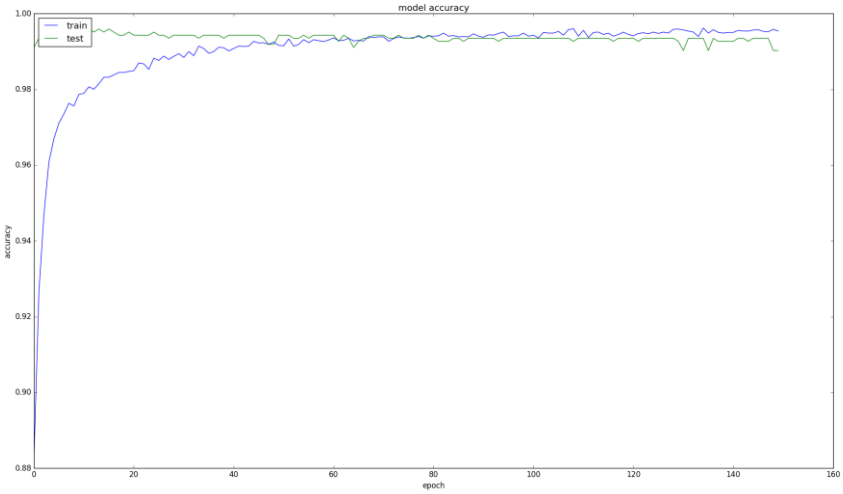

figure 6: 3-layer design with dropout rate of .50

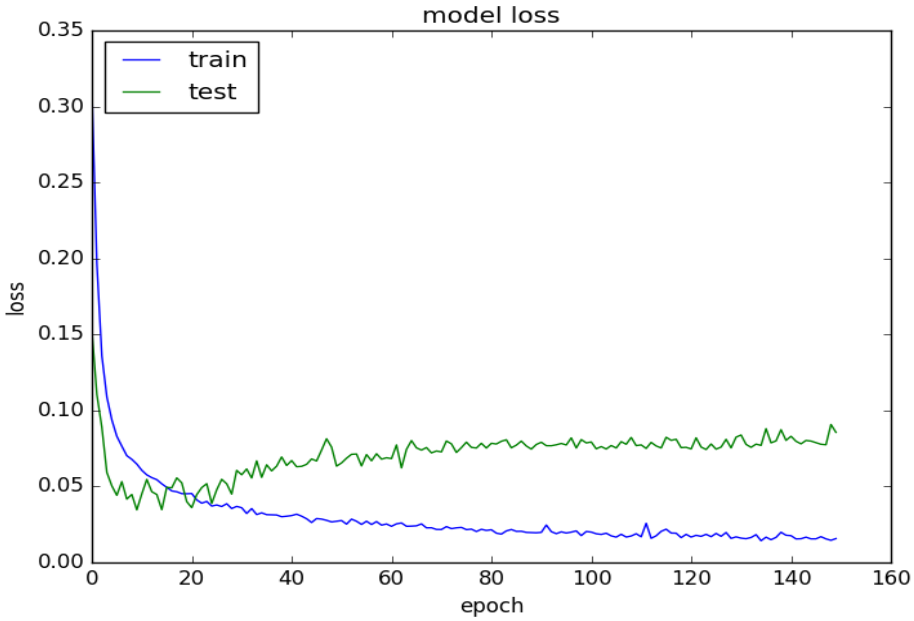

figure 7: 3-layer design with dropout rate of .50

From above observation we can clearly see that the 3-layer design neural network is much better choice for our model as the spikes in train and test accuracy are reduced. We can also observer that the test lost function is changing slowly for train and test so we need to stop here. Moreover to test the accuracy of the 4 layer and 3 layer design we performed a 10 fold cross validation over it and the accuracy can be found in following table.

NO of layers hidden layers	Accuracy Testing(avg)	Best False Positive rate	Worst False Positive rate	10-fold cross-validation
1 sigmoid layer	91%	1%	22%	----------
1 sigmoid and 1 PReLU layer	96.97%	4.1%	5.2%	97.53(+/-58) %
1 sigmoid and 2 PReLU layers	97.77%	3.0%	4.7%	98.10(+/-0.85) %

In our we were able to get the accuracy comparable to the work done by by researchers in “Improving the detection accuracy of unknown malwares by portioning the executables in groups, (Ashu Sharma, Sanjay K Sahay, and Abhishek Kumar)” and were able to reduce the execution time needed for testing by using GPU. This work is able to outperform our previous work of implementing Naive Bayes classifier in terms of accuracy with considerable loss in execution time performance.

The Naive Bayes can be trained faster due to its simple nature and takes 19 seconds on CPU to train while our network takes around 50 min train on CPU and 15 minutes on GPU. While comparing testing performance the parallel Naive Bayes dominates the neural network by considerable

amount. In our study we found the gap to be of 30-40 seconds but still our approach is better than the time taken by the serial version of Naive Bayes classifier by a huge amount. Moreover, in our previous work with Naive Bayes implementation the accuracy obtained was significantly low as compared to neural networks and with increase of total no of samples there was a decrease in accuracy while in neural network it was slightly increasing otherwise constant.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

In this project we studied various parallel implementations of classification algorithms and selected few for our work. We were able to implement a deep neural network approach for malware classification which gave better results compared our previous Naive Bayes approach in terms of accuracy with considerable increase in execution time. This study enabled us to think about some other parallel implementation.

In our future work we would like to extend our work using approaches like parallel Random Forest implementation and convolution neural network out of which parallel implementation of Random Forest is under progress.

BIBLIOGRAPHY

[1] NVIDIA. CUDA C best practice guide. (March), 2011

[2] Ashu Sharma, Sanjay K Sahay, and Abhishek Kumar. Improving the detection accuracy of unknown malwares by portioning the executables in groups.

[3] NVIDIA, CUDA by example, (July), 2010

[4] Nwokedi Idika, Aditys P.Mathur, A survey of malware detection techniques.

[5] Austin Carpenter. cuSVM: A CUDA implementation of support vector classification and regression. patternsonscreen. net/cuSVMDesc. pdf, pages 1–9, 2009.

[6] Qi Li, Raied Salman, and Erik Test. GPUSVM: a comprehensive CUDA based support vector machine package. Open Computer Science, pages 1–22, 2011

[7] Jan Vanˇek, Josef Mich´alek, Josef Psutka, A Comparison of Support Vector Machines Training GPU-Accelerated Open Source Implementations. https://arxiv.org/abs/1707.06470. 2017.

I. APPENDICES

I.1 Dataset

The datasets used in our project are taken from the previous research work and is verified. The malware data set is taken from malicia and the benign dataset is ckecked with the help of virustotal for validity. Although the dataset is genuine but still it is skewed as the no of malware and benign are not equally distributed inside it. Another problem is the less number of samples available in dataset. The actual dataset consists of 1814 attributes in following format (same for benign and malware)

<name> <size><source><class><no_opc><[…….1808features…….]>

For our study it was converted to following form

<class><[……...1808…….]> features.

I.2 Source code

I.2.1 Testing code(load_model_test.py)

```
from keras.models import Sequential

from keras.layers import Dense

from keras.models import model_from_json

import numpy

import time


## run with following command to use only GPU0 for cup only use-1

#CUDA_VISIBLE_DEVICES=0 python test.py

dataset = numpy.loadtxt("/home/mayank/Desktop/data/testing/neural/train_wo_size.csv",
delimiter=",")

#split data-set

#class label

X = dataset[:,0]
```

```
#fefature vectors

Y = dataset[:,1:1809]

dataset1 = numpy.loadtxt("/home/mayank/Desktop/data/testing/neural/test_wo_size.csv",
delimiter=",")

#split data-set

X1 = dataset1[:,0]

Y11 = dataset1[:,100:1809]

Y1 =dataset1[:,1:1809]

print "no of predicions to be made ",len(Y1)

# load json and create model

json_file = open('model3d50G80epc.json', 'r')

loaded_model_json = json_file.read()

json_file.close()

loaded_model = model_from_json(loaded_model_json)

# load weights into new model

loaded_model.load_weights("model3d50G80epc.h5")

print("Loaded model from disk")

# evaluate loaded model on test data

loaded_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

score = loaded_model.evaluate(Y, X, verbose=0)

print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))

#calculating time spent in prediction

start = time.time()

# calculate predictions

predictions = loaded_model.predict(Y1)

end = time.time()

test_time=(end-start)

print "time elapsed in predictions  ", test_time
```

```
# round predictions
rounded = [round(x[0]) for x in predictions]
#print(rounded)
#print(Y1)
## creating confusion matrix
tests = len(rounded)
fp =0
tp =0
fn =0
tn =0
#print (tests)
for i in range(tests):
    if((X1[i] == 0) and (int(rounded[i])==0)):
        tn=tn+1
    if((X1[i] == 1) and (int(rounded[i])== 1)):
        tp=tp+1
    if((X1[i] == 0) and (int(rounded[i])==1)):
        fp=fp+1
    if((X1[i] == 1) and (int(rounded[i])== 0)):
        fn=fn+1
print (" confusion matrix for givem test data is : ")
print ("tp  ","tn   " ,"fp  ","fn")
print (tp,"  ",tn,"   ",fp,"   ",fn)
# prediction accuracy
prediction_accuracy =float((tp+tn))/float((tp+tn+fp+fn))
print("prediction_accuracy is: ", prediction_accuracy*100,"%")
# true positive rate
```

```
true_positive =float((tp))/float((tp+fn))

print("true_positive rate is: ", true_positive*100,"%")

# true negative rate

true_negative =float((tn))/float((tn+fp))

print("true_negative rate is: ", true_negative*100,"%")

# false positive rate

false_positive =float((fp))/float((tn+fp))

print("false_positive rate is: ", false_positive*100,"%")

# false negative rate

false_negative =float((fn))/float((tp+fn))

print("false_negative rate is: ", false_negative*100,"%")

f= "score3Ld50epc80.txt"

fd=open(f,'a+')

msg =str(len(Y1))+" "+str(prediction_accuracy*100)+" "+str(true_positive*100)+"
"+str(true_negative*100)+" "+str(false_positive*100)+" "+str(false_negative*100)+"
"+str(test_time)+"\r\n"

fd.write(msg)
```

I.2.2 Training Code (neural_train.py)

```
import keras

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout, Activation

from keras.models import model_from_json

from keras.models import load_model

import numpy

import time
```

```
## ********** Function for training and dumping the model  **********
def train(infile):
    numpy.random.seed(7)

    #dataset = numpy.loadtxt("../pima-indians-diabetes.csv", delimiter=",")
    dataset = numpy.loadtxt(infile+".csv", delimiter=",")

    #split data-set

    #X = dataset[:,0]
    #Y = dataset[:,1:9]
    #class label
    X = dataset[:,0]
    #fefature vectors
    Y = dataset[:,1:1809]



    #print X
    #print Y

    ## MOdel definitin

    model = Sequential()
    # using gaussian initializer to avoid multiplicative impact of weight aggregation
    # others can also be tried
    #act = keras.layers.advanced_activations.PReLU(init='normal', weights=None)
```

```
#model.add(Dense(1,input_dim=1808, init ='uniform', activation ='sigmoid'))
model.add(Dense(1808,input_dim=1808, init ='uniform'))
model.add(keras.layers.advanced_activations.PReLU())
model.add(Dropout(0.5))
#model.add(Dense(1808,init ='uniform'))
#model.add(keras.layers.advanced_activations.PReLU())
#model.add(Dropout(0.3))
model.add(Dense(1,activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit the model
start_train= time.time()
model.fit(Y, X, epochs=150)
end_train=time.time()
print "time elapsed in training   ",(end_train-start_train)
#calculating time elapsed in predictions
start = time.time()

# calculate predictions
predictions = model.predict(Y)

end =time.time()
print "time elapsed in prediction   ",(end-start)

# evaluate the model
```

```
    scores = model.evaluate(Y, X)
    print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))



    rounded = [round(x[0]) for x in predictions]
    #print(rounded)
    ############----------------------------------###################
    # for dumping a model there are two steps
    # 1 -> serialoze /dump model to json/yml file
    # 2 -> serialize /dump weights to json file

    ## 1-> serailizing model to jason filea
    model_json = model.to_json()
    with open("model3d50G80epc.json","w") as json_file:
        json_file.write(model_json)

    ## 1-> serializing weights
    model.save_weights("model3d50G80epc.h5")
    print("Saved model to disk")




def main():
    f="/home/mayank/Desktop/data/testing/neural/train_wo_size"
    train (f)

if __name__ == "__main__":
```

```
    main()
```

I.2.3 Cross validation (plot_cross_validaton.py)

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout, Activation
from keras.models import model_from_json
from keras.models import load_model
import matplotlib.pyplot as plt
import numpy
import time


## ********** Function for training and dumping the model  **********
def train(infile):
    numpy.random.seed(7)

    #dataset = numpy.loadtxt("../pima-indians-diabetes.csv", delimiter=",")
    dataset = numpy.loadtxt(infile+".csv", delimiter=",")

    #split data-set

    #X = dataset[:,0]
    #Y = dataset[:,1:9]
    #class label
```

```
X = dataset[:,0]
#fefature vectors
Y = dataset[:,1:1809]



#print X
#print Y

## MOdel definitin

model = Sequential()
# using gaussian initializer to avoid multiplicative impact of weight aggregation
# others can also be tried
#model.add(Dense(1,input_dim=1808, init ='uniform', activation ='sigmoid'))
model.add(Dense(1808,input_dim=1808, init ='uniform'))
model.add(keras.layers.advanced_activations.PReLU())
#model.add(Dropout(0.2))
#model.add(Dense(1808,init ='uniform'))
#model.add(keras.layers.advanced_activations.PReLU())
model.add(Dropout(0.5))
model.add(Dense(1,activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit the model
start_train= time.time()
```

```
history = model.fit(Y, X, validation_split=0.10, epochs=150, verbose=1)
end_train=time.time()
print "time elapsed in training   ",(end_train-start_train)
#calculating time elapsed in predictions
print(history.history.keys())

# evaluate the model
scores = model.evaluate(Y, X)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))



plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
def main():
    f="/home/mayank/Desktop/data/testing/neural/train_wo_size"
    train (f)

if __name__ == "__main__":
    main()
```

I.2.4 k-fold cross-validation (cross_validaror.py)

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout, Activation
from keras.models import model_from_json
from keras.models import load_model
from sklearn.model_selection import StratifiedKFold
import numpy
import time



## ********** Function for training and dumping the model  **********
def train(infile):
    seed=7

    dataset = numpy.loadtxt(infile+".csv", delimiter=",")

    #split data-set
    #class label
```

```
X = dataset[:,0]
#fefature vectors
Y = dataset[:,1:1809]

#print X
#print Y

# define 10-fold cross validation test
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
ccvscore =[]
for train,test in kfold.split(Y,X):

    ## MOdel definitin

    model = Sequential()
    # using gaussian initializer to avoid multiplicative impact of weight aggregation
    # others can also be tried
    #act = keras.layers.advanced_activations.PReLU(init='normal', weights=None)
    #model.add(Dense(1,input_dim=1808, init ='uniform', activation ='sigmoid'))
    model.add(Dense(1808,input_dim=1808, init ='uniform'))
    model.add(keras.layers.advanced_activations.PReLU())
    model.add(Dropout(0.5))
    #model.add(Dense(1808,init ='uniform'))
    #model.add(keras.layers.advanced_activations.PReLU())
    #model.add(Dropout(0.3))
    model.add(Dense(1,activation='sigmoid'))
```

```
        # Compile model
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

        # Fit the model
        start_train= time.time()
        model.fit(Y[train], X[train], epochs=10,verbose=0)
        end_train=time.time()
        print "time elapsed in training   ",(end_train-start_train)
        #calculating time elapsed in predictions
        start = time.time()
        end =time.time()
        print "time elapsed in prediction   ",(end-start)

        # evaluate the model
        scores = model.evaluate(Y[test], X[test])
        print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

        ccvscore.append(scores[1]*100)

    print("%.2f%% (+/- %.2f%%)" % (numpy.mean(ccvscore), numpy.std(ccvscore)))



def main():
    f="/home/mayank/Desktop/data/testing/neural/train_wo_size"
    train (f)
if __name__ == "__main__":
    main()
```

1.2.4 Preprocessing scripts

Normalization.py

```
import sys, argparse, csv
from decimal import Decimal


# 1 used for normalization
f= "sorted_merged_b_m_numeric_label.csv"
# open csv file
max =[0] * 1810
print max
print len(max)
with open(f, 'rb') as csvfile:


    # get number of columns
    for line in csvfile.readlines():
        line1=line
        array = line.split(',')
        print len(array)
        first_item = array[0]
        second_item =array[1]
        #print first_item
        for x in range (0,1810):
            if int(max[x]) <= int(array[x]):
                max[x] = int(array[x])



norm =[0] * 1810
```

```
print len(norm)
with open("sorted_merged_b_m_numeric_label.csv", 'rb')as csvfile_r:
    for line in csvfile_r.readlines():
        line1=line
        array_n = line.split(',')
        print len(array)
        norm[0] = array_n[0]
        norm[1] =array_n[1]
        #print first_item
        for y in range (2,1810):
            if int(array_n[y]) > 0:
                norm[y] = format(float(array_n[y])/float(max[y]),'.5f')
            else:
                norm[y]=format(float(0),'.5f')

        with open("./normalized_data/sorted_merged_b_m_numeric_label.csv",'ab')as csvfile_w:
            wr = csv.writer(csvfile_w,lineterminator='\r')
            wr.writerow(norm)
```

Test-train_splitter.py

```
import sys, argparse, csv

norm =[0] * 1810
l_count =0
print len(norm)
with open("45.csv", 'rb')as csvfile_r:
    for line in csvfile_r.readlines():
```

```
line1=line
array_n = line.split(',')
print len(array_n)
norm[0] = array_n[0]
norm[1] =array_n[1]
#print first_item
if (l_count %10) == 0:
    with open("test.csv",'ab')as csvfile_te:
        csvfile_te.write(line)
else:
    with open("train.csv",'ab')as csvfile_tr:
        csvfile_tr.write(line)

l_count =l_count+1
```

I.3 Example

To run training script on GPU

```
CUDA_VISIBLE_DEVICES=0 python2.7 four_layer_train.py
```

To run training script on CPU

```
CUDA_VISIBLE_DEVICES=-1 python2.7 four_layer_train.py
```

similarly, other scripts can also be run

I.4 Readme

To replicate this project, one need the following packages

1→ Cuda programming api

2→ Python2.7

3→ Scikit

4→ Numpy

5→ TensorFlow

6→ Keras

To run the scripts follow the given steps

1→ First remove the extra attributes from the dataset

2→ Run normalization script over it

3→ Run data_splitter script over normalized data.

4→ Now run training script

5→ Run cross-validation script

6→ Finally run the testing script