# ECE 8720
# Artificial Neural Networks
# Takehome #2 Report

**Submitted By**
Ravisutha Sakrepatna Srinivasamurthy
rsakrep@clemson.edu

April 10, 2018

# Part I
# Takehome2 Proposal

## 1 Introduction

I have come up with two ideas to detect "yellow boxes" in the EEG signals. I was reading about different types of RNNs and I came across an RNN called **Long Short Term Memory (LSTM)** [1]. I found some really impressive applications of this technique[2]. I want to implement this technique as a part of takehome 2 and I am proposing two ideas to classify yellow-boxes which are discussed in the next sections.

## 2 Goals

Many researchers have used (different) RNN models to predict next word in a sequence by training it with whole/parts of Wikipedia and/or from other resources[2][3][4][5]. I am proposing a similar approach to our problem but with a different intention. Next section deals with EEG signal and some challenges in the data.

### 2.1 Data

The EEG signals are frequency normalized to 256Hz. EEG for each patient is taken for 30 seconds. That's a total of 7680 samples for each patient and 200 such measurements are taken. The major hurdle in applying LSTM method is that abnormal activities are marked as yellow boxes by neurologists using *different* montages and this hurdle is addressed in the following subsections. For both ideas, I intend to show 60 samples at once to the RNN.

### 2.2 Idea #1

I want to first train the LSTM with EEG signals without yellow boxes (normal brain signals). I may even want to overfit the network with this data. Now if I show an EEG signal without yellow-boxes (normal activity) to this network, it should track this signal pretty closely. And, if I show an EEG signal containing yellow-box (abnormal activity), it might (should) fail to track the signal and thereby indicating the presence of abnormal activity. For this idea to work, I have to choose two signals - one without yellow-box and other with yellow-box but both should be based on same montage (difference in electrode signals).

### 2.3 Idea #2

I can just treat it as a classification problem and train RNN to classify into yellow and non-yellow boxes. There are two ways in which the data can be trained.

1. Use montages specified by neurologists.
2. Use all electrode data i.e 19 electrodes and a ground and a common system reference electrode[6] and let RNN figure out montages.

## 3 Expected Outcomes/Deliverables

As explained in the above section there are two things that I want to accomplish. First, I want to implement LSTM and I will be using **python-3.5**. Next, I will be using this implemented LSTM to **detect abnormal brain activity (yellow-box)**. Hence, final submission will contain python source code and detection results and analysis.

# Appendices

**Preliminary results**
I used LSTM with 4 layers (each layer having 50 memory units) to train just one patient's signal (which doesn't contain yellow-boxes). RNN takes in previous 60 samples and predicts next sample. Following figure shows the predicted vs real signal.

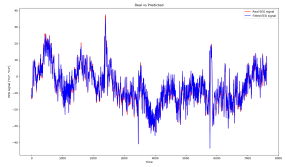

Figure 1: LSTM predicted vs real EEG signal.

# Part II
# Takehome2 Report

## 4 Introduction

For takehome2, I set out to apply LSTM technique to detect "yellow boxes" in EEG signal. I had proposed two techniques (It will be referred to as Idea#1 and Idea#2). Even though I had starter code (written at the beginning of Fall 2017) to pre-process EEG signal, it took lot of time to get everything right(especially sampling frequency and montages). This is when I experienced shear complexity of EEG first hand. Data related discussions are presented in section 5. I was able to implement both ideas with the help of Keras [7]. These ideas are further explored in sections 6.3 and 6.4. Section 8 contains (surprising) results and finally section 9 describes summary, future possible directions and conclusions.

## 5 Data Pre-processing

The dataset was given by Dr. Schalkoff at the beginning of Fall 2017. Though I had written some code to pre-process the data, it was not nearly enough for this project. It took some time to convert raw EEG signals stored in edf files to montage signals as specified by physicians in montage.csv. As these signals were of different frequencies, signals were frequency normalized to 256Hz. Data presented to LSTM varies for two implementations and is discussed in respective subsections. Data preprocessing is handled by *reading_edf.py*. All yellow boxes identified by physicians are stored in *./Outputs/Yellow_box* folder.

## 6 Implementation

### 6.1 LSTM Overview

Unlike Hopfield which uses Hebbian storage prescription, Long Short Term Memory (LSTM) uses iterative, gradient descent approach. Though there are multiple iteration based learning algorithms for RNN, most of them suffer from either vanishing gradient or gradient explosion problem when gradients of error function is back-propagated through time. This implies that RNN loses "contextual" information though it can retain "short-term" memory"[8]. LSTM solves this problem by using what are called "*valves*" or "*gates*". Basic building block of LSTM is called a cell and is as shown in the Figure2. LSTM has three gates - input gate, forget gate and output gate. These gates control the information flow. Input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit[9].

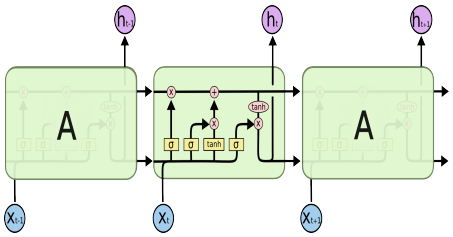


Figure 2: LSTM Simplified Cell

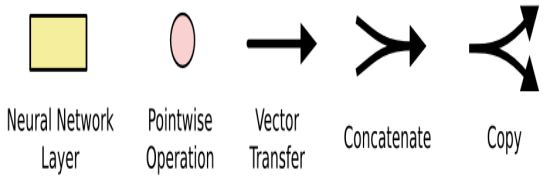


Figure 3: Operations performed in a cell

## 6.2 LSTM Implementation and Architecture

**Implementation** LSTM was implemented using Keras. Reasons behind this decision are - First, Keras parallelizes operations wherever possible. Even after doing so, it took 3 hours to train RNN for 100 epochs. Second, it took more time for me to pre-process the data than I anticipated and hence couldn't write LSTM from scratch.

**Architecture** In this implementation, 4 LSTMs are stacked sequentially (Figure 6). Ouput of the last LSTM is fed to feed-forward layer which gives an output. Each LSTM outputs a sequence of 100 samples. Hence, each LSTM layer has 100 cells. Dropout regularization of 20% is used for better generalization. Length of input sequence was chosen as 30. LSTM looks at a sample, predicts the next output. Once it reaches the end, next sequence is fed to LSTM which is a slided version of 1st sequence. For ex. lets say there are 1000 samples. And if the chosen sequence length is 30 then first sequence will be from 0-29, next sequence will be 1-30, then 2-31 etc. until 970-999. This is repeated epoch number of times. The architecture is depicted pictorially in Figure 6.

## 6.3 Idea #1

First idea was to predict the next output in a sequence. LSTM is trained with only normal EEG signals but tested with yellow box signals. Intuitively, if yellow box signals are presented, LSTM should have difficulty in predicting this signal.

**Training** There are 200 edf files out of which 89 files contain yellow box annotation. Plots of these annotated signals are stored in *Yellow_box* folder. Remaining files do not contain any annotations. Training data was generated by picking up 100 samples from 100 random *normal* signals with random but valid montages. Hence in total, there are $100 * 100 = 10,000$ normal signal samples of different montages. Sample code is shown below.

```
for file_no in range (len(self.edf_files)):
    # If file doesn't contain yellow-box
    if file_no not in self.montage_info:
        # Pick 100 samples from a random start point
        start = int (np.floor(np.random.rand()) * (250 * 30 - 100))
        end = start + 100
        sample = self.read_edf_files(file_no , montage)[start : end]
```

**Testing** To test the learned LSTM, 20 normal signals (each of 100 samples) and 20 yellow box signals were chosen in random. For each of the test cases, RMSE values are measured and plotted. Results are discussed in section 8.

### 6.4 Idea #2

The second idea was to treat this as a classification problem. Training and testing for this model slightly differs.

**Training** Input is gathered similar to idea #1. Only difference is that output will be either 1 or -1. If the input is normal signal, expected output +1 else -1.

**Testing** Learned LSTM is tested using 80 normal and 80 yellow box signals. Confusion matrix is shown in Section 8.

## 7 Justifications for few decisions

Lot of decisions were made during the course of this project. Some justification behind these decisions are listed below.

1. Number of samples picked from each signal was set as 100. This is because, average number of samples in a yellow box is around 60 and max is around 150. Choosing something in between made sense.
2. Decision to employ 100 cells in each LSTM layer was based on a toy example involving Google stock price value prediction. For that experiment, 4 stacked LSTMs with 100 hidden units really worked well. Same architecture was retained.
3. Initially number of epoch was set as 100. But later it was found that setting epoch to 30 wouldn't make too much of a difference in accuracy but training was lot faster.

## 8 Results

### 8.1 Idea #1 output

The root mean square error (RMSE) is calculated for both types of test data - normal and yellow-box. It was found that avg. RMSE for yellow-box was greater than normal signal. But the difference is not that significant. The LSTM tracked yellow box signals pretty well. An example is shown in Figure 4.

**Average RMSE for normal signal**

$$avg = 11.5731$$

**Average RMSE for yellow-box signal**

$$avg = 15.5810$$

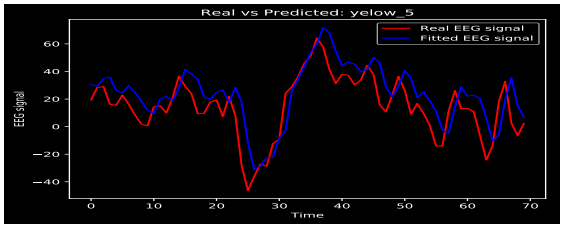


Figure 4: LSTM fitting an yellow box

### 8.2 Idea #2 output

To test idea2, the test data consisted of 80 normal signals and 80 yellow box signals. Following confusion matrix depicts the performance of LSTM.

| | Predicted | |
|---|---|---|
| Actual | TP: 80 | FN: 0 |
| | FP: 0 | TN: 80 |

The accuracy of this model is 100%. I was initially skeptical of the result and hence I repeated the experiment. These 80 normal and yellow box signals are drawn at random. This experiment was repeated many times but result was the same. A screenshot of the output is shown in Figure 5.

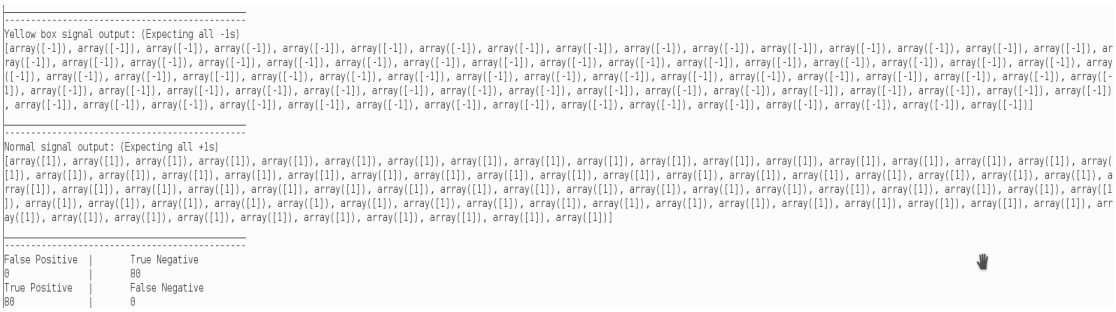


Figure 5: LSTM Idea#2 output.

## 9 Summary and conclusions

**Summary** To summarize, LSTM was trained on EEG signal. Two ideas proposed were implemented. Idea #1 didn't yield conclusive result but surprisingly, Idea#2 performed better than expected.

**Future directions** Though Idea #2 was successful in classifying signals into yellow, normal signals, it will be interesting to see if this technique can classify EEG into AEP, non-AEP classes. Also in the future, other LSTM architectues such as convolutional LSTM could be explored. Finally, what LSTM really learnt is beyond the scope of this project. But this will also be an interesting topic.

# References


[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[2] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[3] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[4] Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 1017–1024, USA, 2011. Omnipress.

[5] Kyunghyun Cho, Bart van Merriënboer, Çalar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[6] Ashish Ganta. Detection and Classification of Epileptiform Transients in EEG signals using Convolution Neural Networks. Number August. 2017.

[7] Keras. `http://keras.io`.

[8] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013.

[9] Long short-term memory. `https://en.wikipedia.org/wiki/Long_short-term_memory`.

# Appendices

**LSTM Model**
Because the figure is large, this is presented in appendices.

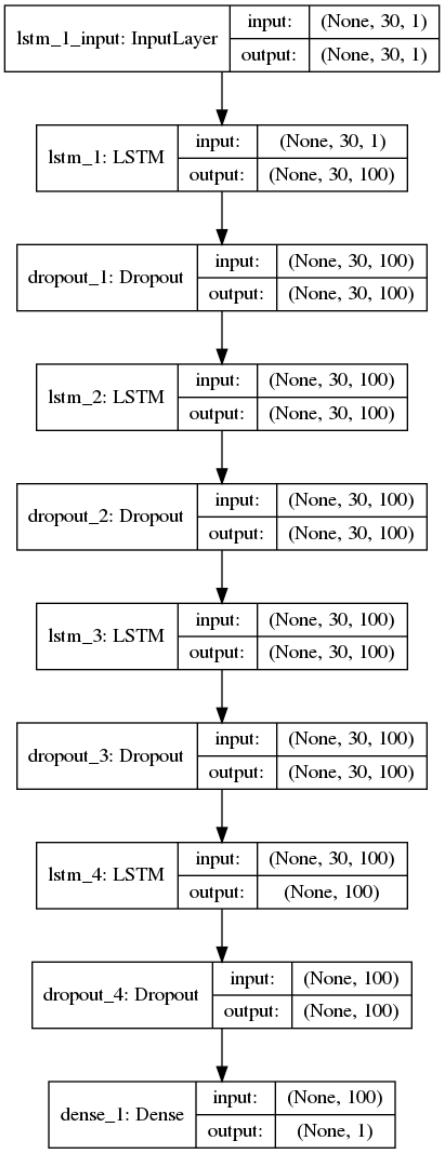


Figure 6: LSTM model.

**Data Pre-processing**

```
#Author : Ravisutha Sakrepatna Srinivasamurthy
#Purpose: EDF to EEG database framework

import random as rand
import csv
import inspect
import matplotlib.pyplot as plt
import pyedflib
import sys
import scipy.signal
import numpy as np
from os import listdir
from os.path import isfile, join

class edf_convert:
    """ Framework for reading edf files and converting the sampled data into a matrix

    def __init__ (self, samp_freq=256):
        """ Run all of the following functions to construct matrix which will be used

        self.samp_freq = 256

        #Get the path for edf files, montage.csv and fix-xx.csv
        self.get_path ()

        #Get names of each edf files
        self.get_files_name ()

        #Get montage list stored in montage.csv
        self.montage_list ()

        #Get signals
        self.get_signals ()

    def get_path (self):
        """ Get the path of folder containing EDF files."""

        if (len (sys.argv) < 4):
            if (len (sys.argv ) < 2):
                self.path_edf = input ('Please provide path for edf files:')
            if (len (sys.argv) < 3):
                self.path_montage = input ('Please provide path for montage.csv:')
            if (len (sys.argv) < 4):
                self.path_fix = input ('Please provide path for fix-xx.csv:')
        else:
            self.path_edf = sys.argv[1]
            self.path_montage = sys.argv[2]
            self.path_fix = sys.argv[3]
```

```
def get_files_name (self):
    """ Get the list of edf files."""

    self.edf_files = []
    for f in listdir (self.path_edf):
        if (isfile(join (self.path_edf, f))):
            self.edf_files.append (join(self.path_edf, f))

def montage_list (self):
    """ Get montage list and store it in a dictionary. """

    self.montage = {}
    self.mont_list = set ()
    with open(self.path_montage, newline='') as csvfile:
        mont = csv.reader(csvfile, delimiter=',')
        for row in mont:
            self.mont_list.add (row[2])
            self.mont_list.add (row[3])
            self.montage[row[0] + row[1]] = [row[2], row[3]]

    #Replace ECG by ECGL and ECGR
    self.mont_list.remove ('avg')
    self.mont_list.remove ('ECG')
    self.mont_list.add ('ECGL')
    self.mont_list.add ('ECGR')

def get_signals (self):
    """ Get signals for the given event and convert it to montage signal """

    self.montage_info = {}
    self.files_mont = []
    self.time_info = []
    self.file = []
    file_no = 0
    j = 0

    with open(self.path_fix, newline='') as csvfile:
        fix = csv.reader(csvfile, delimiter=',')
        for row in fix:

            # Find the file number
            file_no = int (np.ceil (float (row[2]) / 30))

            # For some reason, I'm not considering avg montage
            if (self.montage[row[5]+row[4]][1] == 'avg'):
                continue

            file_temp = []
```

```
                #Store file_no, start_time, end_time, montage number and montage sign
                file_temp.append (file_no)
                file_temp.append (str (np.fmod (float (row[2]), 30)))
                file_temp.append (str (np.fmod (float (row[3]), 30)))
                file_temp.append (self.montage[row[5] + row[4]])
                self.time_info.append ([file_no, file_temp[1], file_temp[2]])

                self.files_mont.append (self.montage[row[5]+row[4]])

                file_temp.append (self.read_edf_files (file_no, self.montage[row[5] +

                # self.file contains all montage information
                self.file.append (file_temp)

                # Put it in a dictionary
                if file_no not in self.montage_info:
                    self.montage_info[file_no] = []
                self.montage_info[file_no].append (file_temp[1:])

    def read_edf_files (self, file_no, mot_signals):
        """ Read given EDF file and return all signals. """

        # Convert file number to file name
        path = self.convert_to_file_name (file_no)

        # Read file
        f = pyedflib.EdfReader(path)

        # Get sampling frequencies
        samp_freq = f.getSampleFrequencies()[0]

        # Allocate two arrays for two electrode signal
        sigbufs = np.zeros((2, f.getNSamples()[0]))

        # Get all available electrode names
        self.label = f.getSignalLabels ()

        # Read first electrode signal
        sigbufs[0, :] = f.readSignal(self.label.index (mot_signals[0]))

        # Read second electrode signal
        if (mot_signals[1] != 'avg'):
            sigbufs[1, :] = f.readSignal(self.label.index (mot_signals[1]))
        else:
            sigbufs[1, :] = self.get_avg (f)

        f._close ()
```

```
        # Montage (difference in electrode signals)
        sig = sigbufs[0, :] - sigbufs[1, :]

        # Take only first 30seconds
        sig = sig[0: 30 * samp_freq]

        # Frequency normalize
        sig = self.frequency_normalize_256 (sig, samp_freq)

        return (sig)

    def convert_to_file_name (self, file_no):
        """ Convert the file no. to edf file name."""

        if (file_no / 100 < 1):
            if (file_no / 10 < 1):
                a = '00' + str(int (file_no))
            else:
                a = '0' + str(int (file_no))
        else:
            a = str (int (file_no))

        path = self.path_edf + '/s3_' + a + '.edf'

        return (path)

    def frequency_normalize_256 (self, sig, samp_freq):
        """ Upsample or downsample the signal to 256 Hz."""

        if (samp_freq > self.samp_freq):
            r = int (samp_freq / 256)
            sig = scipy.signal.decimate (sig, r, zero_phase=True)
            samp_freq = int (samp_freq / int (r))

        if (samp_freq < 256):
            sig = sig[0:samp_freq * 30]
            (r1, r2) = (samp_freq / 256).as_integer_ratio ()
            sig = scipy.signal.resample_poly (sig, r2, r1)

        return (sig)

    def get_avg (self, f):
        """ Get average of all the signals """

        n = f.signals_in_file
        temp = np.zeros ((len (self.mont_list), f.getNSamples ()[0]))
        j = 0

        for i in self.mont_list:
```

```
            temp[j, :] = f.readSignal (self.label.index (i))
            j = j + 1

        M = np.dot (np.ones ((1, len (temp))), temp)
        M = M / n
        return (M)

    def get_normal_signal (self, num_samples=1, sample_num=-1, random=False, truncate
        """ Get signal which doesn't contain yellow-box. """

        if truncate:
            samples = np.zeros ((num_samples, 100))
        else:
            samples = np.zeros ((num_samples, 30 * 256))

        if random:
            rand.shuffle(self.file)

        row = 0;

        if sample_num != -1:
            for file_no in range (len(self.edf_files)):
                file_no += 1
                if file_no not in self.montage_info:
                    row += 1;
                    # Return "Row"th sample
                    if row == sample_num:
                        if truncate:
                            start = int (np.floor (np.random.rand()) * (250 * 30 - 10
                            end = start + 100
                            samples[0] = self.read_edf_files (file_no, montage)[start
                        else:
                            samples[0] = self.read_edf_files (file_no, montage)
                        break

        else:
            for file_no in range (len(self.edf_files)):
                file_no += 1
                if file_no not in self.montage_info:
                    if truncate:
                        start = int (np.floor (np.random.rand()) * (250 * 30 - 100))
                        end = start + 100
                        samples[row] = self.read_edf_files (file_no, montage)[start :
                    else:
                        samples[row] = self.read_edf_files (file_no, montage)
                    row += 1;

                if (row >= num_samples):
                    break
```

```
    return (samples)

def get_yellow_signals (self, num_samples=1, sample_num=-1, random=False, truncat
    """ Get signal which contains yellow-box. """

    if truncate:
        samples = np.zeros ((num_samples, 100))
    else:
        samples = np.zeros ((num_samples, 30 * 256))

    # Number of samples
    if num_samples == "ALL":
        num_samples = len (self.file)

    details = []
    row = 0;

    # Return specific sample
    if sample_num != -1:
        f = self.file[sample_num]
        if truncate:
            start = int (np.floor (float (f[1]) * 250))
            end = start + 100
            samples[0] = f[-1][start:end]
        else:
            samples[0] = f[-1]
        details.append (f[0:-1])

    # Return N-number of samples
    else:
        # Shuffle if random is true
        if random:
            rand.shuffle(self.file)

        for i, f in enumerate (self.file):
            if truncate:
                start = int (np.floor (float (f[1]) * 250))
                end = start + 100
                samples[i] = f[-1][start:end]
            else:
                samples[i] = f[-1]
            details.append (f[0:-1])
            row += 1;

            if (row >= num_samples):
                break

    return ((samples, details))
```

```
def plot_eeg (num=3, only_eeg=False, show=False, save=False):
    """ Plot eeg signals containing yellow box.
    Input:
    num: Number of different signals needed.
    """
    (M, details) = edf_convert().get_yellow_signals (num, random=False)
    print (details)
    for i in range (len(M[:, 1])):
        start = int (np.floor (float (details[i][1]) * 250))
        end = int (np.ceil (float (details[i][2]) * 250))

        x = np.array ([start, end])
        m1 = max (M[i, int (np.ceil (start)) : int (np.ceil (end))])
        m2 = min (M[i, int (np.ceil (start)) : int (np.ceil (end))])
        y1 = np.array ([m1, m1])
        x2 = np.array ([start, start, end, end])
        y2 = np.array ([m1, m2, m2, m1])
        plt.figure (facecolor='b')

        if (only_eeg == True):
            plt.plot (M[i, start : end])
            plt.title ("Plot of yellow box. File no: {0}, start time: {1:0.2f}s, end
        else:
            plt.plot (M[i], label="eeg signal", color="blue")
            plt.plot (x, y1, color="red")
            plt.plot (x2, y2, label="yellow box", color="red")
            plt.legend ()

        if (save):
            plt.savefig ("/home/ravi/Class/ANN/Takehome2/Output/Plots/eeg_" + str(i)
        if show:
            plt.show()

def main ():
    plot_eeg ("ALL", only_eeg=True)

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

**LSTM Code**

```
# Recurrent Neural Network

from __future__ import print_function
import sys
import pydot
sys.path.append('/home/ravi/Class/Thesis/Data/Code')
import reading_edf as data
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.utils import plot_model
#RMSE
import math
from sklearn.metrics import mean_squared_error
# Command Line Arguments
import argparse
# Load modle
from keras.models import load_model
import pickle
# Random
import random

#------------------------------
# Part 1 - Data Pre-processing
#------------------------------

# Importing the training set
def import_training_data (idea=1, num_samples=3):
    """
    Give the training data based on which idea you are implementing.
    Input:
        idea: Idea number
        Idea #1:
            - Train with normal signal and observe what happens when you show "yellow
        Idea #2:
            - You know when yellow boxes start and end. See when you give yellow box
    Output:
        Matrix of training
    """

    a = data.edf_convert()
    if (idea == 1):
```

```
        M = np.zeros ((num_samples, 100))
        for i in range (num_samples):
            montage = a.file[i][3]
            M[i] = data.edf_convert ().get_normal_signal (num_samples=1, sample_num=i

        training_set = np.array (M).reshape (-1, 1)
        points = -1

    elif (idea == 2):
        M = np.zeros ((num_samples*2, 100))
        s = []
        points = np.zeros ((num_samples * 2, 1))
        for i in range (num_samples):
            montage = a.file[i][3]
            i = i * 2
            M[i] = data.edf_convert ().get_normal_signal (num_samples=1, sample_num=i
            points[i, 0] = 1
            s.append ((M[i], points[i, 0]))
            (M[i+1], details) = data.edf_convert ().get_yellow_signals (num_samples=
            points[i+1, 0] = -1
            s.append ((M[i+1], points[i+1, 0]))

        random.shuffle (s)

        for x in s:
            M[i] = x[0]
            points[i, 0] = x[1]

        #(M, details) = data.edf_convert ().get_yellow_signals (num_samples=1)
        training_set = np.array (M).reshape (-1, 1)
        print (points)
        #points = (float(details[0][1]), float(details[0][2]))

    return (training_set, points)

# Feature Scaling
def scale_data (training_set):
    sc = MinMaxScaler(feature_range=(0, 1))
    training_set_scaled = sc.fit_transform (training_set)
    return (sc, training_set_scaled)

#Creating a datastructure with 60 timesteps (window) and 1 output
def split_data (training_set_scaled, points=-1, idea=1):
    if idea == 1:
        n_timesteps = 30
    elif idea == 2:
        n_timesteps = 100

    n_output = 1
```

```
    upper_bound = training_set_scaled.shape[0]
    X_train = []
    Y_train = []

    for i in range (n_timesteps, upper_bound):
      init  = i - n_timesteps
      X_train.append (training_set_scaled[init:i, 0])
      if idea == 1:
          Y_train.append (training_set_scaled[i, 0])
      elif idea == 2:
          if (init % 100 == 0):
              Y_train.append (1)
          else:
              Y_train.append (-1)

    X_train, Y_train = np.array (X_train), np.array (Y_train)

    #Reshaping, keras needs 3D tensor: (batch_size, timesteps, inputdim)
    X_train = X_train.reshape (X_train.shape[0], X_train.shape[1], 1)

    return (X_train, Y_train)

#------------------------------
#Part 2 - Building the RNN
#------------------------------

# Initializing the RNN
def RNN(X_train, n_output=1):
    regressor = Sequential ()

    #Adding the first LSTM layer with Dropout regularisation
    regressor.add (LSTM (units=100, return_sequences=True, input_shape=(X_train.shape
    regressor.add (Dropout (0.2))

    #Adding the second LSTM layer with Dropout regularisation
    regressor.add (LSTM (units=100, return_sequences=True))
    regressor.add (Dropout (0.2))

    #Adding the third LSTM layer with Dropout regularisation
    regressor.add (LSTM (units=100, return_sequences=True))
    regressor.add (Dropout (0.2))

    #Adding the fourth LSTM layer with Dropout regularisation
    regressor.add (LSTM (units=100))
    regressor.add (Dropout (0.2))

    #Ouput Layer
    regressor.add (Dense (n_output))
```

```
    return regressor

#Compiling the keras neural network
def compile(regressor):
    regressor.compile (optimizer='adam', loss='mean_squared_error')

# Fitting the RNN to the Training set
def train (regressor, X_train, Y_train):
    regressor.fit (X_train, Y_train, epochs=30, batch_size=32)

#------------------------------------------------------------------
#Part 3 - Making the predictions and visualising the results
#------------------------------------------------------------------

def import_test_data (sc, num, n_timesteps=100, idea=1):
    if idea==1:
        # Get EEG data
        if (num < 0):
            M = data.edf_convert ().get_normal_signal (num_samples=1, sample_num=num
            points = -1
        else:
            (M, details) = data.edf_convert ().get_yellow_signals (num_samples=1, sam
            print ("Start of abnormal activity: {}".format (details[0][1]))
            print ("End of abnormal activity: {}".format (details[0][2]))
            print ("Montage: {}".format (details[0][3]))
            points = (float (details[0][1]), float (details[0][2]))
    elif idea==2:
        if (num < 0):
            M = data.edf_convert ().get_normal_signal (num_samples=1, sample_num=num
            points = -1
        else:
            (M, details) = data.edf_convert ().get_yellow_signals (num_samples=1, sam
            print ("Start of abnormal activity: {}".format (details[0][1]))
            print ("End of abnormal activity: {}".format (details[0][2]))
            print ("Montage: {}".format (details[0][3]))
            points = (float (details[0][1]), float (details[0][2]))

    test_set = np.array(M).reshape (-1, 1)

    inputs = test_set

    # Scaled inputs and outputs
    inputs = sc.transform (inputs)

    upper_bound = inputs.shape[0]
    X_test = []
    Y_test = []

    for i in range (n_timesteps, upper_bound):
```

```
            init   = i - n_timesteps
            print (init)
            print (i)
            print (upper_bound)
            X_test.append (inputs[init:i, 0])
            if idea == 1:
                Y_test.append (inputs[i, 0])
            elif idea == 2:
                if num < 0:
                    Y_test.append (1)
                else:
                    Y_test.append (-1)

        if n_timesteps == upper_bound:
            X_test.append (inputs[0:100, 0])
            if num < 0:
              Y_test.append (1)
            else:
              Y_test.append (-1)

        X_test = np.array(X_test)
        print (X_test.shape[0])
        print (X_test.shape[1])
        X_test = X_test.reshape (X_test.shape[0], X_test.shape[1], 1)

        Y_test = np.array(Y_test).reshape(-1, 1)
        if idea == 1:
            Y_test = sc.inverse_transform (Y_test)

        return (X_test, Y_test, points)

    # Predict
    def predict (sc, regressor, X_test, idea=1):
        predicted_values = regressor.predict (X_test)
        if (idea == 1):
            predicted_values = sc.inverse_transform (predicted_values) # Inverse scaling
        return (predicted_values)

    #-------------------------------------------------------------------------
    # Part  4: Other useful features
    #-------------------------------------------------------------------------

    # Visualising the results
    def visualize (test_set, predicted_values, signal_type, points=-1, idea=1, save=False

        plt.figure ()
        if (idea == 2):
            plt.subplot (211)
        plt.plot (test_set, color='red', label='Real EEG signal')
```

```
    if (idea == 2):
        plt.subplot (212)
    plt.plot (predicted_values, color='blue', label='Fitted EEG signal')

    # Plot a yellow box
    if (points != -1):
        start = points[0]
        end = points[1]
        #start = int (float (start) * 250)
        #end = int (float (end) * 250)
        start = 0
        end = 100

        m1 = max (test_set[:, 0])
        m2 = min (test_set[:, 0])

        x2 = np.array ([start, start, end, end, start])
        y2 = np.array ([m1, m2, m2, m1, m1])

        #plt.plot (x2, y2, color="yellow")

    if (signal_type < 0):
        signal_type = "normal_" + str (-signal_type)
    else:
        signal_type = "yelow_" + str (signal_type)

    plt.title ('Real vs Predicted: ' + signal_type)
    plt.xlabel ('Time')
    plt.ylabel ('EEG signal')
    plt.legend()

    if save:
        plt.savefig ("/home/ravi/Class/ANN/Takehome2/Output/Temp4/compare_idea_" + st

    #plt.show()

# RMSE (Root Mean Squared Error)
def compute_rmse (test_set, predicted_values):
    rmse = math.sqrt (mean_squared_error (test_set, predicted_values))
    return rmse

# Analyze
def analyze (Y_test, predicted_values, signal_type, n_timesteps=60, points=-1, save=F
    if idea == 1:
        upper_bound = Y_test.shape[0]
        r = []

        n_timesteps = 30
        for i in range (0, upper_bound, n_timesteps):
```

```
            r.append (compute_rmse (Y_test[i:i+n_timesteps, 0], predicted_values[i:i+

        if plot:
            plt.title ('RMSE')
            x =  np.arange(len(r))
            plt.figure ()
            plt.plot (x, r, label="RMSE")

            # Plot the box
            if (points != -1):
                start = float (0 * 250 / n_timesteps)
                end = float ( 100 / n_timesteps)
                m1 = max (r)
                m2 = min (r)
                x2 = np.array ([start, start, end, end, start])
                y2 = np.array ([m1, m2, m2, m1, m1])

            plt.xlabel ('Time in seconds')
            plt.ylabel ('RMSE')
            plt.legend ()

            if save:
                if (signal_type < 0):
                    signal_type = "normal_" + str (-signal_type)
                else:
                    #plt.show()
                    signal_type = "yelow_" + str (signal_type)
                plt.savefig ("/home/ravi/Class/ANN/Takehome2/Output/Temp3/analysis_id

        return (sum (r)/len(r))
    else:
        return (sum(Y_test))

def compute_confusion_matrix (pos_confusion_matrix, neg_confusion_matrix, idea=1):
    if idea == 1:
        neg_max = max (neg_confusion_matrix)

        false_positive = 0;
        for c in pos_confusion_matrix:
            if c < neg_max:
                false_positive += 1

        true_negative = len (pos_confusion_matrix) - false_positive

        print ("False Positive\t|\tTrue Negative")
        print ("{}              \t|\t{}".format (false_positive, true_negative))

        neg_max = sum (neg_confusion_matrix) // len (neg_confusion_matrix)
```

```
        false_positive = 0;
        for c in pos_confusion_matrix:
            if c <= neg_max:
                false_positive += 1

        true_negative = len (pos_confusion_matrix) - false_positive

        print ("False Positive\t|\tTrue Negative")
        print ("{}              \t|\t{}".format (false_positive, true_negative))

    else:
        true_negative = 0
        true_positive = 0
        false_negative = 0
        false_positive = 0
        for c in pos_confusion_matrix:
            if c > 0:
                false_positive += 1
            else:
                true_negative += 1

        for c in neg_confusion_matrix:
            if c < 0:
                false_negative += 1
            else:
                true_positive += 1

# Save model
def my_save_model (model):
    model.save('my_model5.h5')  # creates a HDF5 file 'my_model.h5'

# Load model
def my_load_model ():
    model = load_model('my_model5.h5')
    return (model)

# Save weights
def my_save_weights (model):
    model.save_weights('my_weights5.h5')

# Load weights
def my_load_weights (model):
    model.load_weights('my_weights5.h5')

#-----------------------------------------------------
# Part 5: Call the functions
#-----------------------------------------------------

# Main function
```

```
def train_model (idea=1, load=False, save=False, num_samples=3):

    if (load == True):
        model = my_load_model ()

        # Train
        my_load_weights (model)

        # Load sc
        sc = pickle.load (open ("sc.p", "rb"))

    else:
        # Get data
        (train_data, points) = import_training_data (idea=idea, num_samples=num_samp

        # Scale data
        sc, train_data_scaled = scale_data (train_data)

        # Split data into input and output signals
        (X_train, Y_train) = split_data (train_data_scaled, idea=idea, points=points)

        # Define
        model = RNN (X_train)

        # Compile
        compile (model)

        # Train
        train (model, X_train, Y_train)

    # Save the trained model
    if (save == True):
        my_save_model (model)
        my_save_weights (model)
        pickle.dump (sc, open ("sc.p", "wb"))

    return (model, sc)

def test_model (model, sc, idea=1, save=False, plot=False):
    # Use dark background
    plt.style.use('dark_background')

    # Signal Types
    signal_type = list (range (-80, 80))

    pos_confusion_matrix = []
    neg_confusion_matrix = []

    for i in signal_type:
```

```
            # Get test data
            (X_test, Y_test, points) = import_test_data (sc, i, idea=idea)

            # Predict
            predicted_values = predict (sc, model, X_test, idea=idea)

            # Visualize
            if plot:
                visualize (Y_test, predicted_values, i, points=points, idea=idea, save=sa

            # Analyze
            max_rmse = analyze (Y_test, predicted_values, i, points=points, save=save, id

            if (i < 0):
                neg_confusion_matrix.append (max_rmse)
            else:
                pos_confusion_matrix.append (max_rmse)

        print ("-------------------------------------------")
        print ("-------------------------------------------")
        print ("Yellow box signal output: (Expecting all -1s)")
        print (pos_confusion_matrix) # Expecting all -1
        print ("-------------------------------------------")
        print ("-------------------------------------------")
        print ("Normal signal output: (Expecting all +1s)")
        print (neg_confusion_matrix) # Expecting all 1
        print ("-------------------------------------------")
        print ("-------------------------------------------")

        # Find out true negatives and false positives
        compute_confusion_matrix (pos_confusion_matrix, neg_confusion_matrix, idea=idea)

    if __name__ == "__main__":

        # Train the model
        #(model, sc) = train_model (idea=2, load=False, save=True, num_samples=100)
        (model, sc) = train_model (idea=2, load=True, save=False, num_samples=1)

        #plot_model(model, to_file='model.png', show_shapes=True)

        # Test it
        test_model (model, sc, idea=2, save=False, plot=True)
```