# AssignmentAnomalyDetection

| VERSION AUTHOR | LAST UPDATED | LANGUAGE |
| --- | --- | --- |
| Sayandeb Ghosh | 23 Sep 2018, 9:00 PM | Python 3.5 with Spark 2.1 |

# Graded Programming Assignment

In this assignment, you will implement re-use the unsupervised anomaly detection algorithm but turn it into a simpler feed forward neural network for supervised classification.

You are training the neural network from healthy and broken samples and at later stage hook it up to a message queue for real-time anomaly detection.

We've provided a skeleton for you containing all the necessary code but left out some important parts indicated with ### your code here ###

After you've completed the implementation please submit it to the autograder

# imports

try to import all necessary 3rd party packages which are not already part of the default data science experience installation

note: If you are running outside data science experience you might need a "!pip install pandas scikit-learn tensorflow"

In [1]:
```
!pip install ibmiotf
import pip

try:
    __import__('keras')
except ImportError:
    pip.main(['install', 'keras'])

try:
    __import__('ibmiotf')
except ImportError:
    pip.main(['install', 'ibmiotf'])
```

```
Requirement already satisfied: ibmiotf in /gpfs/global_fs01/sym_shared/
YPProdSpark/user/sc3a-964534234723e9-4e882cbe4448/.local/lib/python3.5/
site-packages (0.4.0)
Requirement already satisfied: paho-mqtt>=1.3.1 in /gpfs/global_fs01/sy
m_shared/YPProdSpark/user/sc3a-964534234723e9-4e882cbe4448/.local/lib/p
ython3.5/site-packages (from ibmiotf) (1.4.0)
Requirement already satisfied: requests>=2.18.4 in /usr/local/src/conda
3_runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-packages
(from ibmiotf) (2.18.4)
Requirement already satisfied: pytz>=2017.3 in /usr/local/src/conda3_ru
ntime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-packages (fro
m ibmiotf) (2018.4)
Requirement already satisfied: requests-toolbelt>=0.8.0 in /gpfs/global
_fs01/sym_shared/YPProdSpark/user/sc3a-964534234723e9-4e882cbe4448/.loc
al/lib/python3.5/site-packages (from ibmiotf) (0.8.0)
Requirement already satisfied: iso8601>=0.1.12 in /usr/local/src/conda3
_runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-packages
(from ibmiotf) (0.1.12)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/src/
conda3_runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-pack
```

```
ages (from requests>=2.18.4->ibmiotf) (3.0.4)
Requirement already satisfied: idna<2.7,>=2.5 in /usr/local/src/conda3_
runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-packages (f
rom requests>=2.18.4->ibmiotf) (2.6)
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /usr/local/src/
conda3_runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-pack
ages (from requests>=2.18.4->ibmiotf) (1.22)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/src/con
da3_runtime.v44/home/envs/DSX-Python35-Spark/lib/python3.5/site-package
s (from requests>=2.18.4->ibmiotf) (2018.8.24)

Using TensorFlow backend.
```

Now we import all the rest

In [2]:
```
import numpy as np
from numpy import concatenate
from matplotlib import pyplot
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
import sklearn
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import LSTM
from keras.callbacks import Callback
from keras.models import Sequential
from keras.layers import LSTM, Dense, Activation
import pickle
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import ibmiotf.application
import sys
from queue import Queue
import pandas as pd
import json
%matplotlib inline
```

We grab the files necessary for taining. Those are sampled from the lorenz attractor model implemented in NodeRED. Those are two serialized pickle numpy arrays

In [3]:
```
!rm watsoniotp.*
!wget https://raw.githubusercontent.com/romeokienzler/developerWorks/ma
ster/lorenzattractor/watsoniotp.healthy.phase_aligned.pickle
!wget https://raw.githubusercontent.com/romeokienzler/developerWorks/ma
ster/lorenzattractor/watsoniotp.broken.phase_aligned.pickle
!mv watsoniotp.healthy.phase_aligned.pickle watsoniotp.healthy.pickle
!mv watsoniotp.broken.phase_aligned.pickle watsoniotp.broken.pickle
```

```
--2018-09-23 13:19:20--  https://raw.githubusercontent.com/romeokienzle
r/developerWorks/master/lorenzattractor/watsoniotp.healthy.phase_aligne
d.pickle
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.
101.48.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|15
1.101.48.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
```

```
HTTP request sent, awaiting response... 200 OK
Length: 194639 (190K) [text/plain]
Saving to: ‘watsoniotp.healthy.phase_aligned.pickle’

100%[======================================>] 194,639     --.-K/s   in
0.1s

2018-09-23 13:19:20 (1.38 MB/s) - ‘watsoniotp.healthy.phase_aligned.pic
kle’ saved [194639/194639]

--2018-09-23 13:19:21--  https://raw.githubusercontent.com/romeokienzle
r/developerWorks/master/lorenzattractor/watsoniotp.broken.phase_aligne
d.pickle
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.
101.48.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|15
1.101.48.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 185401 (181K) [text/plain]
Saving to: ‘watsoniotp.broken.phase_aligned.pickle’

100%[======================================>] 185,401     --.-K/s   in
0.006s

2018-09-23 13:19:21 (31.2 MB/s) - ‘watsoniotp.broken.phase_aligned.pick
le’ saved [185401/185401]
```

De-serialize the numpy array containing the training data

In [4]:
```
data_healthy = pickle.load(open('watsoniotp.healthy.pickle', 'rb'), enc
oding='latin1')
data_broken = pickle.load(open('watsoniotp.broken.pickle', 'rb'), encod
ing='latin1')
```

Reshape to three columns and 3000 rows. In other words three vibration sensor axes and 3000 samples

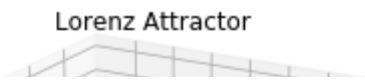
In [5]:
```
data_healthy = data_healthy.reshape(3000,3)
data_broken = data_broken.reshape(3000,3)
```

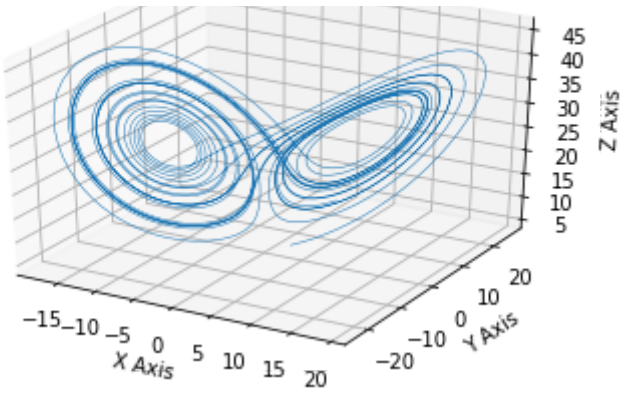
Since this data is sampled from the Lorenz Attractor Model, let's plot it with a phase lot to get the typical 2-eyed plot. First for the healthy data

In [6]:
```
fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(data_healthy[:,0], data_healthy[:,1], data_healthy[:,2],lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")
```

Out[6]:
```
Text(0.5,0.92,'Lorenz Attractor')
```



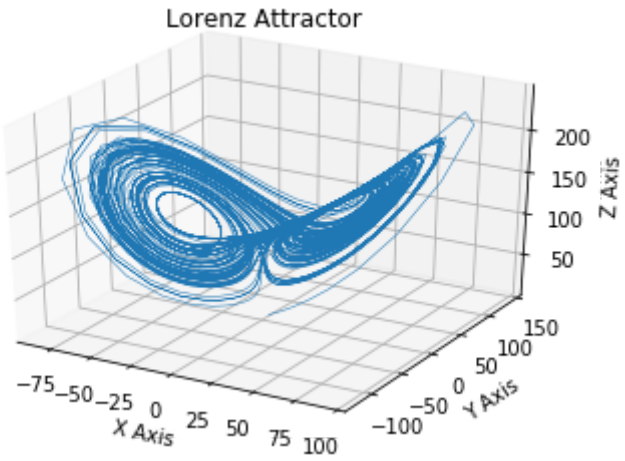



Then for the broken one

```
In [7]: fig = plt.figure()
        ax = fig.gca(projection='3d')

        ax.plot(data_broken[:,0], data_broken[:,1], data_broken[:,2],lw=0.5)
        ax.set_xlabel("X Axis")
        ax.set_ylabel("Y Axis")
        ax.set_zlabel("Z Axis")
        ax.set_title("Lorenz Attractor")
```

```
Out[7]: Text(0.5,0.92,'Lorenz Attractor')
```




In the previous examples, we fed the raw data into an LSTM. Now we want to use an ordinary feed-forward network. So we need to do some pre-processing of this time series data

A widely-used method in traditional data science and signal processing is called Discrete Fourier Transformation. This algorithm transforms from the time to the frequency domain, or in other words, it returns the frequency spectrum of the signals.

The most widely used implementation of the transformation is called FFT, which stands for Fast Fourier Transformation, let's run it and see what it returns

```
In [8]: data_healthy_fft = np.fft.fft(data_healthy)
        data_broken_fft = np.fft.fft(data_broken)
```

Let's first have a look at the shape and contents of the arrays.

```
In [9]: pprint (data_healthy_fft.shape)
```

```
print (data_healthy_fft)
```

```
(3000, 3)
[[  9.42619266+0.j          -1.59309633+0.51569445j
   -1.59309633-0.51569445j]
 [  9.90369920+0.j          -1.67640640+0.15919595j
   -1.67640640-0.15919595j]
 [ 10.43779318+0.j          -1.75331404-0.20569613j
   -1.75331404+0.20569613j]
 ...,
 [ 24.96901270+0.j          -5.47609496+2.75768205j
   -5.47609496-2.75768205j]
 [ 25.97317526+0.j          -5.51210266+2.35412545j
   -5.51210266-2.35412545j]
 [ 27.08054148+0.j          -5.56743205+1.96286582j
   -5.56743205-1.96286582j]]
```

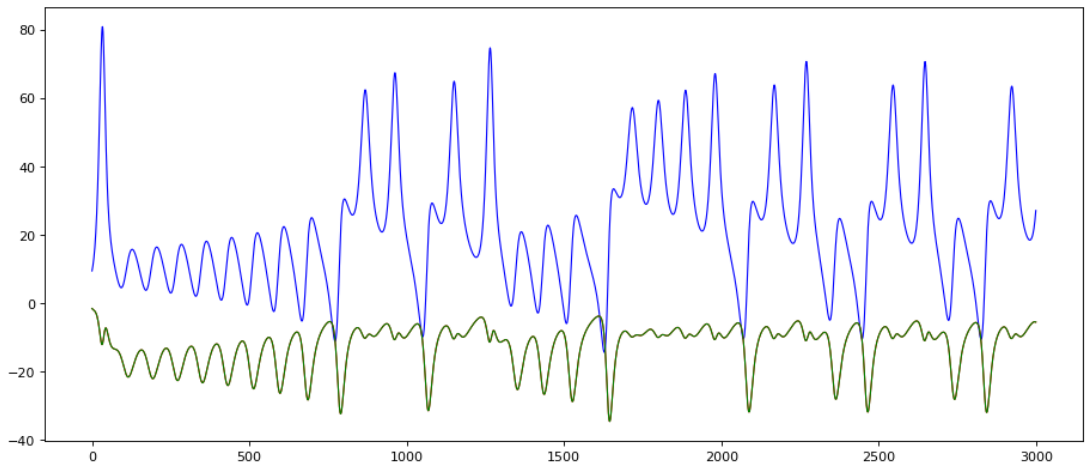
First, we notice that the shape is the same as the input data. So if we have 3000 samples, we get back 3000 spectrum values, or in other words 3000 frequency bands with the intensities.

The second thing we notice is that the data type of the array entries is not float anymore, it is complex. So those are not complex numbers, it is just a means for the algorithm the return two different frequency compositions in one go. The real part returns a sine decomposition and the imaginary part a cosine. We will ignore the cosine part in this example since it turns out that the sine part already gives us enough information to implement a good classifier.

But first let's plot the two arrays to get an idea how a healthy and broken frequency spectrum differ

In [10]:
```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w'
, edgecolor='k')
size = len(data_healthy_fft)
ax.plot(range(0,size), data_healthy_fft[:,0].real, '-', color='blue', a
nimated = True, linewidth=1)
ax.plot(range(0,size), data_healthy_fft[:,1].real, '-', color='red', an
imated = True, linewidth=1)
ax.plot(range(0,size), data_healthy_fft[:,2].real, '-', color='green',
animated = True, linewidth=1)
```

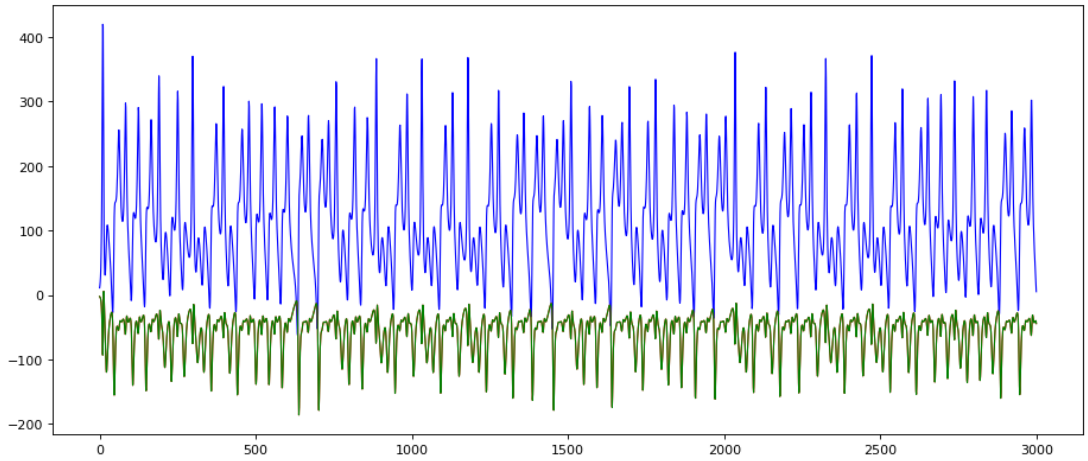
Out[10]:
```
[<matplotlib.lines.Line2D at 0x7fc4e60ad6a0>]
```




In [11]:
```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w'
, edgecolor='k')
size = len(data_healthy_fft)
ax.plot(range(0,size), data_broken_fft[:,0].real, '-', color='blue', an
```

```
imated = True, linewidth=1)
ax.plot(range(0,size), data_broken_fft[:,1].real, '-', color='red', ani
mated = True, linewidth=1)
ax.plot(range(0,size), data_broken_fft[:,2].real, '-', color='green', a
nimated = True, linewidth=1)
```

Out[11]: [<matplotlib.lines.Line2D at 0x7fc4e6075828>]




So, what we've been doing is so called feature transformation step. We've transformed the data set in a way that our machine learning algorithm – a deep feed forward neural network implemented as binary classifier – works better. So now let's scale the data to a 0..1

In [12]:
```
def scaleData(data):
    # normalize features
    scaler = MinMaxScaler(feature_range=(0, 1))
    return scaler.fit_transform(data)
```

And please don't worry about the warnings. As explained before we don't need the imaginary part of the FFT

In [13]:
```
data_healthy_scaled = scaleData(data_healthy_fft)
data_broken_scaled = scaleData(data_broken_fft)
```

```
/usr/local/src/conda3_runtime/home/envs/DSX-Python35-Spark/lib/python3.
5/site-packages/sklearn/utils/validation.py:433: ComplexWarning: Castin
g complex values to real discards the imaginary part
  array = np.array(array, dtype=dtype, order=order, copy=copy)
/usr/local/src/conda3_runtime/home/envs/DSX-Python35-Spark/lib/python3.
5/site-packages/sklearn/utils/validation.py:475: DataConversionWarning:
Data with input dtype complex128 was converted to float64 by MinMaxScal
er.
  warnings.warn(msg, DataConversionWarning)
```

Now we reshape again to have three examples (rows) and 3000 features (columns). It's important that you understand this. We have turned our initial data set which containd 3 columns (dimensions) of 3000 samples. Since we applied FFT on each column we've obtained 3000 spectrum values for each of the 3 three columns. We are now using each column with the 3000 spectrum values as one row (training example) and each of the 3000 spectrum values becomes a column (or feature) in the training data set

In [14]:
```
data_healthy_scaled.shape = (3, 3000)
data_broken_scaled.shape = (3, 3000)
```

```
data_broken_scaled.shape = (3, 3000)
```

# Start of Assignment

The first thing we need to do is to install a little helper library for submitting the solutions to the coursera grader:

In [15]:
```
!rm -f rklib.py
!wget https://raw.githubusercontent.com/romeokienzler/developerWorks/master/coursera/ai/rklib.py
```

```
--2018-09-23 13:19:22--  https://raw.githubusercontent.com/romeokienzler/developerWorks/master/coursera/ai/rklib.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.48.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.48.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2029 (2.0K) [text/plain]
Saving to: ‘rklib.py’

100%[======================================>] 2,029       --.-K/s   in 0s

2018-09-23 13:19:22 (19.4 MB/s) - ‘rklib.py’ saved [2029/2029]
```

Please specify you email address you are using with cousera here:

In [16]:
```
from rklib import submit, submitAll
key = "4vkB9vnrEee8zg4u9l99rA"
all_parts = ["O5cR9","0dXlH","ZzEP8"]

email = "sayandeb@gmail.com"
```

# Task

Given, the explanation above, please fill in the following two constants in order to make the neural network work properly

In [17]:
```
#### your code here ###
dim = 3000
samples = 3
```

## Submission

Now it’s time to submit your first solution. Please make sure that the secret variable contains a valid submission token. You can obtain it from the courser web page of the course using the grader section of this assignment.

In [18]:
```
part = "O5cR9"
```

```
secret = "ARGwkQYMkpNtz6z3"

submitAll(email, secret, key, dict((p, json.dumps({}) if p != part else
 json.dumps({"dim": dim, "samples": samples})) for p in all_parts))
```

```
Submission successful, please check on the coursera grader page for the
status
```

To observe how training works we just print the loss during training

In [19]:
```
class LossHistory(Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        sys.stdout.write(str(logs.get('loss'))+str(', '))
        sys.stdout.flush()
        self.losses.append(logs.get('loss'))

lr = LossHistory()
```

# Task

Please fill in the following constants to properly configure the neural network. For some of them you have to find out the precise value, for others you can try and see how the neural network is performing at a later stage. The grader only looks at the values which need to be precise

In [20]:
```
number_of_neurons_layer1 = 3000
number_of_neurons_layer2 = 3000
number_of_neurons_layer3 = 3000
number_of_epochs = 100
```

## Submission

Please submit your constants to the grader

In [21]:
```
parts_data = {}
parts_data["0dXlH"] = json.dumps({"number_of_neurons_layer1": number_of
_neurons_layer1, "number_of_neurons_layer2": number_of_neurons_layer2,
"number_of_neurons_layer3": number_of_neurons_layer3, "number_of_epoch
s": number_of_epochs})
parts_data["O5cR9"] = json.dumps({"dim": dim, "samples": samples})
parts_data["ZzEP8"] = None


secret = "ARGwkQYMkpNtz6z3"


submitAll(email, secret, key, parts_data)
```

```
Submission successful, please check on the coursera grader page for the
status
```

# Task

Now it’s time to create the model. Please fill in the placeholders. Please note since this is only a toy example, re don't use a separate corpus for training and testing. Just use the same data for fitting and scoring

In [22]:
```
# design network
from keras import optimizers
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)

model = Sequential()
model.add(Dense(number_of_neurons_layer1,input_shape=(dim, ), activatio
n='relu'))
model.add(Dense(number_of_neurons_layer2, activation='relu'))
model.add(Dense(number_of_neurons_layer3, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer=sgd)

def train(data,label):
    model.fit(data, label, epochs=number_of_epochs, batch_size=72, vali
dation_data=(data, label), verbose=0, shuffle=True,callbacks=[lr])

def score(data):
    return model.predict(data)
```

In [23]:
```
#some learners constantly reported 502 errors in Watson Studio.
#This is due to the limited resources in the free tier and the heavy re
source consumption of Keras.
#This is a workaround to limit resource consumption

from keras import backend as K

K.set_session(K.tf.Session(config=K.tf.ConfigProto(intra_op_parallelism
_threads=1, inter_op_parallelism_threads=1)))
```

We prepare the training data by concatenating a label “0” for the broken and a label “1” for the healthy data. Finally we union the two data sets together

In [24]:
```
label_healthy = np.repeat(1,3)
label_healthy.shape = (3,1)
label_broken = np.repeat(0,3)
label_broken.shape = (3,1)

train_healthy = np.hstack((data_healthy_scaled,label_healthy))
train_broken = np.hstack((data_broken_scaled,label_broken))
train_both = np.vstack((train_healthy,train_broken))
```

Let’s have a look at the two training sets for broken and healthy and at the union of them. Note that the last column is the label

In [25]:
```
pd.DataFrame(train_healthy)
```

Out[25]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.250133 | 1.000000 | 1.000000 | 0.255146 | 0.997476 | 0.997476 | 0.260753 | 0.995146 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1** | 0.352582 | 0.754714 | 0.754714 | 0.346867 | 0.758104 | 0.758104 | 0.341282 | 0.761698 |
| **2** | 0.506425 | 0.759679 | 0.759679 | 0.492759 | 0.755800 | 0.755800 | 0.480019 | 0.752204 |

3 rows × 3001 columns

In [26]:
```
pd.DataFrame(train_broken)
```

Out[26]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.151988 | 0.957909 | 0.957909 | 0.159267 | 0.954314 | 0.954314 | 0.170598 | 0.949721 |
| **1** | 0.144590 | 0.666782 | 0.666782 | 0.150818 | 0.606449 | 0.606449 | 0.174423 | 0.524868 |
| **2** | 0.491165 | 0.739152 | 0.739152 | 0.539779 | 0.720249 | 0.720249 | 0.596680 | 0.697156 |

3 rows × 3001 columns

In [27]:
```
pd.DataFrame(train_both)
```

Out[27]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.250133 | 1.000000 | 1.000000 | 0.255146 | 0.997476 | 0.997476 | 0.260753 | 0.995146 |
| **1** | 0.352582 | 0.754714 | 0.754714 | 0.346867 | 0.758104 | 0.758104 | 0.341282 | 0.761698 |
| **2** | 0.506425 | 0.759679 | 0.759679 | 0.492759 | 0.755800 | 0.755800 | 0.480019 | 0.752204 |
| **3** | 0.151988 | 0.957909 | 0.957909 | 0.159267 | 0.954314 | 0.954314 | 0.170598 | 0.949721 |
| **4** | 0.144590 | 0.666782 | 0.666782 | 0.150818 | 0.606449 | 0.606449 | 0.174423 | 0.524868 |
| **5** | 0.491165 | 0.739152 | 0.739152 | 0.539779 | 0.720249 | 0.720249 | 0.596680 | 0.697156 |

6 rows × 3001 columns

So those are frequency bands. Notice that although many frequency bands are having nearly the same energy, the neural network algorithm still can work those out which are significantly different.

# Task

Now it's time to do the training. Please provide the first 3000 columns of the array as the 1st parameter and column number 3000 containing the label as 2nd parameter. Please use the python array slicing syntax to obtain those.

The following link tells you more about the numpy array slicing syntax https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html)

In [28]:
```
features = train_both[:,:3000]
labels = train_both[:,:-1]
```

Now it's time to do the training. You should see the loss trajectory go down, we will also plot it later. Note:

We also could use TensorBoard for this but for this simple scenario we skip it. In some rare cases training doesn't converge simply because random initialization of the weights caused gradient descent to start at a sub-optimal spot on the cost hyperplane. Just recreate the model (the cell which contains *model = Sequential()*) and re-run all subsequent steps and train again

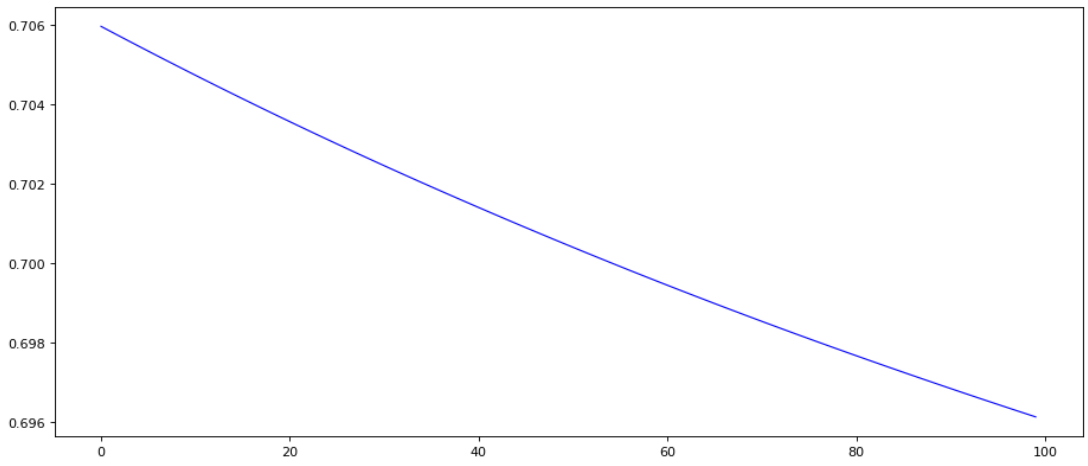
In [29]:
```
train(features,labels)
```

```
0.705961, 0.705835, 0.70571, 0.705585, 0.705461, 0.705338, 0.705216, 0.
705094, 0.704972, 0.704851, 0.704731, 0.704612, 0.704493, 0.704375, 0.7
04257, 0.70414, 0.704023, 0.703907, 0.703792, 0.703678, 0.703564, 0.703
45, 0.703338, 0.703226, 0.703114, 0.703003, 0.702893, 0.702783, 0.70267
3, 0.702565, 0.702457, 0.702349, 0.702242, 0.702136, 0.70203, 0.701925,
0.70182, 0.701716, 0.701612, 0.701509, 0.701406, 0.701304, 0.701202, 0.
701101, 0.701, 0.700899, 0.7008, 0.7007, 0.700601, 0.700503, 0.700405,
0.700307, 0.70021, 0.700113, 0.700017, 0.699921, 0.699826, 0.699731, 0.
699637, 0.699543, 0.69945, 0.699357, 0.699265, 0.699173, 0.699082, 0.69
8991, 0.6989, 0.69881, 0.698721, 0.698631, 0.698543, 0.698454, 0.69836
6, 0.698279, 0.698192, 0.698105, 0.698019, 0.697933, 0.697848, 0.69776
3, 0.697679, 0.697594, 0.697511, 0.697427, 0.697344, 0.697261, 0.69717
9, 0.697097, 0.697015, 0.696934, 0.696853, 0.696773, 0.696692, 0.69661
3, 0.696533, 0.696454, 0.696375, 0.696297, 0.696218, 0.69614,
```

Let's plot the losses

In [30]:
```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w'
, edgecolor='k')
size = len(lr.losses)
ax.plot(range(0,size), lr.losses, '-', color='blue', animated = True, l
inewidth=1)
```

Out[30]:
```
[<matplotlib.lines.Line2D at 0x7fc4e4807fd0>]
```




Now let's examine whether we are getting good results. Note: best practice is to use a training and a test data set for this which we've omitted here for simplicity

In [31]:
```
score(data_healthy_scaled)
```

Out[31]:
```
array([[ 0.4319531 ,  0.54440451,  0.53071636, ...,  0.56813502,
         0.56232989,  0.63781351],
       [ 0.42009684,  0.48979682,  0.48394272, ...,  0.53854758,
         0.57865644,  0.69650954],
       [ 0.47152919,  0.49487889,  0.49914151, ...,  0.53473109,
         0.54538131,  0.62711018]], dtype=float32)
```

In [32]:
```
score(data_broken_scaled)
```

Out[32]:
```
array([[ 0.45530191,  0.51898301,  0.49370205, ...,  0.54086167,
         0.57359928,  0.64333034],
       [ 0.44157276,  0.4968217 ,  0.47782418, ...,  0.54605198,
         0.59635472,  0.65362787],
       [ 0.44114637,  0.48349559,  0.50071597, ...,  0.56005901,
         0.54205269,  0.62277263]], dtype=float32)
```

# Task

Now it's time to hook this up to the message queue for real-time data analysis. Please provide your credentials here. You've learned how to obtained them in the 1st week of the course in the lecture called "Setup the NodeRED Boilerplate", at the end of the video. Of course, you need to install your own instance in the IBM Cloud.

IMPORTANT NOTE: In case you haven't setup the TestData Generator, please do so. You can watch me doing this in the video "Setup the NodeRED Boilerplate" of Week 1. The important steps are shown around 2 minutes 30 seconds.

If you want to learn more about the physical model and the simulator as a whole, I've documented this here:

https://www.ibm.com/developerworks/analytics/library/iot-deep-learning-anomaly-detection-2/index.html?ca=drs- (https://www.ibm.com/developerworks/analytics/library/iot-deep-learning-anomaly-detection-2/index.html?ca=drs-)

BTW: This article is part of a 5 part series on anomaly detection I've written last year :)

If you just need the JSON of the NodeRED flow, you can find it here:

https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/simulatorflow.jso
(https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/simulatorflow.jso

I'm deeply thankful to Nicole Finnie, one of the learners, to point out this missing information.

In [33]:
```
options = {"org": "a4j8lg", "id": "a2g6k39sl6r5", "auth-method": "apike
y", "auth-key": "a-a4j8lg-112asckaqr", "auth-token": "b9KN0uoRqvw5MT@JQ
v"}
client = ibmiotf.application.Client(options)
client.connect()
```

Let's create a python queue for our data

In [34]:
```
q = Queue(7000)
```

To subscribe to real time data, we need to specify a callback handler where we extract and reformat the data, register the callback handler to the MQTT client and select the type of events we want to subscribe to

In [35]:
```
def myEventCallback(event):
```

```
In [35]: def myEventCallback(event):
             sample = event.data
             point = [sample["x"], sample["y"],sample["z"]]
             q.put(point)

         client.deviceEventCallback = myEventCallback
         client.subscribeToDeviceEvents("0.16.2", "lorenz", "osc")
```

```
2018-09-23 13:20:28,825   ibmiotf.application.Client  INFO    Connected
successfully: a:a4j8lg:a2g6k39sl6r5
```

```
Out[35]: 1
```

Before we can call the neural network scoring (or predicting) function we need to pre-proceed the raw data in the same way as we have done for the static data we've used for training. Note that we take the average score over the three samples. Finally, we push the result back the message queue.

## Submission

In case you feel confident that everything works as it should you can set *submit_work* to True and again please make sure that the secret variable contains a valid submission token. Please make sure to re-run the whole scenario end-2-end from the TestDataGenerator in NodeRED so that the grader can pick up your result based on the live stream of data.