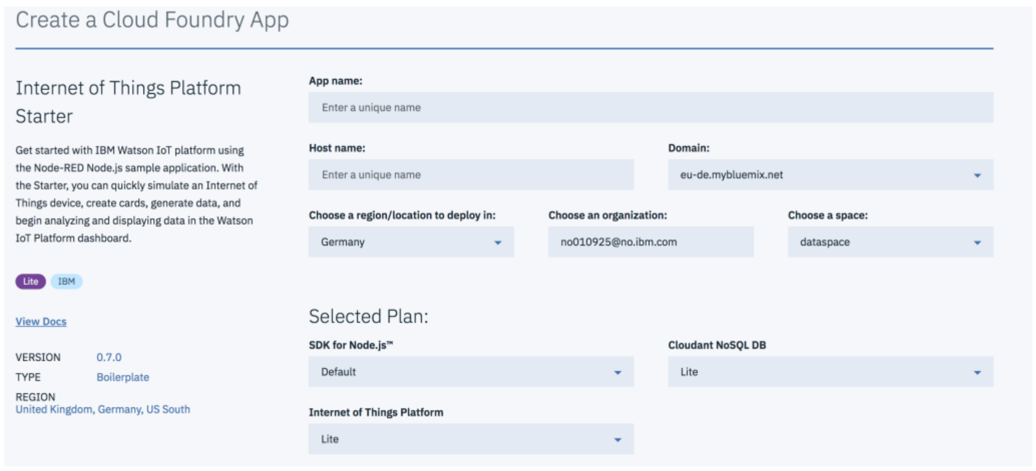
Lab 2 & 3

Creating the test data simulator

You can get the simulator up and running in four main steps:

1. **Log in to IBM Cloud (https://bluemix.net)**
2. **Go to the Catalog -> Boilerplates and select the Internet of Things Platform Starter**
3. **Fill in app name and select UK or Germany as location**



Create a Cloud Foundry App

Internet of Things Platform Starter

Get started with IBM Watson IoT platform using the Node-RED Node.js sample application. With the Starter, you can quickly simulate an Internet of Things device, create cards, generate data, and begin analyzing and displaying data in the Watson IoT Platform dashboard.

Lite IBM

View Docs

VERSION 0.7.0
TYPE Boilerplate
REGION United Kingdom, Germany, US South

App name: Enter a unique name

Host name: Enter a unique name

Domain: eu-de.mybluemix.net

Choose a region/location to deploy in: Germany

Choose an organization: no010925@no.ibm.com

Choose a space: dataspace

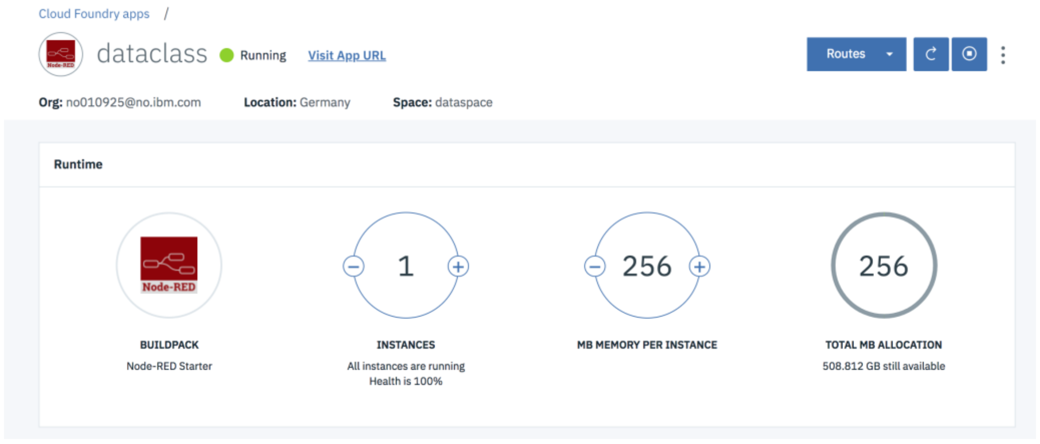
Selected Plan:

SDK for Node.js™: Default

Cloudant NoSQL DB: Lite

Internet of Things Platform: Lite

4. **Press create and wait until the environment gets deployed.**



Cloud Foundry apps /

dataclass Running Visit App URL

Org: no010925@no.ibm.com Location: Germany Space: dataspace

Runtime

BUILDPACK: Node-RED Starter

INSTANCES: 1 — All instances are running, Health is 100%

MB MEMORY PER INSTANCE: 256

TOTAL MB ALLOCATION: 256 — 508.812 GB still available

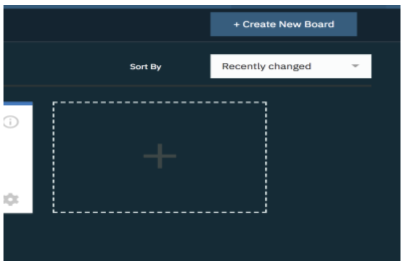
5. **Press the Visit App URL link!**
6. **After optionally setting up a user and a password go to the Flow Editor**
7. **From the hamburger menu select import from clipboard**
8. **Copy the json flow from this url: https://github.com/ethilesen/IoT-og-Deep-Learning-Workshop-med-IBM/blob/master/node-red-flow-iot.json**
9. **Press deploy and look in the debug tab that the flow is working.**

Create a dashboard to view the data…

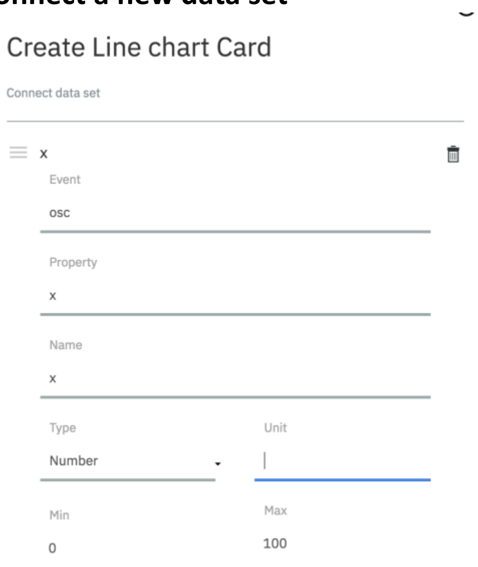
Her må det definers en device type før du kan lage et chart.

1. **Go back to your IBM Cloud Dashboard and select the Internet of Things Platform you created earlier**
2. **Under manage, launch the dashboard.**

3. **Create a new Board..**



4. **Open that Board and add a new card**
5. **Select line chart**
6. **Select your device (lorenz)**
7. **Connect a new data set**



Create Line chart Card

Connect data set

x

Event: osc

Property: x

Name: x

Type: Number

Unit:

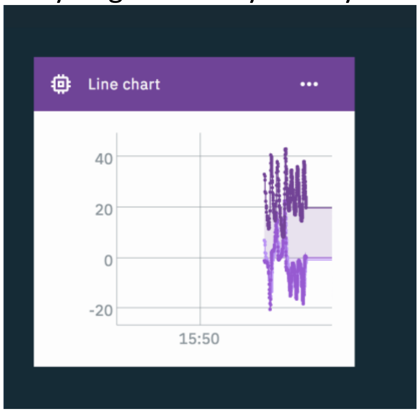
Min: 0

Max: 100

Repeat for y and z

You can have all x,y,z on the same chart or separate – up to you.

Verify that you get data – you may need to hit reset in the node-red flow…



Setting up your Watson Studio development environment

Before we talk about the deep learning use case, spend some time setting up your development environment. We use a Jupyter Notebook running inside Watson Studio.
We'll start with a little example of two "pickled" data sets (the pickle interface is the serialization and deserialization framework in Python) – one containing healthy data and one containing broken data to develop the neural network.
Afterward we'll turn this notebook into a real-time anomaly detector accessing data directly from an MQTT topic by using the IBM Watson IoT Platform.
1. Log in to https://eu-gb.dataplatform.ibm.com.
2. Go into your project if you have one else create a new one..
3. In you project under Assets -> create a new Jupyter notebook
4. Give it a name and select "From Url" and paste in this url in the Notebook URL field: https://github.com/ethilesen/IoT-og-Deep-Learning-Workshop-med-IBM/blob/master/innovasjonsloftet.ipynb

5. Select a free runtime…
6. Create notebook

This notebook are ready to run but first let's walk trough the code

1. Install and import the dependencies
Initially, we make sure that all dependencies are installed.

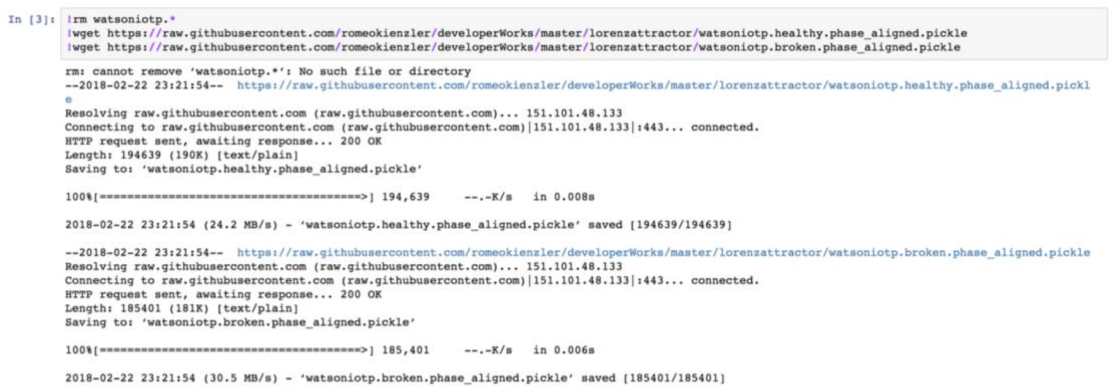
```
import pip

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

try:
    __import__('h5py')
except ImportError:
    pip.main(['install', 'h5py'])

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

Next, we review the notebook and the list of dependencies. (**Note:** Keras is already preinstalled in Watson Studio.)

```
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.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
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
from Queue import Queue
%matplotlib inline
```

2. Download broken and healthy data

Next, we need to download two data files that contain some samples of broken and healthy data.



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

rm: cannot remove 'watsoniotp.*': No such file or directory
--2018-02-22 23:21:54--  https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/watsoniotp.healthy.phase_aligned.pickl
e
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: 194639 (190K) [text/plain]
Saving to: 'watsoniotp.healthy.phase_aligned.pickle'

100%[======================================>] 194,639     --.-K/s   in 0.008s

2018-02-22 23:21:54 (24.2 MB/s) - 'watsoniotp.healthy.phase_aligned.pickle' saved [194639/194639]

--2018-02-22 23:21:54--  https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/watsoniotp.broken.phase_aligned.pickle
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: 185401 (181K) [text/plain]
Saving to: 'watsoniotp.broken.phase_aligned.pickle'

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

2018-02-22 23:21:54 (30.5 MB/s) - 'watsoniotp.broken.phase_aligned.pickle' saved [185401/185401]
```

```
!rm watsoniotp.*
!wget https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/
watsoniotp.healthy.phase_aligned.pickle
!wget https://raw.githubusercontent.com/romeokienzler/developerWorks/master/lorenzattractor/
watsoniotp.broken.phase_aligned.pickle
```

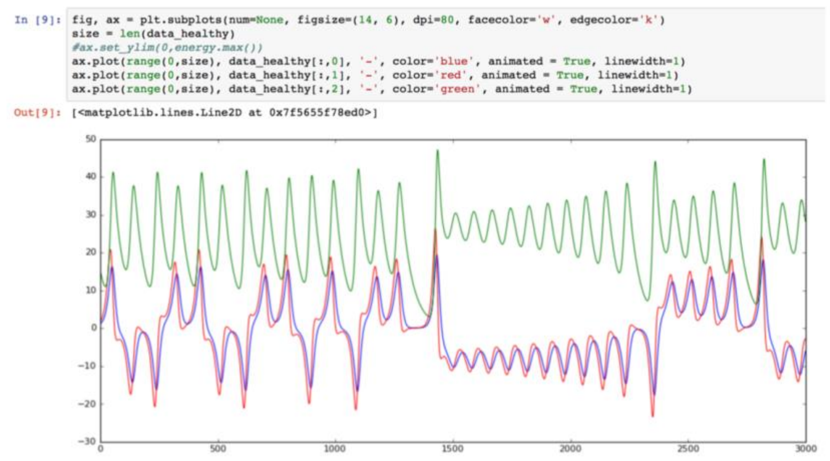
3. Deserialize the two numpy arrays

The two data files are serialized numpy arrays using the Python pickle library. We need to de- serialize (rematerialize) those two arrays from the files.

```
data_healthy = data_healthy.reshape(3000,3)
data_broken = data_broken.reshape(3000,3)
```

5. Visually inspect the data

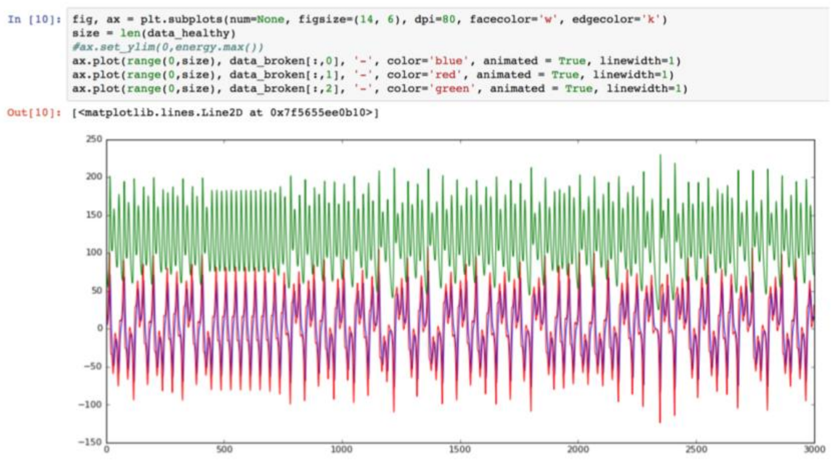
First, we have a look at the healthy data. Notice that while this system oscillates between two semi-stable states, it is hard to identify any regular patterns.



```
In [9]: fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
        size = len(data_healthy)
        #ax.set_ylim(0,energy.max())
        ax.plot(range(0,size), data_healthy[:,0], '-', color='blue', animated = True, linewidth=1)
        ax.plot(range(0,size), data_healthy[:,1], '-', color='red', animated = True, linewidth=1)
        ax.plot(range(0,size), data_healthy[:,2], '-', color='green', animated = True, linewidth=1)
Out[9]: [<matplotlib.lines.Line2D at 0x7f5655f78ed0>]
```

```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
size = len(data_healthy)
#ax.set_ylim(0,energy.max())
ax.plot(range(0,size), data_healthy[:,0], '-', color='blue', animated = True, linewidth=1)
ax.plot(range(0,size), data_healthy[:,1], '-', color='red', animated = True, linewidth=1)
ax.plot(range(0,size), data_healthy[:,2], '-', color='green', animated = True, linewidth=1)
```

Next, run the cell look at the same chart after we've switched the test data generator to a broken state. The obvious result is that we see much more energy in the system. The peaks are exceeding 200 in contrast to the healthy state which never went over 50. Also, in my opinion, the frequency content of the second signal is higher.



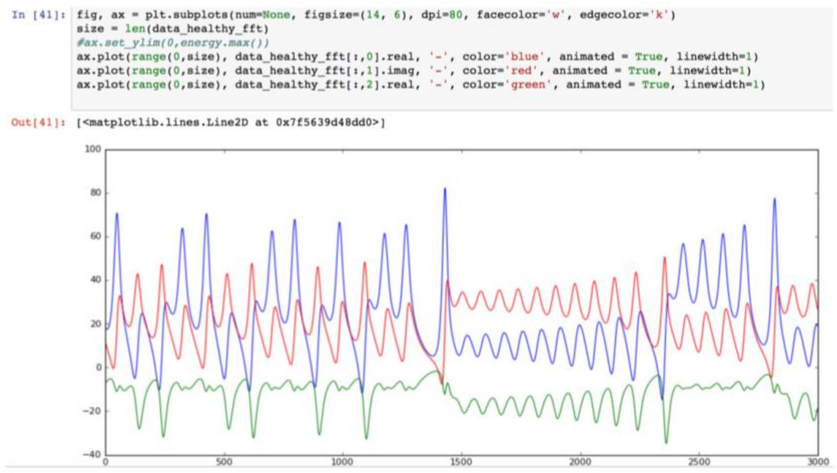
```
In [10]: fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
         size = len(data_healthy)
         #ax.set_ylim(0,energy.max())
         ax.plot(range(0,size), data_broken[:,0], '-', color='blue', animated = True, linewidth=1)
         ax.plot(range(0,size), data_broken[:,1], '-', color='red', animated = True, linewidth=1)
         ax.plot(range(0,size), data_broken[:,2], '-', color='green', animated = True, linewidth=1)
Out[10]: [<matplotlib.lines.Line2D at 0x7f5655ee0b10>]
```

```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
size = len(data_healthy)
#ax.set_ylim(0,energy.max())
ax.plot(range(0,size), data_broken[:,0], '-', color='blue', animated = True, linewidth=1)
ax.plot(range(0,size), data_broken[:,1], '-', color='red', animated = True, linewidth=1)
ax.plot(range(0,size), data_broken[:,2], '-', color='green', animated = True, linewidth=1)
```

Let's confirm the frequency of the second signal is higher by transforming the signal from the time to the frequency domain.

```
data_healthy_fft = np.fft.fft(data_healthy)
data_broken_fft = np.fft.fft(data_broken)
```

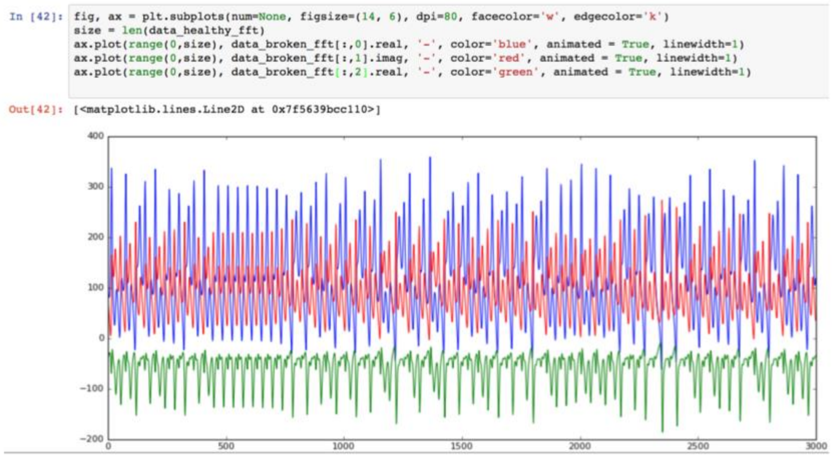
The chart now contains the frequencies of the healthy signal.



```
In [41]: fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
         size = len(data_healthy_fft)
         #ax.set_ylim(0,energy.max())
         ax.plot(range(0,size), data_healthy_fft[:,0].real, '-', color='blue', animated = True, linewidth=1)
         ax.plot(range(0,size), data_healthy_fft[:,1].imag, '-', color='red', animated = True, linewidth=1)
         ax.plot(range(0,size), data_healthy_fft[:,2].real, '-', color='green', animated = True, linewidth=1)

Out[41]: [<matplotlib.lines.Line2D at 0x7f5639d48dd0>]
```

```
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', animated = True, linewidth=1)
ax.plot(range(0,size), data_healthy_fft[:,1].imag, '-', color='red', animated = True, linewidth=1)
ax.plot(range(0,size), data_healthy_fft[:,2].real, '-', color='green', animated = True, linewidth=1)
```

Note: We are plotting the imaginary part of the red dimension to see three lines because two dimensions on this dataset are completely overlapping in frequency and the real part is zero. Remember, the way FFT (fast Fournier transform) works is retuning the sine components in the real domain and the cosine components in the imaginary domain. Just a hack mathematicians use to return a tuple of vectors.
Let's contrast this healthy data with the broken signal. As expected, there are a lot more frequencies present in the broken signal.



```
In [42]: 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', animated = True, linewidth=1)
         ax.plot(range(0,size), data_broken_fft[:,1].imag, '-', color='red', animated = True, linewidth=1)
         ax.plot(range(0,size), data_broken_fft[:,2].real, '-', color='green', animated = True, linewidth=1)

Out[42]: [<matplotlib.lines.Line2D at 0x7f5639bcc110>]
```

```
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', animated = True, linewidth=1)
ax.plot(range(0,size), data_broken_fft[:,1].imag, '-', color='red', animated = True, linewidth=1)
ax.plot(range(0,size), data_broken_fft[:,2].real, '-', color='green', animated = True, linewidth=1)
```

6. Create unsupervised machine learning

We now have enough evidence to construct an anomaly detector based on *supervised* machine learning (with a state-of-the-art model like a gradient boosted tree). But we want *unsupervised* machine learning because we have no idea which parts of the signal are normal and which are not.
A simple approach to unsupervised machine learning is to feed those 3,000 frequency bands into an ordinary feed-forward neural network. Remember, DFT (discrete Fourier transform)

returns as many frequency bands as we have samples in the signal, and because we are sampling with 100 Hz for 30 seconds from the physical model this is also the number of frequency bands.
With this approach we have transformed our three-dimensional input data (the three accelerometer axes we are measuring) into a 9,000 dimensional data set (the 3,000 frequency bands per accelerometer axis). This is our new 9,000 dimensional input feature space. We can use the 9,000 dimensional input space to train a feed-forward neural network. Our hidden layer in the feed- forward neural network has only 100 neurons (instead of the 9,000 we have in the input and output layer). This is called a *bottleneck* and turns our neural network into an autoencoder.
We train the neural network by assigning the inputs on the input and output layers. The neural network will learn to reconstruct the input on the output. But the neural network has to learn the reconstruction going through the 100 neuron hidden-layer bottleneck. This way we prevent the neural network from learning about any noise or irrelevant data. We will skip this step here because the performance of such an anomaly detector is usually quite low.

When working with neural networks it is always good to scale data to a range between zero and one. Because we are planning to turn this notebook into a real-time anomaly detector for IoT sensor date we are defining a function instead of transforming the data so that we can make use of the transformer at later stage.

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

We next scale our two sample data arrays.

```
data_healthy_scaled = scaleData(data_healthy)
data_broken_scaled = scaleData(data_broken)
```

7. Improve anomaly detection by adding LSTM layers
We can outperform state-of-the-art time series anomaly detection algorithms and feed-forward neural networks by using long-short term memory (LSTM) networks.
Based on recent research (the 2012 Stanford publication titled *Deep Learning for Time Series Modeling* by Enzo Busseti, Ian Osband, and Scott Wong), we will skip experimenting with deep feed-forward neural networks and directly jump to experimenting with a deep, recurrent neural network because it uses LSTM layers. Using LSTM layers is a way to introduce memory to neural networks that makes them ideal for analyzing time-series and sequence data.
To get started let's reshape our data a bit because LSTMs want their input to contain windows of times.

```
timesteps = 10
dim = 3
samples = 3000
data_healthy_scaled_reshaped = data_healthy_scaled
#reshape to (300,10,3)
data_healthy_scaled_reshaped.shape = (samples/timesteps,timesteps,dim)
```

This way instead of 3000 samples per dimension (per vibration axis) we have 300 batches of length 10. What we want to do is given the last 10 time steps of the signal predict the future 10.

Let's walk through that code a bit. The first thing we need to do in Keras is create a little callback function which informs us about the loss during training. The loss is basically a measure how well the neural network fits to the data. The lower the better (unless we are not overfitting).

```
losses = []

def handleLoss(loss):
        global losses
        losses+=[loss]
        print loss

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

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        handleLoss(logs.get('loss'))
```

It gets interesting because we are now defining the neural network topology.

```
# design network

model = Sequential()
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
model.add(Dense(3))
model.compile(loss='mae', optimizer='adam')

def train(data):
    data.shape = (300, 10, 3)
    model.fit(data, data, epochs=50, batch_size=72, validation_data=(data, data), verbose=0,
 shuffle=False,callbacks=[LossHistory()])
    data.shape = (3000, 3)

def score(data):
    data.shape = (300, 10, 3)
    yhat =  model.predict(data)
    yhat.shape = (3000, 3)
    return yhat
```

Let's walk through this code. First, we create an instance of a Sequential model. This allows us to add layers to the model as we go.

```
model = Sequential()
```

Then, we add an LSTM layer as first layer with 50 internal neurons, input shape of 10 by three, and we want the layer to return a sequence of the predicted future time steps.

```
model.add(LSTM(50,input_shape=(timesteps,dim),return_sequences=True))
```

We do this eleven times, making it an eleven layers deep LSTM neural network. Why did I choose eleven? This is part of the black magic. The more layers you add the more accurate the predictions are (up to a certain point), but the more compute power is necessary. In

most cases, tuning the neural network topology and (hyper)-parameters is considered "black magic" or "trial- and-error."
To get back to normal, we finalize with a normal, fully connected feed-forward layer to bring down the dimensions to three again.

```
model.add(Dense(3))
```

Finally, we compile the model with two parameters:
• loss=mae, which means that the training error during training and validation is measured using the "mean absolute error" measure.
• adam, which is a gradient decent parameter updater. model.compile(loss='mae', optimizer='adam')
For convenience we create two functions for training and scoring. (**Note:** We provide the same data as input and output, because we are creating an auto-encoder.)

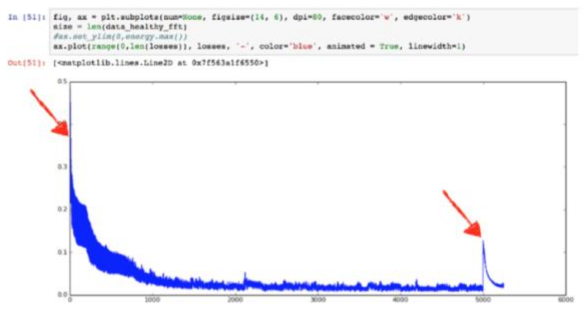
```
model.fit(data, data, epochs=50, batch_size=72,…
```

Let's do the test and train the neural network twenty times with healthy data and one time with broken data. Remember that your callback function handleLoss adds the trajectory of losses (time- series) to an array which we can plot. Therefore, we should see a spike whenever the healthy data pre-trained neural network sees broken data. Not because it knows what healthy or broken is, but because it tells us that it is unlikely that this time-series pattern the neural network currently sees has been seen before.

```
for i in range(20):

    print "----------------"
    train(data_healthy_scaled)
    yhat_healthy = score(data_healthy_scaled)
    yhat_broken = score(data_broken_scaled)
    data_healthy_scaled.shape = (3000, 3)
    data_broken_scaled.shape = (3000, 3)


print "----------------broken"
train(data_broken_scaled)
yhat_healthy = score(data_healthy_scaled)
yhat_broken = score(data_broken_scaled)
data_healthy_scaled.shape = (3000, 3)
data_broken_scaled.shape = (3000, 3)
```

After the training is finished we see this by the absence of the star symbol in the cell caption. It is replaced by a number indicating the execution is finished.

Let's plot the losses to see if we can detect a spike on the abnormal (broken) data.



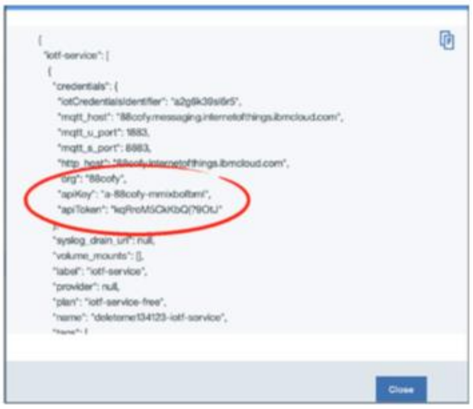
```
In [51]: fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
         size = len(data_healthy_fft)
         #ax.set_ylim(0,energy.max())
         ax.plot(range(0,len(losses)), losses, '-', color='blue', animated = True, linewidth=1)
Out[51]: [<matplotlib.lines.Line2D at 0x7f563a1f6550>]
```

```
fig, ax = plt.subplots(num=None, figsize=(14, 6), dpi=80, facecolor='w', edgecolor='k')
size = len(data_healthy_fft)
#ax.set_ylim(0,energy.max())
ax.plot(range(0,len(losses)), losses, '-', color='blue', animated = True, linewidth=1)
```

And here it is. On time step 5000 we can clearly see that something is going on. Note that the initial loss was higher, but this is because initially the weights of the neural network have been initialized randomly, therefore leading to bad predictions.

8. Analyze the data in real-time with the IBM Watson IoT Platform using MQTT
The last step is to hook this anomaly detector up to the IBM Watson IoT Platform using MQTT to analyze data in real-time. To hook-up our neural network to the platform is straightforward. The values you need are highlighted in Figure 2: org, apiKey, and apiToken. These credentials were generated when you created an IBM Cloud app using the Internet of Things Platform Starter.

Figure 2. IBM Cloud app credentials
Må beskrive hvordan man finner dette.



{
 "iotf-service": [
  {
   "credentials": {
    "iotCredentialsIdentifier": "a2g6k39sl6r5",
    "mqtt_host": "88cofy.messaging.internetofthings.ibmcloud.com",
    "mqtt_u_port": 1883,
    "mqtt_s_port": 8883,
    "http_host": "88cofy.internetofthings.ibmcloud.com",
    "org": "88cofy",
    "apiKey": "a-88cofy-mmixbofbml",
    "apiToken": "kqRroMSCkKbQ(?9OtJ"
   },
   "syslog_drain_url": null,
   "volume_mounts": [],
   "label": "iotf-service",
   "provider": null,
   "plan": "iotf-service-free",
   "name": "deleteme134123-iotf-service",

Let's connect to the MQTT message broker directly from the notebook.

```
options = {"org": "mh741h", "id": "anything", "auth-method": "apikey", "auth-key": "a-mh741h-hwv6qlpmsv",
 "auth-token": "zBcbK&rKTfbW*OfneW"}
client = ibmiotf.application.Client(options)
client.connect()
```

9. Perform streaming analysis by creating a count-based tumbling window
We need to create a window for our data (actually we create a Python queue).

q = Queue(7000)

We are working on count-based tumbling windows of size 3000. We can store more than two such windows in the queue until we prevent the Watson IoT Platform's MQTT message broker to accept new messages. Nevertheless, we need to make sure we are running fast enough to cope with the data arrival rate, otherwise the system will eventually trash. We subscribe to data coming fromthe MQTT message broker and define a call-back function which puts the data into our queue.

```
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")
```

Then we define a function which gets called whenever a count based tumbling window of 3000 samples is full.

```
def doNN(data):
    data_scaled = scaleData(data)
    train(data_scaled)
    yhat = score(data_scaled)
    data_scaled.shape = (3000, 3)
```

Note that we are scaling the data before we call the "train" function which trains our anomaly detector with data (and here we don't know whether it is healthy or broken data). We want to send the current loss during training (remember, we are always training) back to the platform so that we are creating a real-time anomaly score.

```
def handleLoss(loss):
        myData={'loss' : str(loss)}
        client.publishEvent("0.16.2", "lorenz", "status", "json", myData)
```

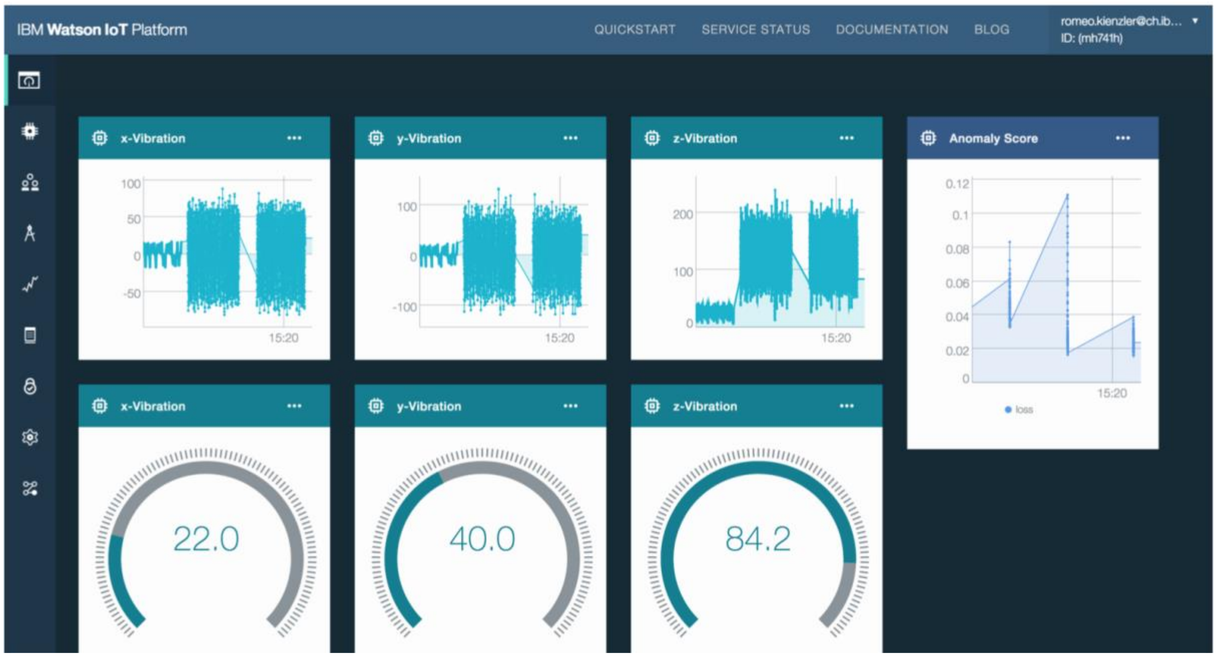
Here it became handy that we can override the handleLoss function to add that functionality.

10. Create a continuous application loop for training
Now we need to start an infinitely running event loop which does the following:

1. Loop infinitely a. Loop over the queue if it is not empty
    - i. If there is data, put it to a temporary array
    - ii. If the count-based sliding window is full then
        1. Send the window to the neural network
        2. Empty the array

```
import numpy as np
while True:
    while not q.empty():
        point = q.get()
        try:
            data
        except NameError:
            data = np.array(point)
        else:
            data = np.append(data,point)
        if data.size>=9000:
            data = np.reshape(data,(3000,3))
            print data
            doNN(data)
            del data
```

Because the actual training loss is sent back to the IBM Watson IoT Platform in real-time we can create a little dashboard to visualize the vibration sensor data together with the anomaly score in real-time – we will create this next….



Conclusion

This completes our third deep-learning tutorial for IoT time-series data and concludes the series. We've learned how TensorFlow accelerates linear algebra operations by optimizing executions and how Keras provides an accessible framework on top of TensorFlow. Finally, we've shown that even an LSTM network can outperform state-of-the-art anomaly detection algorithms on time-series sensor data – or any type of sequence data in general.