Developing cognitive IoT solutions for anomaly detection by using deep learning, Part 1: Introducing deep learning and long-short term memory networks

Detecting anomalies in IoT time-series data by using deep learning

Romeo Kienzler

This article is the first in a five-part series, "Developing cognitive IoT solutions for anomaly detection by using deep learning." This article explains what deep learning is, what neural networks are, and how they can be used to analyze the large amount of data that IoT sensors gather.

Although predictions are always controversial, Gartner says that there are 8.4 billion connected IoT devices in 2017 (not counting smartphones) and some analysts say that by 2020 there will be 50 billion. Even if over-estimated, soon the number of IoT devices will exceed the number of humans on the planet. And guess what, all these devices are continuously generating data; data that is useless unless you can analyze it.

A cognitive system provides a set of technological capabilities such as artificial intelligence (AI), natural language processing, machine learning, and advanced machine learning to help with analyzing all that data. Cognitive systems can learn and interact naturally with humans to gather insights from data and help you to make better decisions. In my last article I stated cognitive computing is not just human-computer interaction (HCI), it is advanced machine learning driven by powerful algorithms (models) and nearly unlimited data processing capabilities.

To understand a cognitive system that uses IoT sensors and deep learning analysis, you first need to understand the leap from advanced machine learning to neural networks. In this article, I try to help you make that leap. In the coming weeks, I will present three different tutorials about anomaly detection on time-series data on Apache Spark using the Deeplearning4j, ApacheSystemML, and TensorFlow (TensorSpark) deep learning frameworks to help you fully understand how to develop cognitive IoT solutions for anomaly detection by using deep learning.
From machine learning to neural networks and deep learning

If a cognitive system is based on models, you need to look first at what a machine-learning model is. It is a statistical model (black box) that in contrast to a physical model (white box) has been trained with data to learn a hidden pattern.

Look at Table 1. It contains historic data on different parameters, measured observations on a manufacturing pipeline, and a binary outcome.

Table 1. Machine-learning model

<table>
<thead>
<tr>
<th>Part No.</th>
<th>Max Temp. 1</th>
<th>Min Temp. 1</th>
<th>Max Vibration 1</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>35</td>
<td>35</td>
<td>12</td>
<td>Healthy</td>
</tr>
<tr>
<td>101</td>
<td>36</td>
<td>35</td>
<td>21</td>
<td>Healthy</td>
</tr>
<tr>
<td>130</td>
<td>56</td>
<td>46</td>
<td>3412</td>
<td>Faulty</td>
</tr>
</tbody>
</table>

In this highly artificial example, the numbers speak for themselves. As you might guess, a high temperature and high vibration led to a faulty part. The root cause for this situation might be the result of a broken bearing in a machine.

In this example, a (supervised) machine-learning algorithm is capable of considering all this data (and much more) to learn and predict faults from pure data. The component that such an algorithm produces is called a machine-learning model.

A special type of machine-learning algorithm is a neural network. It is highly adaptable to data, and it is able to learn any hidden mathematical function between the data and the outcome. The only catch with neural networks is the tremendous amount of computational resources and data that they need to perform well. So why am I talking about neural networks at all? We live in an IoT world with tremendous amounts of data available and also (nearly) unlimited computational power available by using the cloud. This situation makes neural networks especially interesting for IoT data processing.

Neural networks are inspired by the human brain, and so are deep learning networks. The main difference between a neural network and a deep learning one is the addition of multiple neural layers. The most obvious example of how deep learning is outperforming traditional machine learning is with image recognition. Every state-of-the-art system uses a special type of deep learning neural network (called a convolution neural network) to perform their tasks.

For example, deep-learning-based image recognition algorithms are capable of distinguishing healthy parts from faulty parts in a manufacturing pipeline. I'd call this machine intelligence and it is available as in the IBM Watson Visual Recognition Service. For this particular example, the machine might accomplish the same task with the same accuracy as a human. The only machine advantage is that it never sleeps, never calls in sick, and never gets hurt. And, if you need to double the throughput, just double the amount of hardware or cloud resources. But applying a root-cause analysis on why parts are sometimes faulty is still the domain of human experts. However, this scenario is where cognitive solutions applying deep learning.
In fact, a visual recognition service returns much more information than just a binary outcome of "healthy" or "faulty." Just like a human, the service detects structures and regions in the images that deviate from the norm. Now, if you were to correlate all sound and vibration sensor data with all visual recognition data, I'm sure such a system could detect the root causes of faults as well as, or even better than, humans.

### How artificial neural networks work

If the IoT sensors that connected to a message broker (like the MQTT-based IBM Watson IoT Platform) are the central nervous system of cognitive solutions, then deep learning is the brain. And, to understand deep learning, you need some basic understanding of regression, perceptrons, biological and artificial neural networks, and hidden layers.

#### Start with linear and logistic regression

A ton of scientific literature exists on regression, so I'll try to give you a short-path explanation that is tailored for developers. Consider Table 2. It is the same as Table 1 except here I've turned the outcome into a binary representation.

<table>
<thead>
<tr>
<th>Part No.</th>
<th>Max Temp. 1</th>
<th>Min Temp. 1</th>
<th>Max Vibration 1</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>35</td>
<td>35</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>46</td>
<td>35</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>130</td>
<td>56</td>
<td>46</td>
<td>3412</td>
<td>0</td>
</tr>
</tbody>
</table>

It's pretty easy to write a piece of software to make the classification.

```python
def predict (datapoint):
    if datapoint.MaxVibration1 > 100 :
        return 0
    else :
        return 1
```

This example of a white box model is where your knowledge of the underlying physical system (or a domain expert looking at the data) helped to create some rules. The idea of machine learning is to have the machines learn those rules from data. And the way this learning is accomplished is by using a skeleton and an optimization algorithm to fill in the missing parts of the skeleton. I'll use such a skeleton (linear regression):

```python
def predict (dp) :
    return a + b * dp.MaxTemp1 + c * dp.MinTemp1 + d * dp.MaxVibration1
```

As you can see, if I choose parameters b and c to be 0, a to be 1, and d to be -0.0002930832 (which is -1/3412), I end up pretty close to my required result. And the good news is that parameters a, b, c and d can be learned from an algorithm. The result: machine learning!
You might notice that the results are not exactly 1 or 0 because linear regression is a continuous model (trying to predict values of type float). But what I want is a binary classifier. Let me turn this algorithm into a logistic regression model by just adding a helper function called `sigmoid`.

```python
import math
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
def predict(dp):
    return sigmoid(a + b * dp.MaxTemp1 + c * dp.MinTemp1 + d * dp.MaxVibration1)
```

The `sigmoid` function does nothing else other than squashing a range from minus infinity to plus infinity to a range 0 - 1. The only thing that I need to do is define a threshold, for example turning values < 0.5 to 0 and > 0.5 to 1.

**Figure 1. The sigmoid function map range**

![Sigmoid Function Graph](image)

**The most simple neural network: perceptron**

You’ve now mastered the simplest neural network: a *perceptron*. It is similar to a logistic regression model – with some negligible differences. But the term "neural networks" sounds far cooler, doesn’t it? Look at the perceptron in Figure 2.

**Figure 2. A perceptron**

![Perceptron Diagram](image)

I have our input to the model $x_1$ to $x_n$, and the model parameters $w_1$ to $w_n$. And then I sum things up and squash it with an activation function (for example, `sigmoid`). If I write this model a bit differently, it looks like this:

$$\text{out}(x_1, x_2, \ldots, x_n) = \text{activation}_\text{function}(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

And if I replace `activation_function` with `sigmoid` I am back to logistic regression. The only thing is... it now looks a bit more like the structure of a brain. But what does the human brain look like?
From biological neural networks to artificial neural networks

Look at the biological neuron in Figure 3. In mathematical terms, the dendrites are the "inputs" where upstream data gets fed in, the cell body (and the synapses) compute the sum and the "activation function," and the "result" is sent downstream over the axon to subsequent downstream neurons.

Figure 3. Biological neuron

And because it is fun (at least it is to me), look at Figure 4 about a biological synapse connecting neurons (biochemically) together. This figure represents the set of $w_{in}$ values in the perceptron.

Figure 4. Biological synapse

If you hook multiple neurons together, you'll get a brain or an artificial neural network as shown in Figure 5.

Figure 5. Artificial neural network

The difference is in the hidden layer

The biggest difference between this artificial network and the perceptron is the hidden layer. What's this all about? Is it useful? Maybe. At least in the human brain, hidden layers seem to be useful because we have hundreds of stacked hidden layers in our brain. In this feed-forward neural network example in Figure 6, you can see that the models are stacked.
Can't see it? Look how this model would look in a programming language. But before we do this, let me tell you a secret. A secret that mathematicians use to scare away everyone. But it is totally simple. I'm talking about vector multiplication. So what is a vector? Just a list of numbers, nothing else. Programmers might call it an array. And multiplication on vectors is defined slightly differently. It says if you have a vector \( \mathbf{x}=(x_1, x_2, \ldots, x_n) \) and another vector \( \mathbf{w}=(w_1, w_2, \ldots, w_n) \) and if you multiply them \( \mathbf{x} \cdot \mathbf{w} \) the result is a single value:

\[
\sum_{i=1}^{n} x_i w_i
\]

Does this model look familiar to you? If not, look at the linear regression again. I can compute a single neuron by calculating \( \text{sigmoid}(\mathbf{x} \cdot \mathbf{w}) \). And because Python doesn't support vector multiplication out of the box, I can use a library called \texttt{numpy} for doing so. I'll start to compute a single neuron:

```python
import numpy as np
datapoint = {"MaxTemp1": 35, "MinTemp1": 35, "MaxVibration1": 12}

# randomly initialize weights
w_layer1 = np.random.rand(4)

def neuron1(dp):
x = np.array([1, dp["MaxTemp1"], dp["MinTemp1"], dp["MaxVibration1"]])
return sigmoid(x.dot(w_layer1))

print(neuron1(datapoint))
```

If I want to compute multiple neurons at the same time (for example, all neurons of a layer), then I can use this function multiple times. But mathematicians invented one more thing to confuse you (in case you've survived vector multiplication): matrix multiplication. In neural networks, you can apply computations like in linear regression various times in parallel, and you can write this up as a matrix multiplication in this form.

Again, I assume that our input is in vector \( \mathbf{x}=(x_1, x_2, \ldots, x_n) \). But now I want to compute all neurons in the hidden layer at the same time. Instead of multiplying \( \mathbf{x} \) with a weight vector \( \mathbf{w} \) to compute a single neuron I multiply \( \mathbf{x} \) with a weight matrix \( \mathbf{w} = \)

\[
\begin{bmatrix}
w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\
w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\
\vdots & \vdots & \ddots & \vdots \\
w_{n,1} & w_{n,2} & \cdots & w_{n,m}
\end{bmatrix}
\]
Matrix multiplication defines $w^*x =$

$$
(w_{1_1}x_1, \ w_{1_2}x_2, \ldots, \ w_{1_m}x_n, \\
w_{2_1}x_1, \ w_{2_2}x_2, \ldots, \ w_{2_m}x_n, \\
\ldots \\
w_{n_1}x_1, \ w_{n_2}x_2, \ldots, \ w_{n_m}x_n)
$$

My result is a vector of float values that I then can feed into the next layer. Here the same code in Python:

```python
import numpy as np

# make sigmoid function applicable to vectors instead of scalars only
def sigmoid(z):
    s = 1.0 / (1.0 + np.exp(-1.0 * z))
    return s

datapoint = {"MaxTemp1": 35, "MinTemp1": 35, "MaxVibration1": 12}

# randomly initialize weights, now as a matrix for the four hidden layer neurons
w_layer1 = np.random.rand(4,4)

def layer1(dp):
    x = np.array([1, dp["MaxTemp1"], dp["MinTemp1"], dp["MaxVibration1"]])
    return sigmoid(x.dot(w_layer1))

print(layer1(datapoint))
```

As you can see, the code fragment `x.dot(w_layer1)` doesn't change at all. But internally a matrix multiplication is applied instead of a vector multiplication because `w_layer1` is now a matrix instead of a vector. Finally, I need to add the output layer and I am done:

```python
w_layer2 = np.random.rand(5,2)

def layer2(x):
    x = np.concatenate(([1], x))
    return sigmoid(x.dot(w_layer2))

print(layer2(layer1(datapoint)))
```

Of course, the weight matrix $w$ isn't trained, but only randomly initialized. The training of a neural network is beyond the scope of this article.

**Note:** This random neural network outputs two float numbers. These numbers are the probabilities of an item belonging to one class or the other. Again, because the neural network is untrained these numbers don't make sense for now.

$$\begin{bmatrix} 0.97435552 & 0.89144977 \end{bmatrix}$$

**Going deep into the hidden layer**

What is the point of the hidden layer? Each layer can be seen as an individual machine learning algorithm on its own. The output of an upstream layer is used as input, and its output is passed to the next layer, which then takes it as input to the next layer, and so forth. This behavior means that when stacking layers and creating a deep neural network the system is learning intermediate
representations of data to help a downstream layer to perform better. The most inspiring example is the vision system of the human brain. The first layers (the ones right after your photoreceptors) don’t recognize faces, cars, and cats, but do recognize shapes such as lines, circles, rectangles, and points. The deeper you go, the more things a layer can recognize...until you eventually recognize the whole image.

This brings me to the conclusion of this section: A deep feed-forward neural network can learn any mathematical function (given enough data and compute power to train it).

**Time-series data needs long-short term memory networks**

Hopefully you are convinced that neural networks are quite powerful. But unfortunately when it comes to times-series data (and IoT data is mostly time-series data), feed-forward networks have a catch. These networks are bad in recognizing sequences because they don’t hold memory. One solution to this problem is adding feedback loops.

**Figure 7. Feedback loops**

These feedback loops somewhat fix the problem because now the network can remember. But it works only on temporally close events. If the network is to remember more distant temporal events, memory cells must be added to the neurons to get a long-short term memory (LSTM) network as shown in Figure 8.
Training needs computing power

What is the downside to an LSTM network? It needs a lot of data and a lot of computing power to achieve good performance when using neural networks. The good news is current graphics processing unit (GPU) cards have the same computing power as a 16,000 core cluster had in 2009. And, IBM is in the process of releasing a neuromorphic chip as an offering from the IBM Watson Group with the same compute power, but consuming only 3 watts. So there is hope for future performance improvements and energy savings.

LSTM networks are outperforming the state-of-the-art algorithms in time-series forecasting and anomaly detection, as discovered in the following publications:

- Long Short Term Memory Networks for Anomaly Detection in Time Series
- Deep Learning for Time Series Modeling CS 229 Final Project Report
- Variational Autoencoder based Anomaly Detection using Reconstruction Probability

LSTM networks can learn any algorithm

Why am I telling you all this for a cognitive IoT scenario? Because an LSTM network is Turing complete. This means that an LSTM network can learn any algorithm that a computer can perform. Period.

And that contrasts with many of the state-of-the-art machine learning algorithms that are used for time-series forecasting and anomaly detections, which are the disciplines that cognitive IoT is all about. In cognitive IoT applications, you either need to predict the future state of the system to take regulative actions in advance or you want to be notified if something is out of order.

So what is the problem with state-of-the-art algorithms? Nothing. I’m saying LSTMs usually perform better than state-of-the-art; sometimes significantly, sometimes a little, and rarely worse (especially if you have enough data to train the system). It depends on your use case if the gain in accuracy justifies the additional computational costs.
Demo use case: Anomaly detection for IoT time-series data

I want you to start thinking about this demo use case. This use case will be the basis for the follow-up tutorials. Consider the task of detecting anomalies in vibration (accelerometer) sensor data that is measuring a bearing. You would attach a sensor to the bearing as shown in Figure 9.

Figure 9. Accelerometer sensor on a bearing

An accelerometer sensor records vibrations on each of the three geometrical axes x, y, and z. More sophisticated sensors exist that also take rotational movement patterns into account, but we are fine with this simple one as it can be found in every smartphone. (If you want to learn how to access and analyze your accelerometer in your smartphone you can follow this [tutorial](https://www.example.com).) As it is hard to take such a system with you, I will need to implement a simulator in Node-RED to generate data by using a physical Lorenz attractor model. I'll use this generated data in the tutorials to detect anomalies, basically predicting when a bearing is about to break. You can read the next tutorial in this series, [Generating data for anomaly detection](#), to see how I created this test data simulator.

I can switch the test data generator between two states: healthy and faulty. Figure 10 is a phase plot showing the three vibration dimensions on a time series in a healthy state.
Figure 10. Phase plot of a healthy state

![Phase plot of a healthy state](image)

With the same phase plot in Figure 10, you can observe when changing the parameters of the physical model slightly we get a faulty state.

Figure 11. Phase plot in a faulty state

![Phase plot in a faulty state](image)

If you’re not familiar with phase plots, here is a run chart in Figure 12 and Figure 13 for the three axis values (again in healthy and faulty state).
One common technique is to transform this data from the time to the frequency domain by using DFT (discrete Fourier transform) or wavelets. Again, I'm showing in Figure 14 and Figure 15 the DFT for healthy and faulty states.
You can clearly see that the faulty state has more energy and additional frequencies present. This data would be sufficient to train a classifier as you've learned before. But we can do better. We will construct a system capable of learning normal behavior from data, and after it sees new (unseen) data (or sequential patterns) it will raise an alert.

Such a system is a LSTM-based auto-encoder as shown in Figure 16.
This system compresses the vast amount of data through a neural bottleneck to try to reconstruct the same data it has seen, but, of course, by using a bottleneck it loses vast amounts of irrelevant data. Such a neural network learns how a system normally behaves, and as soon as new patterns are seen it has a hard time reconstructing the data, and it raises an alert.

**Conclusion**

At this point, you should have sufficient understanding of neural networks and what applying deep learning concepts to your data can do for your IoT data in your cognitive system.

In the next set of tutorials, I'll focus on using different technologies on Apache Spark to implement such a deep learning system and further cement your understanding of the impact that such technology can have in your systems. I'll talk about open standard and open source solutions running in the IBM Cloud such as Deeplearning4j, Apache SystemML, and TensorFlow (TensorSpark).
Related topics

- A beginner's guide to artificial intelligence, machine learning, and cognitive computing
- 5 things you need to know when creating a cognitive app
- Recurrent neural networks deep dive
- Read more on IoT use cases with Deep Learning beyond structured data

© Copyright IBM Corporation 2017
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)