Expertise
and insight
for the future

Miika Toikkanen

A Cloud-Based Analysis Tool for Vibration Monitoring with Neural Networks

Metropolia University of Applied Sciences

Bachelor of Engineering

Electrical and Automation Engineering

Bachelor's Thesis

11 November 2019

Author Title Number of Pages Date	Miika Toikkanen A Cloud-Based Analysis Tool for Vibration Monitoring with Neural Networks 34 pages + 5 appendices 11 November 2019
Degree	Bachelor of Engineering
Degree Programme	Electrical and Automation Engineering
Professional Major	Electronics
Instructors	Heikki Valmu, Senior Lecturer Kalle Ahola, Technology Manager

The aim of this thesis project was to investigate the usefulness of machine learning and the cloud platform provided by Microsoft Azure Machine Learning service in the projects of the Testing and Quality Assurance team of Protacon Technologies Ltd. In addition to this, it was meant to serve as instructional material and as a pilot for upcoming projects utilizing similar concepts.

The project was carried out by developing methods for expanding the functionality of an existing vibration measurement system with a cloud-based analysis tool employing neural networks. First, the effects of degradation in the vibration of rotating bearings was studied. Then, a recurrent neural network capable of detecting the degradation was developed with Keras on Python. Finally, a method for deploying neural networks as web-services was investigated.

As the outcome of this project, a functional tool for analysing vibration signals was implemented as a web-service hosted in the Azure cloud. From the data set which the analysis tool was developed and tested on, bearing degradation was detected and quantified millions of rotations prior to failure.

The result of this work shows that the techniques utilized are viable for the intended purpose and provides a baseline for developing a more sophisticated and universal analysis tool in further works.

Keywords	Azure, Neural Networks, Vibration Analysis, Keras

Tekijä Otsikko Sivumäärä Aika	Miika Toikkanen Pilvipohjainen analyysityökalu värähtelyn tarkkailuun neuroverkkoja hyödyntäen 34 sivua + 5 liitettä 11 Marraskuu 2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Sähkö- ja Automaatiotekniikka
Ammatillinen pääaine	Elektroniikka
Ohjaajat	Heikki Valmu, Yliopettaja Kalle Ahola, Technology Manager

Insinöörityö-projektin tavoitteena oli tutkia koneoppimisen ja Azure Machine Learning service-pilvipalveluiden hyödyllisyyttä Protaconin Testing and Quality Assurance-tiimin projekteissa. Tämän lisäksi sen tarkoituksena oli tuottaa materiaalia, joka voisi toimia ohjeistuksena ja lähtökohtana tuleviin projekteihin.

Projekti toteutettiin kehittämällä metodeja jo olemassa olevan tärinämittaus-järjestelmän toiminnallisuuden laajentamiseksi pilvessä toteutetulla neuroverkkoihin perustuvalla analyysityökalulla. Aluksi tutkittiin laakerien toimintakunnon heikkenemisen merkkejä värinäsignaaleista. Sitten kehitettiin Keras-kirjastoa ja Python ohjelmointikieleltä käyttäen neuroverkko, joka kykeni havaitsemaan toimintakunnon muutokset mittadatasta. Lopuksi tutkittiin metodeja Keras-kirjastolla kehitettyjen neuroverkkojen käyttöönottamiseksi web-palveluina.

Projektin tuloksena syntyi toimiva web-työkalu, joka kykenee analysoimaan värinädataa ja mittaamaan laakerien kunnon heikkenemistä. Testaamiseen käytetystä datasta ongelmat voitiin havaita ja niiden vakavuus mitata laakereilla ollessa vielä miljoonia pyörähdyksiä jäljellä ennen vahingoittumista.

Työn lopputulos osoittaa että konseptit, joita testattiin ovat sopivia tähän käyttötarkoitukseen ja työ toimii hyvänä lähtökohtana kehittyneemmän ja yleispätevämmän analyysityökalun kehittämiseen.

Avainsanat	Azure, Neuroverkot, Värähtelyanalyysi, Keras

Contents

List of Abbreviations

AI	Artificial intelligence. A field of science concerned with intelligent machines.
ANN	Artificial neural network. A collection of artificial neurons.
API	Application programming interface. A set of functions.
BPTT	Backpropagation through time. The backpropagation algorithm applied over several timesteps.
CSV	Comma-separated values. A file format represented as text.
HTTP	Hypertext transfer protocol. A protocol that defines how data is formatted on the internet.
JSON	JavaScript object notation. An open standard file format.
LSTM	Long-short term memory. A recurrent neural network architecture capable of learning long-term dependencies.
ONNX	Open neural network exchange. An open format for neural networks.
REST	Representational state transfer. A popular web programming architecture.
RMS	Root mean square. A quantity measuring the average magnitude of an alternating signal.
RNN	Recurrent neural network. A neural network with feedback elements.

1 Introduction

Recently artificial Intelligence has become an incredibly universal discipline, as it is linked to most aspects of modern life in one way or another. The array of fascinating applications in the field outperform humans in increasingly complicated tasks. Harnessing this power appropriately releases limited time resources from the tedious and repetitive tasks prone to human errors all too often reserving the valuable time of professionals. In this project, yet another attempt to employ the machine learning techniques for automating laborious tasks is made by investigating the potential of cloud-based artificial neural networks for the testing and quality assurance projects of Protacon Technologies Ltd.

Artificial intelligence is already over half a century old field and especially artificial neural networks, the specific area of interest for this work, has already been well researched with the material for getting started in just about anything related to it readily available online. The background knowledge provided in this paper is useful as an introduction or reference to anyone interested in the specific application of vibration analysis and monitoring with neural networks in the cloud.

The choice of topic was motivated by a few factors, the first of which are the needs of this thesis work commissioner. Cloud-based solutions for computing and data storage are becoming more and more important and the quantity of data collected from each system is substantial. Performing analysis on this massive amount of data using traditional tools such as Excel is arduous and limited in effectiveness. For these reasons it is necessary to consider automated systems with analytical capabilities extended beyond those of humans and the access to data right where it is stored in the cloud. Furthermore, a thorough research into neural networks is of interest to the author of this paper due to the aspiration for strong expertise in the field of artificial intelligence for further study.

The project is carried out by developing the methods for extending the functionality of an existing vibration measurement system VibLog. At its present state it gathers data, calculates key performance indicators and stores them locally for later analysis. While this is enough in many cases, it leads to a situation, where the data is separated into multiple destinations for storage and analysis in memory sticks, databases and computer drives. Moreover, the interval, in which the analysis is performed, depends on human judgement and is not necessarily regular enough for sufficient reliability in some applications. In addition to this, some slowly rotating machinery is problematic to analyse, as the conditions which would indicate faults do not occur often enough to be noticeable with some tools. As of now, there are no labelled data sets measured by the VibLog system available for this project, therefore a publicly available data set with similar type of data is used to develop the needed functionality.

The problems discussed above will be addressed by many changes to the measurement system. This work focuses on implementing one of them by finding the techniques for developing an analysis tool hosted in Azure cloud, capable of detecting deterioration of bearings in rotating machinery and predicting a possible failure prior to its occurrence based on vibration data. From this description, the following questions are formulated:

- Can a fault likely to occur be reliably determined from the data using artificial neural networks?
- Can the deterioration of performance leading up to failure be quantified numerically and used to predict the remaining lifetime using artificial neural networks?
- Can the artificial neural networks discover fault conditions from slowly rotating machinery, which might be otherwise problematic to analyze?

The objective of this project is to develop a neural network capable of predicting failures in rotating machinery based on their vibration, then implementing it as an analysis tool into the cloud and in the process of doing so, answering the questions generated above.

2 Theoretical Background

This section introduces the theoretical concepts and gives a brief explanation of the background knowledge required in the project. Although no derivation of equations is performed and theory is explained on a general level, a basic grasp of linear algebra and calculus is assumed from the reader to understand the notation.

2.1 Vibration Monitoring

Abrupt failures of critical machinery in manufacturing plants or other facilities can lead to enormous expenditures in repair fees and losses from the disturbance caused to other processes which may depend on the damaged equipment. This can be especially disastrous, if it is to happen during the night or a weekend when maintenance teams are less capable of dealing with such issues, while overtime fees further add to the financial damages sustained.

Condition monitoring is the procedure for examining the state or health of machinery in order to predict potential problems and prevent costly breakages before they are likely to occur. The indicators of the impending failures can be observed by measuring various quantities such as flow, temperature or pressure. Vibration, the mechanical oscillation observed in rotating machinery, is the most common evidence relied on for predicting incoming problems. [1.]

Often the key component for maintenance is the rolling bearing. Imbalances, misalignments, wear and breakages in the bearing and other components of the system affect the vibration observed and can reveal much about the system's health. In the case of bearings, physical dimensions, rotational velocity and characteristics such as the number of rolling elements contained within affect the frequencies, at which the resonation is detected. Consequently, the patterns are unique to the situation and the system. Imperfections and defects from manufacturing add to the difficulty of analyzing the oscillations, and even identical machines cannot be guaranteed to vibrate in the same frequencies. [1; 2.]

Vibration is typically measured with piezoelectric sensors or accelerometers. Every measurement system comes with their own constraints posed by the sensor characteristics, transformations performed on the signal and the rate, at which the samples can be recorded [3]. If the vibration is measured with an insufficiently capable system, the signs of degradation may not be present in the signal at all, or they may have been unintentionally filtered out by the time the signal has reached the digital format intended for analysis by computers.

2.2 Neural Networks

The fundamental concepts utilized in artificial neural networks (ANN) are derived from nature and to fully comprehend them, it is important to first have a basic understanding of some elementary neuroscience. This section provides an overview of the principles that go with the biological paradigm. It should be noted however, that ANNs are not able to perfectly model their natural counterparts due to the enormous complexity, nor are they meant to. Rather, they abstract the useful ideas for practical computation and are merely an attempt at exploiting the information processing capabilities encountered in natural nervous systems. [4,1.] As complex as the nervous systems are, they comprise of relatively simple building blocks, neurons, as depicted by figure 1.

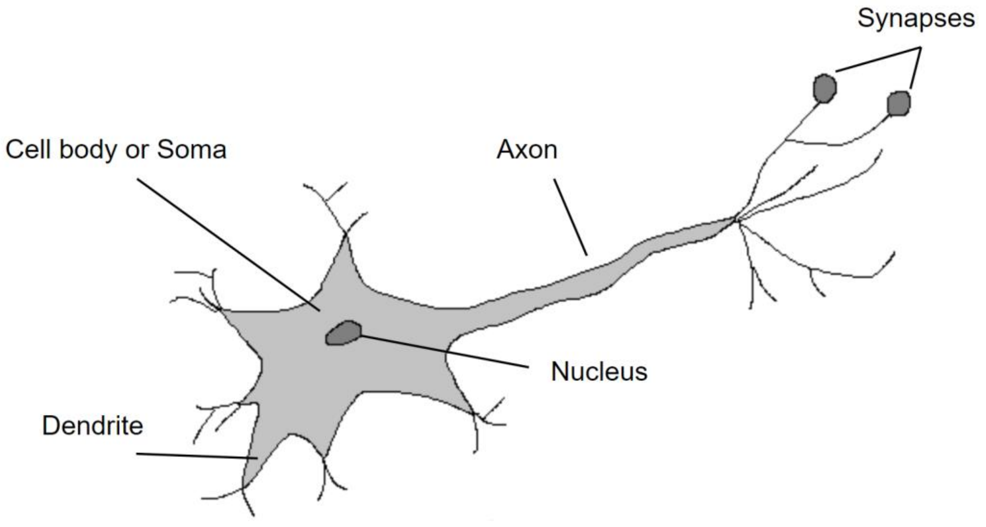

Figure 1. A simple representation of a biological neuron. Redrawn from Artificial Intelligence, A Modern Approach [11,5].

The soma, also known as the cell body, hosts various organelles, which produce chemicals and energy needed for continuous operation of the neuron. Attached to the cell body are several dendrites, acting as transmission channels for incoming signals, while the axon serves as an output channel through which electrical signals propagate towards other neurons. The communication media are connected via the interfaces between neurons, synapses, which facilitate the chemical transmission between cells and determine the directionality of signals. [4,18-19;5,11.]

The cells in biological nervous systems process information by means of chemical and electrical processes and are as a result of millions of years of evolution, “sophisticated self-organizing systems”. In fact, the complexity exhibited by the internal mechanical processes have resulted in research towards a control system of the neuron. The neurons are as complicated as personal computers, and consequently, they are often more appropriately referred to as computing units. [4,22.] While several orders of magnitude slower than logic gates embedded onto silicon wafers, when arranged as complex formations in massive numbers, they are capable of calculations which the traditional computers cannot efficiently perform. Each of the neurons may be connected from just tens of other neurons up to hundreds of thousands [4,4;5,11-12]. The massive neural network structures in living creatures' brains formed by primitive computation units facilitate the intricate responses to their perceptions of the environment [4,4].

2.3 Artificial Neural Networks

With the groundwork laid in the previous section, artificial neural networks, from now on referred to as neural networks are much more easily understood. First, the reader is familiarized with the basic computational unit, artificial neuron shown in figure 2. That knowledge is then applied with network architectures comprising of those basic nodes. Finally, the learning algorithms and some concepts utilized in the implementation of the analysis tool are explained on a general level.

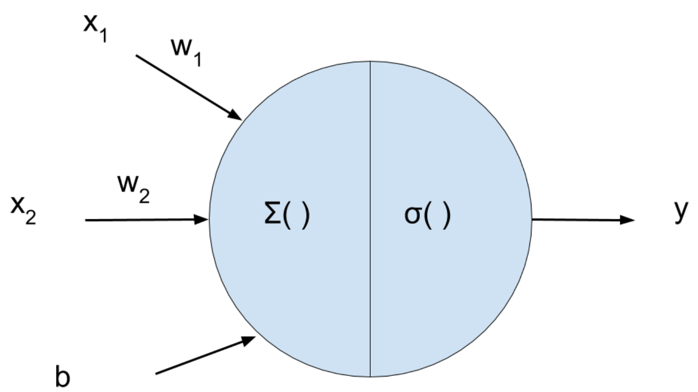

Figure 2. An artificial neuron.

Comparing the artificial neuron in figure 2 with the biological neuron, the following parallels can be drawn. The inputs denoted with x_i are analogous to axons of other neurons, the weights w_i regulate the degree of excitation similarly to the synapses and the bias b serves as an activation threshold. In the artificial version of the cell body, the excitations observed at the inputs are summed together and passed through the activation function, which determines if the neuron has "fired" or not [5,728]. This can be mathematically expressed as equation (1), where n is the number of nodes and σ is the sigmoid function.

$$y = \sigma(b + \textstyle\sum_{i=0}^{n} w_i x_i) \tag{1}$$

There exists a multitude of activation functions, each useful for specific applications. The sigmoid activation function σ in equation (2) is used here as an example because it is a simple general-purpose activation function, which maps the real number line into the range 0-1 in a continuous and differentiable manner [6,14].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Neural networks can be depicted as graphs regarding the neurons as nodes and edges as weights. Generally, the neural network topologies can be divided into two categories, feedforward networks and recurrent networks. First of the two, the feedforward network is shown in figure 3. It is formed by an input layer x, output layer y and zero or more consecutive hidden layers h in a strictly one-directional arrangement. [5,729.]

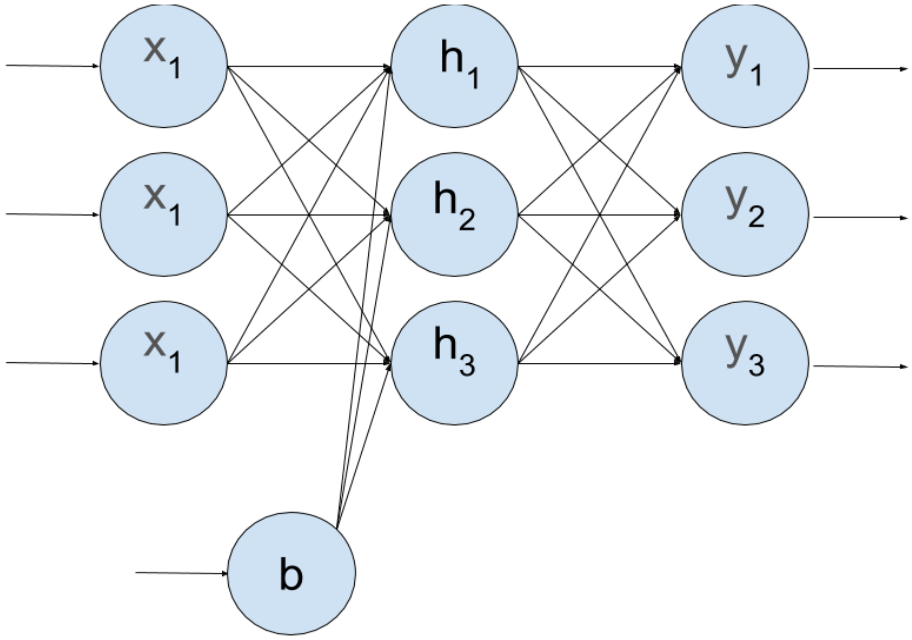

Figure 3. A simple feedforward neural network

Neural networks are seldom limited to one layer or one neuron. Therefore, it is convenient to represent equation (1) as the vector equivalent equation (3), where W is a matrix of weights, y is the vector of outputs, x is the vector of inputs, b is the vector of biases. The activation function σ acts on every element of the resulting vector, but in figure 2, the vectors y and b only have a single element.

$$y = \sigma(Wx + b) \quad (3)$$

The network in figure 3 can be mathematically expressed by (4) and (5), where the vectors x, h and y represent the input, hidden and output layers respectively, and the matrices W_h and W_y contain weights for the connections between those layers.

$$h = \sigma(W_h x + b) \tag{4}$$

$$y = \sigma(W_y h + b) \tag{5}$$

Much larger neural networks can be represented by replicating the equation (4) for each consecutive hidden layer. The calculation of these values for each layer is referred to as feed-forward or the forward pass in context of learning algorithms.

2.3.1 Learning Algorithms

A learning algorithm, sometimes referred to as the optimization algorithm automatically finds parameters, or the weights of the neural network which lead to the desired behavior by iteratively comparing the networks predictions to the actual expected values and taking corrective steps based on the error observed [4,77]. In contrast to the parameters that can be learned, there are various hyperparameters that cannot be learned and must be decided on prior to the learning process. Number of layers, nodes or the learning algorithm itself can be considered as hyperparameters.

Inference made by the neural network is never entirely accurate. The loss function is used to quantify the difference by comparing the predictions made with the desired outcome, known as the label [7,9-10]. Many different loss functions for various purposes are available. A commonly used example would be the mean squared error (6), where n is the number of outputs, y_i is the prediction and $\hat{y}_i$ is the label.

$$L = \sum_{i=0}^{n} (\hat{y}_i - y_i)^2 \tag{6}$$

Often the learning in neural networks is implemented as the backpropagation algorithm, which uses a method, such as the gradient descent for finding the minimum error and calculates the new values for each parameter in every layer [5,733]. The error is propagated from the output layer to the hidden layers by applying chains of derivatives to compute the contribution of each parameter to the total error at the output layer [6,40-41;7,52].

The process of training begins with the initialization of the network's weights to random values. The error gradient is then calculated, and corrective steps taken until convergence to a minimum. Each weight w_i is updated using the equation (7) where L is the loss function and r is the learning rate, a parameter controlling the step size. [5,719.]

$$w_i = w_i - r\frac{\partial L}{\partial w_i} \tag{7}$$

This algorithm works well for any neural network, however modern tools implement some more advanced and efficient variants of the gradient descent, such as the stochastic gradient descent (SGD), RMSProp and Adam, which utilize the concept of momentum to get past the local minima into the global minimum while converging faster than the basic gradient descent. [6,30;7,63.]

2.3.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are a class of neural networks suitable for processing sequences of values. They can map sequences to vectors, vectors to sequences or sequences to sequences. Figure 4 shows two representations of an RNN which maps a sequence x to a sequence o and produces the loss L, given the label y. On the left, all the time steps are represented at once with a time delay denoted as a black square, while on the right, the same RNN is unfolded across time.

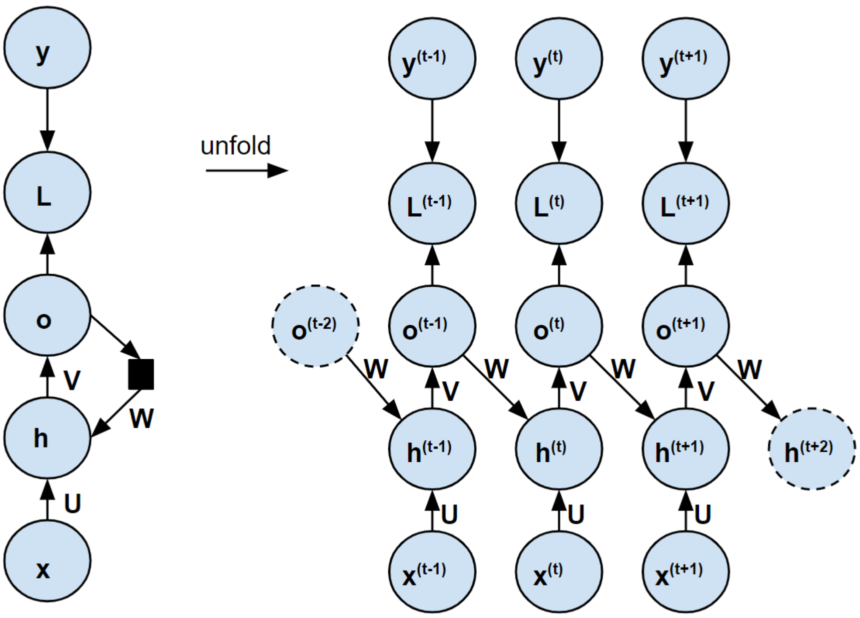

Figure 4. Two representations of a recurrent neural network in a computational graph. Redrawn from Deep Learning [8,378].

As figure 4 shows, the parameters of the network, U, V and W are shared through time and the feedback provides information on past states. As the RNN can internally loop through the information in previous states, the time sequence can be processed step by step. Using a regular feedforward network would require transforming each time instant into a single fixed size input, with individual parameters for each. It would be very constrained in terms of the sequence length and its ability to detect patterns would depend on their location in the time sequence [7,196].

The feedforward for the RNN is calculated starting from an initial state $h^{(0)}$ by using the equations (8) - (11) for each timestep from t = 1 to the final timestep t = τ. The input of the hidden layer is represented as the vector $a^{(t)}$ in (8), where W and U are the weight matrices for previous state $h^{(t-1)}$ and the present input $x^{(t)}$ respectively summed with the bias b_a. The state of the hidden layer $h^{(t)}$ at the present time instant is obtained by applying the hyperbolic tangent activation function to $a^{(t)}$. [8,374.]

$$a^{(t)} = Wh^{(t-1)} + Ux^{(t)} + b_a \quad (8)$$

$$h^{(t)} = tanh(a^{(t)}) \quad (9)$$

The values produced by the newly calculated hidden layer $h^{(t)}$ are then found using equation (10), where V is the weight matrix and b_o is the bias. Finally, the output vector can be obtained with (11) by passing the intermediate value to the SoftMax activation function. [8,374.] The activation need not necessarily be SoftMax or the hyperbolic tangent, any differentiable function suffices in these examples.

$$c^{(t)} = Vh^{(t)} + b_o \tag{10}$$

$$o^{(t)} = SoftMax(c^{(t)}) \tag{11}$$

The loss function (12) for the whole sequence x with respect to the label sequence y is the sum of all the losses for each time step t [8,374].

$$L_{total} = \sum_{t=0}^{\tau} L(o^{(t)}, y^{(t)}) = \sum_{t=0}^{\tau} L^{(t)} \tag{12}$$

Backpropagation for recurrent neural networks is known as backpropagation through time (BPTT), which simply applies the above equations (8) - (12) with the regular back-propagation on a computation graph unfolded through time as shown in Figure 4 on the right [8,376].

There are some challenges with the RNN, namely the vanishing or exploding gradient problem. This is where the gradients propagated through several timesteps tend to zero or explode beyond the representation capability of the computer. [8,396.] Considering the initial value $x^{(0)}$ of a single node with a weight w in equation (13) the problem can be easily understood. As the number of time steps n tends to infinity, the value $x^{(n)}$ vanishes or explodes as shown in (14) and (15) respectively.

$$x^{(n)} = w^n x^{(0)} \tag{13}$$

$$\lim_{n\to\infty} x^{(n)} = 0 \text{ , w < 1} \tag{14}$$

$$\lim_{n\to\infty} x^{(n)} = \infty \text{ , w > 1} \tag{15}$$

For this reason, the elementary forms RNNs are limited in their performance on long sequences. It is not impossible to learn patterns from the part of the parameter space with vanishing gradients, but the amount of computation required makes it a difficult and long process [8,398].

2.3.3 Long-Short Term Memory

A solution to the vanishing or exploding gradient problem is a variant of the RNN known as the long-short term memory (LSTM) neural network. The principle is very similar, but instead of one parameter to learn, there are four, and in addition to the hidden state for the layer, there is a hidden state for each cell. The additional nodes contained within the cell can perform alterations to the hidden state through gating mechanisms depending on the kind of patterns that are captured by the parameters. Since the network can remember the hidden state for arbitrarily long time, patterns of any length may be learned. [6,187;8,404.] Figure 5 shows a single LSTM cell that could replace a single node in a traditional RNN [6,189].

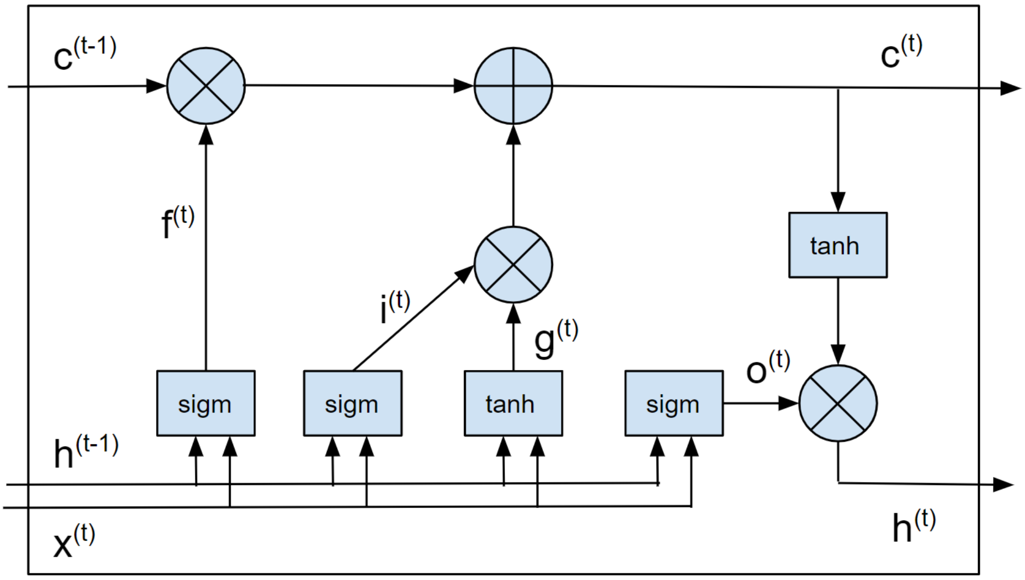

Figure 5. A LSTM cell. Redrawn from Deep Learning with Keras [6,197].

The feedforward and backpropagation are done through each timestep in a similar manner as it is for the RNN, but more parameters need to be computed due to the gating mechanisms shown in Figure 5 [8,412;8,188]. The input gate $i^{(t)}$ controls how much of the new state $g^{(t)}$ is remembered in the cell state $c^{(t)}$. Conversely, the forget gate $f^{(t)}$ controls how much of the previous cell state $c^{(t-1)}$ is forgotten by adjusting its magnitude. Finally, the output gate $o^{(t)}$ controls the magnitude of the hidden state $h^{(t)}$. Considering the gates as vectors, the values can be calculated with (16) - (19). Since they are gating other values, the sigmoid activation is used to obtain a scalar value in range 0-1. [6,188.]

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)}) \tag{16}$$

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)}) \tag{17}$$

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)}) \tag{18}$$

$$g^{(t)} = tanh(W_g h^{(t-1)} + U_g x^{(t)}) \tag{19}$$

The new cell state $c^{(t)}$ is calculated with (20) and the new hidden state $h^{(t)}$ with (21).

$$c^{(t)} = c^{(t-1)} f^{(t)} + g^{(t)}\, i^{(t)} \tag{20}$$

$$h^{(t)} = tanh(c^{(t)}) o^{(t)} \tag{21}$$

As the LSTM cell directly replaces the nodes in the RNN, the BPTT algorithm is implemented by replacing the equations (8) - (11) with the equations (16) - (21) for LSTM networks.

2.3.4 Autoencoder

Autoencoders are neural networks with equally sized input and output layers and one or more hidden layers. They can be implemented using either feedforward or recurrent network topologies [6,225]. The hidden layers are, describing a code which represents some fundamental relationships of the input data. Structurally, the autoencoder shown in figure 6 is like a regular feedforward neural network, the major difference being that its purpose is to output a reconstructed version of the input rather than some new information based on it. As such, they can be considered a special case of the basic feedforward neural networks with all the same training techniques applying to them [8,499].

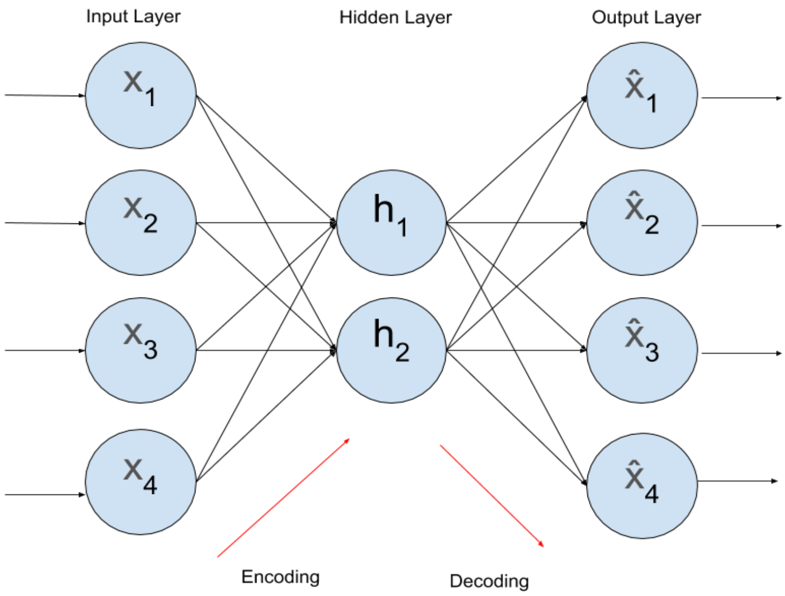

Figure 6. A simple autoencoder.

The architecture of the autoencoder can be divided into two components, an encoder and a decoder. The former is used to compress the information from the input vector x into a lower dimension for the hidden layer h, also known as latent- or the code layer. Conversely the latter section recreates the input x from the compacted data by applying the inverse operation on the hidden layer h to create the reconstruction x̂. [8,499-500.] In some applications, the output of the trained autoencoder is employed as it is, but often the decoder section is discarded, and the encoder is used to encode information from input data or to compress it into a more convenient form [6,224].

Some structural limitations are imposed in order to force the model to learn useful features from the data rather than just the identity function. For instance, a hidden layer with smaller dimension than that of the input layer may be chosen, this is known as an undercomplete autoencoder. A more complex autoencoder network can be built by using regularized autoencoders, which are overcomplete, but capable of learning useful information from the data. These autoencoders utilize loss functions, which penalize too high activations in the hidden layer to constrain the information flow. Another technique is the denoising autoencoder, which is trained on a corrupted version of the input data. Both methods give rise to desirable properties such as robustness to noise or missing data points. [8;505-509.]

3 Tools

This section gives a brief explanation of the tools used. It is intended to serve only as an introduction to them and the relevant terminology. Any greater technical detail will be given as necessary in the later sections.

3.1 Azure

Microsoft Azure, initially released under the name Windows Azure in 2010, is a cloud computing platform developed by Microsoft. It offers its users an option of provisioning computing resources as needed, thus eliminating the time-consuming burden of hosting and managing servers on premises. Their infrastructure with data centres all around the world has established availability everywhere. The services range from complete software managed for the customer known as Software-as-a-Service (SaaS), to Platform-as-a-Service (PaaS) for running custom software on ready-made environments and finally Infrastructure-as-a-Service (IaaS) for utilizing hardware managed by Microsoft in their data centres. For developers the platform offers storage and database services, tools for developing and deploying applications as well as AI and machine learning tools, in which Microsoft has recently invested substantially. [9.]

3.1.1 Machine Learning in Azure

Azure includes a variety of options for utilising machine learning, which may be chosen depending on the level of control and the type of deployment target that the user wants. One of those options, the Azure Cognitive Services offers pre-trained, but tailorable models for specific tasks in vision, speech, language, searching and decision making. They are easily accessible through APIs and allow quick implementation without extensive knowledge about machine learning. [10.] However, in complex applications, with detailed requirements, the limitations of the Cognitive Services emerge quickly, as they are designed for generalized solutions and do not allow selection of algorithms or training on specific data. [11.]

For more control, Azure offers two approaches. First of them is using the Machine Learning Studio, a graphical, “zero code” tool for creation of ML models with minimal effort and the aid of its visual drag and drop-interface. An option with more flexibility is creating the machine learning model with industry standard open source Python packages like Keras, Tensorflow and Scikit-learn and then deploying to the cloud using Azure Machine Learning Service. [12.]

3.1.2 Azure Notebooks

Azure Notebooks provides a convenient way to run Jupyter Notebooks, a web-based programming environment for Python, R and various other languages. It comes with the basic dependencies such as numpy for mathematics and matplotlib for plotting, as well as the most commonly used machine learning packages such as Keras and Scikit-learn. In most cases no installations are needed at all. It runs on the free computation resource provided by Microsoft Azure, with the option of increasing the performance with Azure subscription. They are ideal for quick prototyping but can be used for more extensive development as they provide all the necessary functionality for most tasks. [13.] All the code in this project is written in an Azure Notebook using Python 3.6 with Anaconda3-5.3.0.

3.2 Keras

Keras is a high-level programming library developed for building and training neural networks. It is the official frontend for TensorFlow, and directly after TensorFlow is also the second most used deep learning framework for industry and research. It abstracts the complex task of building neural nets into simple API calls and minimizes the amount of code required for the most common neural network development tasks. It is designed to be as human-readable as possible and provides a quick way to start prototyping without needing much prior programming experience. Regardless of the simplicity, it provides great flexibility as it supports multiple backend engines such as TensorFlow and Microsoft Cognitive Toolkit. Furthermore, it allows directly calling the lower-level languages and consequently provides the same level of control. [14.]

3.3 ONNX

Open Neural Network Exchange (ONNX) is an open format for representing deep learning models. It is intended to provide interoperability between the increasing number of platforms and tools that exist. Training and tuning machine learning models is very time consuming, computationally expensive and often requires high-end hardware. On the other hand, once deployed, the model should be able to run on a variety of computation environments and would be better if optimized for the lower performance of most devices. Moreover, every development tool has their own benefits, and some are better suited for building neural networks for certain applications than others. Therefore, it can be advantageous to implement and train the solution in the preferred environment with the preferred tools before transferring it to cloud or the physical device as needed. Each of the tools comes with their own mutually incompatible representations of neural networks, which may be interfaced using the common format that ONNX provides. [15;16.] Figure 7 shows some of the supported platforms and tools, including the Azure Machine Learning services and Keras used in this project.

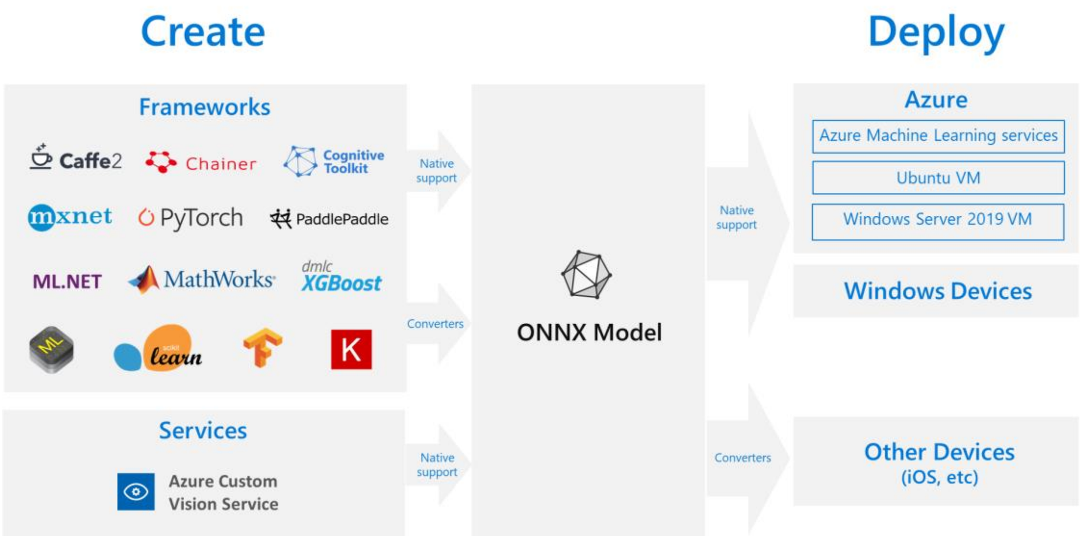

Figure 7. Some of the platforms and tools with ONNX support. Reprinted from Azure documentation [16].

ONNX runtime is a powerful inference engine based on the ONNX-format. It is designed to perform well in the cloud as well as on the local hardware. With the support Azure provides for it, deployment as a web-service to the cloud is straightforward. Additional benefit of using ONNX Runtime for inference is better performance, which Microsoft has reported doubling on average in some of their applications. [16.]

4 Methods

This section describes the specifics on the implementation of the analysis tool. The source code for relevant functionality is provided as appendices, which are separated into training, deployment and testing scripts, as well as dependencies for them. Many of the Illustrations are generated with matplotlib in python, but the source code for that is omitted due to its length and irrelevance to the objective. To avoid repetition, some of the functions used in multiple places are defined separately in appendix 1 along with parameters used and a full list of the imported packages.

4.1 Analysis of Bearing Degradation

A bearing vibration experiment published by the Centre for Intelligent Maintenance Systems (IMS) was selected, as the data it contains is very similar to that produced by the VibLog measurement system. The data set is collected from a test jig with four bearings being rotated at constant speed of 2000 RPM under 6000 lbs load. It consists of three smaller datasets of individual runs until failure. [17.]

Each dataset contains several CSV-files of 20478 samples measured over one second every 10 minutes. Units of measurement are not given in the documentation but are assumed to be meters per second squared (m/s^2). The unit is not relevant as the data will be normalized in the pre-processing for the neural network. The datasets are very similar to each other, apart from the number of measurement channels. In the first dataset both x- and y-axes are recorded, whereas only a single axis measurement is used for the other two. [17.]

Observing the contents of the data sets plotted with python reveals that the first and third measurement runs contain many short anomalous bursts and irregularities, while the second run gives a smooth and stable graph until the degradation begins. All the runs exhibit similar exponential increase in RMS amplitude towards the failure, in which the amplitude rises to approximately 0.5 m/s^2 over a period of 3-4 days. The second dataset, plotted in Figure 8, was chosen for further analysis and experimentation while the other two were discarded.

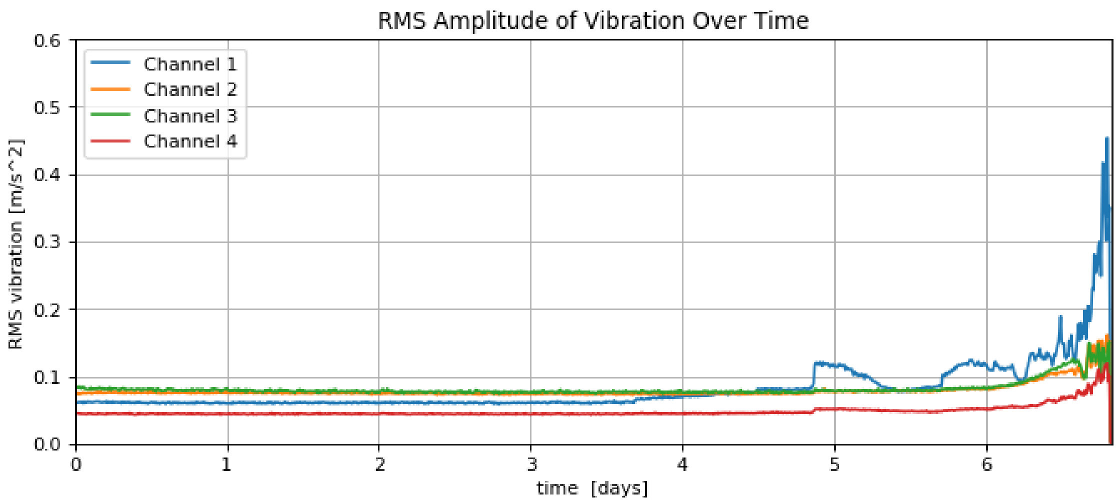

Figure 8. A measurement run until failure from the dataset.

Figure 9 compares the raw vibration of a single bearing soon after beginning the experiment and 40 hours prior to failure. The peak amplitudes undergo a significant threefold increase as the bearing degrades. This seems to be the result of the high frequency components increasing in their amplitude. The same lower frequency vibration is noticeable from both signals in the time domain representation.

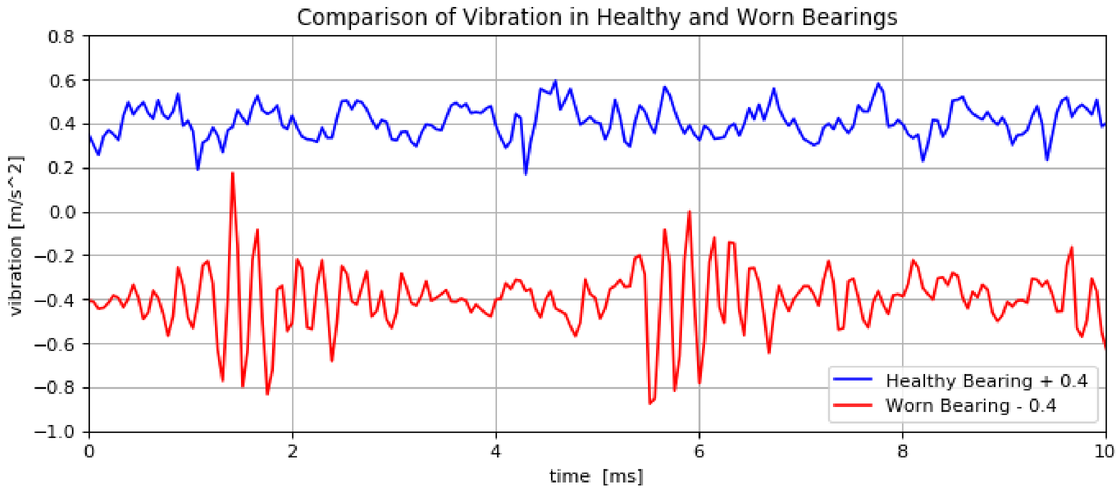

Figure 9. Raw vibration measurement from a single bearing. The worn bearing has approximately 40 hours of lifetime left. Bias is added for clarity.

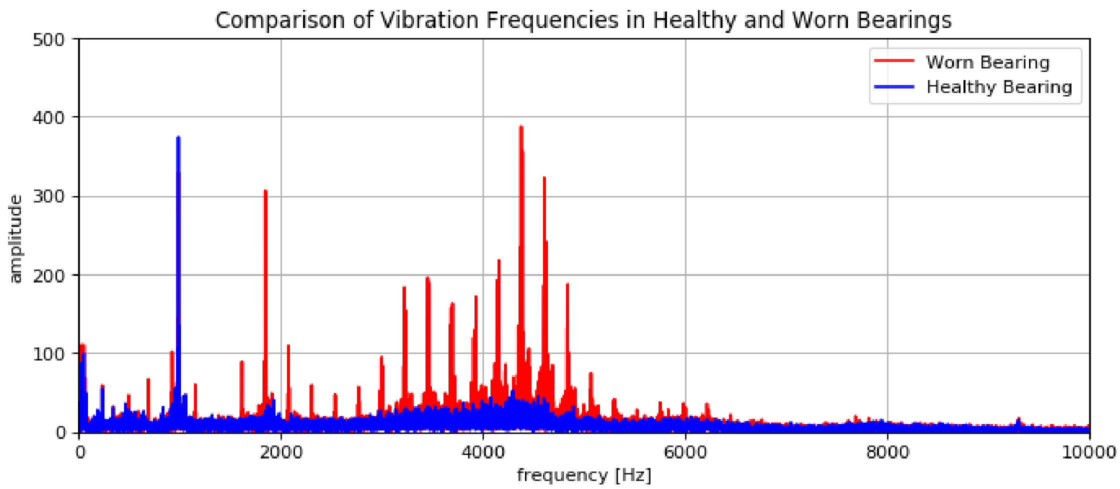

Figure 10. Frequency spectrum of a raw measurement. The worn bearing has approximately 40 hours of lifetime left.

Observing the frequency spectrum in figure 10 confirms that the high frequency components are increasing in amplitude and quantity as the breakage is nearing. In the healthy state of the bearing, there is clearly one main frequency component around 1 kHz, which stays approximately constant in amplitude over the degradation process. As previously noted, the higher frequency components have a much stronger presence in the signal measured from a worn bearing. Using Parseval's theorem for energy signals (22), it is also clear from the frequency domain representation that the total energy in the system increases as more high frequency components appear.

$$E = \int |X(f)|^2 \, df \qquad (22)$$

From these observations it can be concluded that the high frequency vibrations in the bearing increase as its condition degrades and the total energy in the system grows accordingly. Developing a method to successfully detect these trends from the measured data makes it possible to determine if the bearings are vibrating in a normal manner as well as to quantify the degree in which they are worn out for approximating the remaining lifetime.

4.2 Developing a Solution

Based on the previous section, the problem is to detect the changes over time in the signal. This can be achieved if a function is capable of reproducing the vibration signal of healthy bearings, while being incapable of doing the same for the worn-out bearings. A method proposed to solve this problem is applying the sequence to sequence LSTM autoencoder for quantifying the degradation. It has been shown that a basic autoencoder alone can be sufficient to detect degradation in bearing vibration data [18]. However, since the data being analysed is in time-series, leveraging the ability of recurrent neural networks to capture temporal structures may give a better prediction and therefore warn about potential problems earlier and in greater accuracy. In this solution, no averaging or other transformations leading to information loss are applied on the data.

4.2.1 LSTM Autoencoder

The operation of a sequence to sequence LSTM autoencoder is illustrated in Figure 11. First, the continuous input sequence is broken into finite length batches $x^{(i)}$ and pre-processed. For each batch, the neural network attempts to reconstruct the observed signal $X^{(i)}$ as $\hat{X}^{(i)}$ but is unable to do so perfectly when encountering unseen patterns resulting in some error $E^{(i)}$. This reconstruction error serves as a metric of how different the data being inferred from at present is in comparison to the data on which the neural network has been trained on previously.

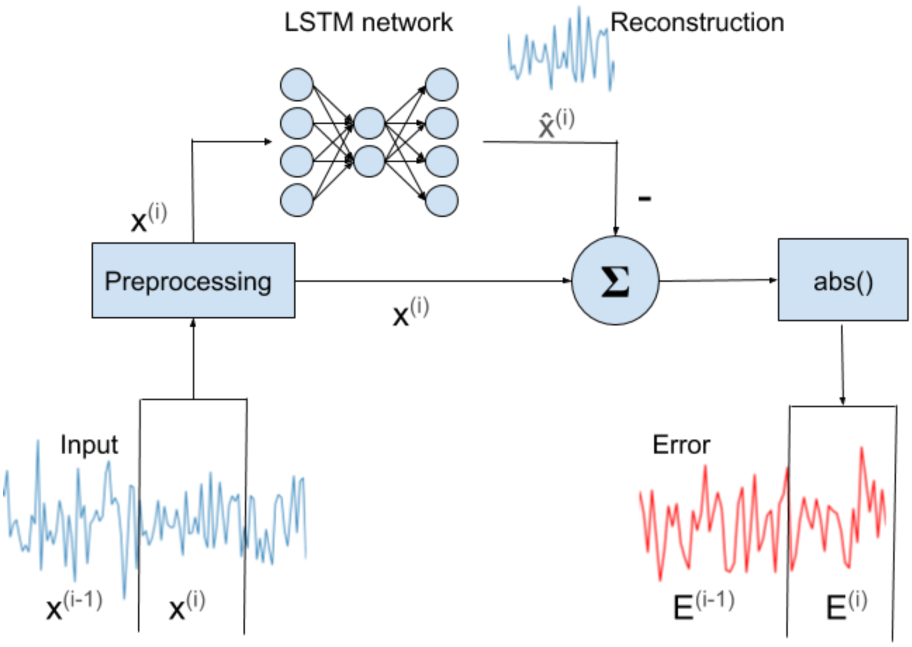

Figure 11. The concept of the sequence to sequence LSTM Autoencoder.

The neural network structure and the hyperparameters were chosen somewhat arbitrarily based on experimentation with different configurations. In general, there are no formal rules for finding the suitable architecture and the solution is often based on educated guesses and optimization techniques, such as exhaustively searching through the space of possible parameters [7,263].

The structure consists of three LSTM layers and does not incorporate any compression in the temporal dimension. This is because such reductions in time dimension led to the neural network being unable to learn anything useful from the data and furthermore, the basic LSTM structures with any compression at all yielded similar performance as those without it.

4.2.2 Building and Training the Model in Keras

The appendix 2 contains the code used for building and training the neural network and shows the specific parameters and architecture used. The training script can be run after appendix 1.

The first step is loading and pre-processing the data set. The training data directory contains a selection of csv-files depicting the vibration signals produced by healthy bearings from the data set [17]. The function load_data combines the contents of the directory into a single array for training. Very large inputs or significant differences in the ranges of the individual data channels may prevent the network from converging, so the values are normalized using the MinMaxScaler from sklearn preprocessing-library [7,101]. The scaler is then saved so that the data used for predictions later can be normalized with respect to the training data.

The first layer of the network provides information about the dimensions of the data and the number of samples used for each parameter update by the learning algorithm. This is followed by three LSTM layers with 16, 4 and 16 nodes respectively, each using hyperbolic tangent activation function. A densely connected output layer is added at the end to return the original shape of the data. To make the predictions of the network more general, some dropout is added to each layer. This means that random elements from the output vector are set to zero or “dropped out” [7,109-110]. In addition to this, the LSTM layers must be set as stateful. Normally the hidden internal states would be reset after every batch of data. However, this is not what is wanted when trying to reconstruct a sequence from small batches. The internal values of the LSTM cells should be maintained to keep the information about previous states.

LSTM-layers require three dimensional arrays as inputs; therefore the data is reshaped into a suitable form with the reshape_input-function prior to feeding it to the neural network. The model is then trained 50 times with the training data set, after which no more noticeable improvement occurs.

4.2.3 Measuring Bearing Degradation with the Trained Model

The trained neural network was tested with previously unseen data pre-processed in the same manner as the training data. Figure 12 shows the vibration signal from healthy bearing being compared to the reconstruction made by the neural network and the resulting error. Clearly the reconstruction can approximate the vibration quite well. A few anomalous spikes are present, but they show up as increased error as expected and on average, the reconstruction error is low.

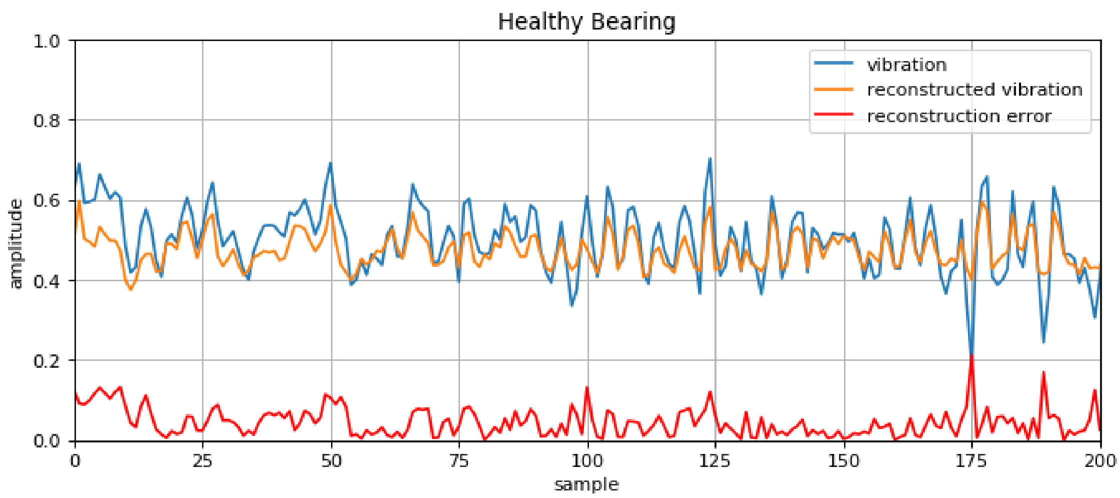

Figure 12. Vibration of a healthy bearing and the reconstructed signal.

Comparing the worn bearing approximately 40 hours prior to failure shown in figure 13 to the healthy bearing in figure 12, much more error is present. The neural network struggles to reconstruct the high amplitudes and rapid changes that were absent from the healthy vibration signal it was trained on. Overall, the reconstruction error is much higher as a result of these anomalies and interestingly, their locations can even be found accurately from the error vector.

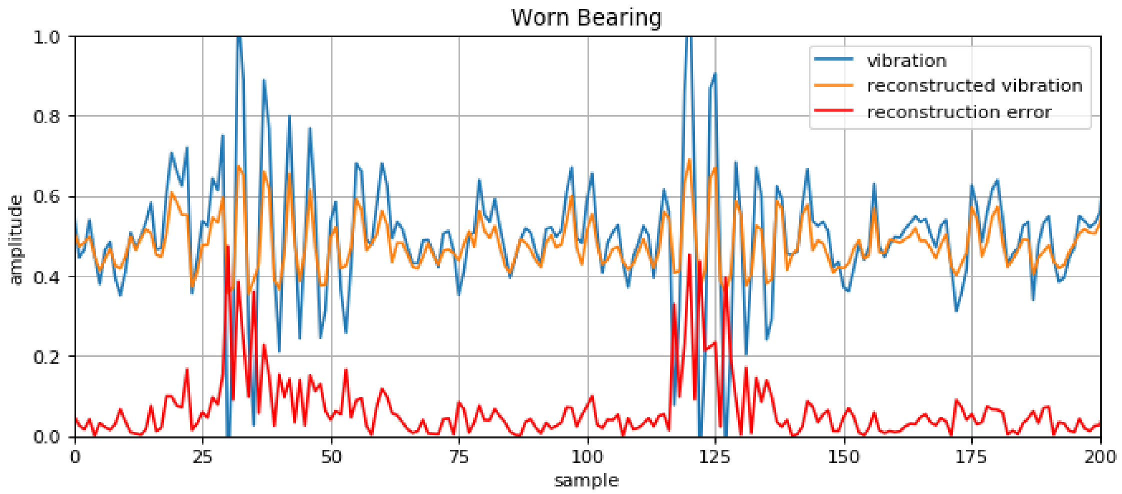

Figure 13. Vibration of a worn bearing and the reconstructed signal approximately 40 hours prior to failure.

The probability distribution of the reconstruction error for a healthy bearing is shown in Figure 14. To classify a point as an anomaly or an outlier, a threshold had to be chosen. Most of the data points are contained within three standard deviations, so that was chosen as a rudimentary boundary. The mean reconstruction error of healthy bearing data on this dataset with the trained model is 0.042 and the standard deviation is 0.036 resulting in threshold of 0.149.

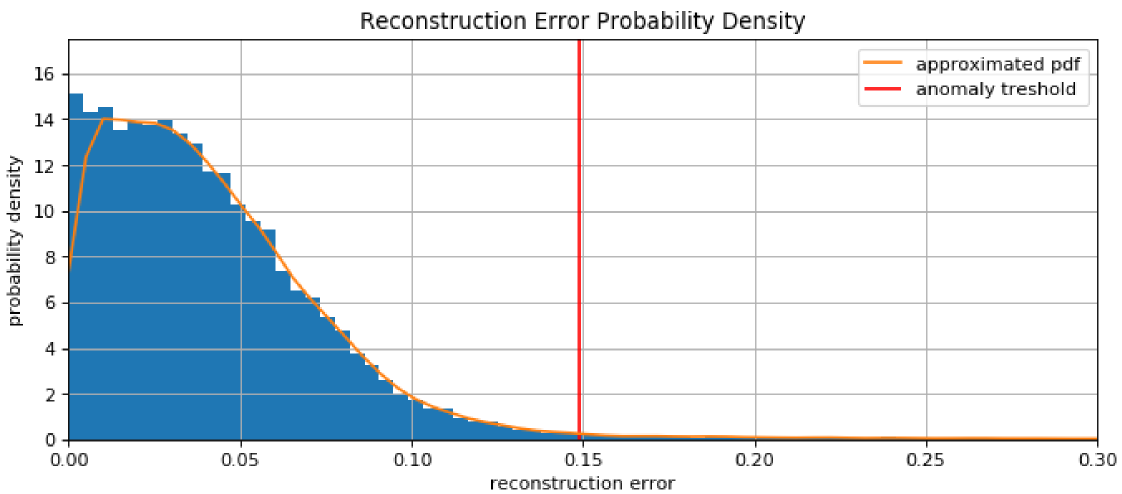

Figure 14. Approximated probability distribution function of the vibration signal from a healthy bearing with the anomaly threshold at three times the standard deviation.

Results above give a good indication of the anomalies. However, averaging these values and observing their change over time is a better way of evaluating the condition of the bearings. The reconstruction error E_{batch} for single batch measurements is then calculated as the mean of all features and samples with (23), where $E_{i,j}$ is the reconstruction error for sample i on feature j, $N_{samples}$ is the number of samples and $N_{features}$ is the number of features in the sequence.

$$E_{batch} = \frac{1}{N_{samples}} \sum_{i=0}^{N_{samples}} \left(\frac{1}{N_{features}} \sum_{j=0}^{N_{features}} (E_{i,j}) \right) \quad (23)$$

Calculating (23) for a batch of 20000 samples recorded at 20478 Hz every 2 hours over the whole data set yields a relatively smooth curve depicting the degradation over time as shown in Figure 15. The anomaly threshold here is calculated for the new vector of E_{batch} separately using the corresponding standard deviation.

Figure 15. Reconstruction error averaged from validation data batches.

The problems can be seen more than three days prior to the failure, at the rate of 2000 RPM this corresponds to around 10 million rotations. Furthermore, the same exponential growth over 3-4 days is observed as in the RMS vibration data. Based on this, clearly the degradation of the bearings can be detected quite reliably and quantified for predicting their lifetime.

4.3 Operationalization

This section describes the process of deploying the neural network model as a web-service in the Azure cloud. Figure 16 outlines the steps that were taken for a successful operationalization and lists the dependencies that must be provided. [19;20.] In addition to the trained model, an Azure subscription with sufficient rights to create workspaces is required.

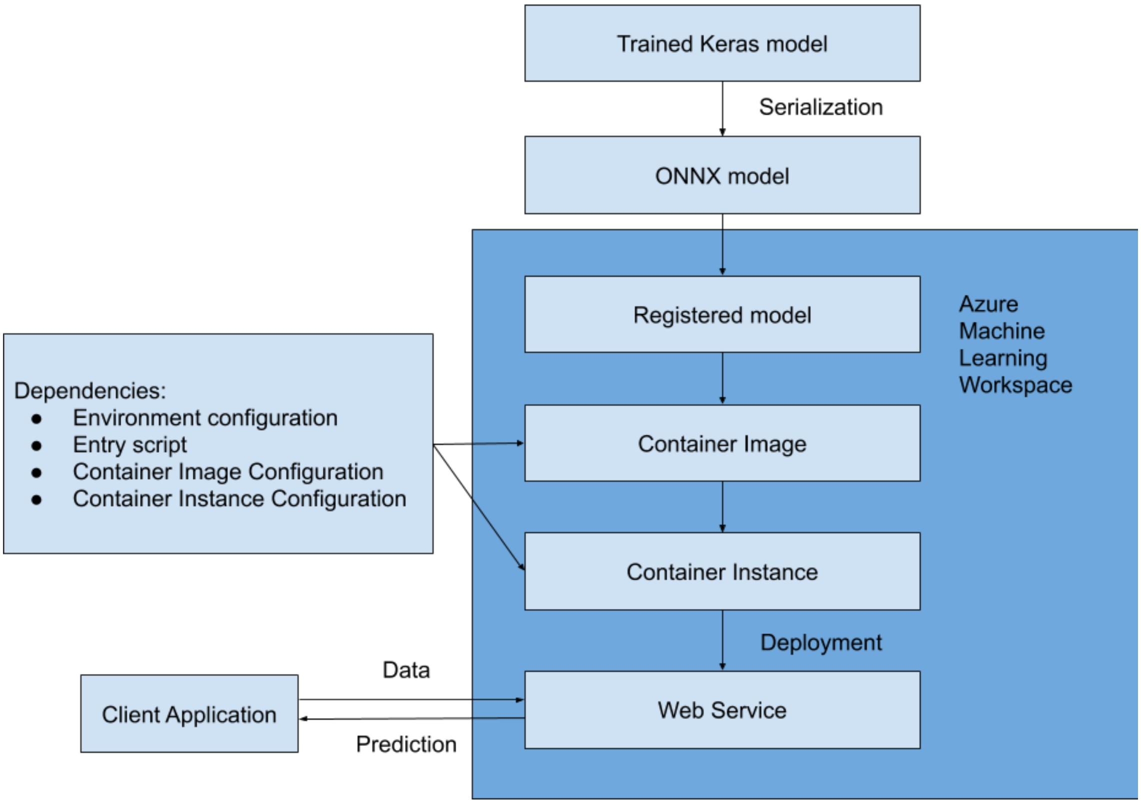

Figure 16. Operationalizing a Keras-trained neural network model as a web-service.

After a successful deployment, the web-service is accessible through a REST API and new predictions can be made from virtually any device on any programming language. The client application in figure 16 must be provided with the uniform resource identifier (URI) pointing to the REST API address and the authentication key if authentication is configured for the web service. [21.]

4.3.1 Deploying as a Web Service

Appendix 3 contains the code for deploying the neural network model and shows the parameters used in the process. The deployment script can be run after appendices 1 and 2.

Before the neural network can be deployed, it must be converted into a form compatible with the Azure Machine Learning service. The model can be serialized into ONNX-format by using the functions provided in the onnxmltools-library. [20.] In some cases, the weight mappings of the models might not get converted correctly and other functionality supported by Keras might not work identically in ONNX, leading to inconsistent inference results between the two runtimes. The predictions of the Keras model should be compared with those made by the ONNX-model to confirm expected behavior.

A workspace object is required for managing the resources created to Azure. It provides methods for manipulating the workspace such as deleting it to prevent charges when it is no longer needed. The workspace itself is created automatically along with the object, but this may also be done via the Azure Portal web-interface. The workspace is created in a resource group, which is a container for grouping multiple related workspaces. After registration to the machine learning workspace, the model is available on Azure cloud and can be accessed for deployment or download. This provides a convenient way of version control as registering another model with the same name will increment the version number and additional metadata such as a python dictionary of tags and properties can be given when registering [22].

Azure container Instances can be used for deploying the trained model into the cloud. This requires a few configurations to be made as shown in figure 16 under dependencies. The entry script, given in appendix 4 is run automatically as the web service starts. It initializes the model and acts as a common interface between the web service and the incoming requests. Two functions must be provided for it, an init function for loading the model and performing any other required initialization, as well as a run function for reformatting the data from received requests, performing inference and returning the result in a suitable format. The web service uses the JSON-format for passing requests and responses. This requires some additional transformations as numpy arrays used by the model are not serializable to JSON. [19;20.]

The environment configuration describes the external packages utilized by the entry script. By default, it includes some common dependencies and any additional packages can be included here. The container image configuration provides information on how to run the model for the web service, here it associates the entry script and the environment configuration file with the container instance. [19;20.]

With all the dependencies provided, the image can be deployed. The deploy_configuration-function defines the computational resources and other parameters related to the virtual machine given to this web-service. The hardware requirements depend on the specific application, here a simple configuration with 1 GB of memory and one CPU core is defined before the web-service is deployed with the deploy_from_image-function. [19;20.]

4.3.2 Using the Analysis Tool

The test script in appendix 5 demonstrates the functionality of the deployed web-service. It can be run after successfully executing the scripts in appendices 1-3.

The web_service_predict-function breaks the input array into the correct batch size for the deployed neural network model and calls the send_request-function to obtain reconstructed arrays for each batch. The send_request-function transforms the data into the JSON-format and makes a HTTP-request to the web-service. The predictions are combined back into the original shape and returned to be evaluated by the score_output-function, which calculates the mean reconstruction error for the whole input to quantify the degradation of the bearing.

```
time: 2004.02.12.10.32.39, reconstruction error: 0.0539
time: 2004.02.13.06.32.39, reconstruction error: 0.0556
time: 2004.02.14.02.32.39, reconstruction error: 0.0554
time: 2004.02.14.22.32.39, reconstruction error: 0.0552
time: 2004.02.15.18.32.39, reconstruction error: 0.0549
time: 2004.02.16.14.32.39, reconstruction error: 0.0579
time: 2004.02.17.10.32.39, reconstruction error: 0.0694
time: 2004.02.18.06.32.39, reconstruction error: 0.0705
time: 2004.02.19.02.32.39, reconstruction error: 0.1016
```

Listing 1. Responses from the web service for validation data.

The testing script functions as follows. First, the program loads an array of validation data from the dataset [17]. The data is pre-processed into the correct shape in the same manner as the training data using the previously saved scaler. The saved web-service address is read and used to make requests to the deployed analysis tool. Listing 1 shows the terminal output with reconstruction errors calculated for the validation data. The values match the locally calculated numbers in figure 15 as expected.

5 Results

The goal set at the beginning of this project was reached. The analysis tool developed is very simple and crude but works well within the confines of the type of data it was experimented with. The tight schedule that this project was constrained to, severely limited the standards at which the work could be carried out, leaving a great deal of room for improvement on the result. However, the methods for implementing a neural network model for analysing bearing vibration data and deploying it as a web service were successfully found and can be easily extended into a more sophisticated solution.

The questions formulated at the beginning of the work can be now answered. Clearly, artificial neural networks can be used to reliably predict impending faults in rolling bearings given that there are noticeable anomalies in the vibration data. Furthermore, the degradation observed from the signals can be quantified and used for estimating the remaining lifetime of the bearings. The data available for this work did not contain examples of different rotational velocities. Therefore, it could not be confirmed, if the data from slowly rotating machinery would be more challenging to analyse. However, since LSTM autoencoder networks can learn patterns of arbitrary length and even a single spike in amplitude can be detected, this should not be a problem. The detected anomalies could be, for instance, compared to known failure patterns to classify them as different faults regardless of their frequency or length.

The degradation pattern seen in the RMS vibration is very similar in shape to the reconstruction error made by the analysis tool. This raises the question if such a complex solution is necessary to solve this problem. However, the neural network learnt to identify this relationship between the degradation in bearings and the increase in energy, without it being explicitly told to do so. Therefore, failures can be foreseen without having to destroy any equipment to generate examples of such events.

6 Conclusions

The goal was to implement a cloud-based analysis tool for extending the functionality of the VibLog measurement system. A suitable method to do this was found and a functional solution developed for the available data set. This work serves as a proof of concept and a starting point to the subsequent projects. The result is far from finished and in order to successfully adapt this functionality to VibLog, much more development and testing is required.

The analysis tool could be for instance improved by a more thorough research into the different neural network architectures available. In this application, very generic solution suitable for multiple different systems would be preferable. Only seldom do machines rotate at a constant speed until failure, therefore the analysis tool would have to take different configurations and rotational velocities into account.

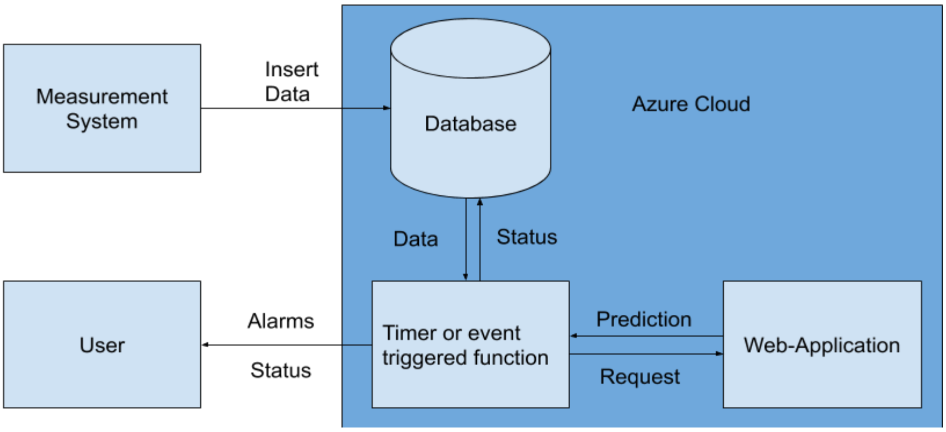

Figure 17. The system diagram of a possible extension to the VibLog Analysis Tool web-application.

Figure 17 shows a possible extension to this work. The measurements being streamed to a database could be sent for analysis based on a timer or some other trigger, such as a certain amount of data accumulating. The result of the analysis could be used to create alerts and status information to be shown on a dashboard or emails sent to personnel responsible for the equipment.

References

1 Lace. The Role of Vibration Monitoring in Predictive Maintenance [online]. URL:https://www.schaeffler.com/remotemedien/media/_shared_media/08_media_library/01_publications/schaeffler_2/technicalpaper_1/download_1/the_role_of_vibration_monitoring.pdf. Accessed 27 October 2019.

2 Yung. Vibration analysis: what does it mean? [online]. URL:https://www.plantservices.com/articles/2006/154. Accessed 27 October 2019.

3 National Instruments. Measuring Vibration with Accelerometers [online]. URL:https://www.ni.com/en-tr/innovations/white-papers/06/measuring-vibration-with-accelerometers.html. Accessed 27 October 2019.

4 Rojas. Neural Networks: A Systematic Introduction. Springer-Verlag; 1996

5 Russel, Norvig. Artificial Intelligence, A Modern Approach. Third Edition. Prentice Hall; 2010

6 Chollet. Deep Learning with Python. Manning: 2018

7 Gulli, Pal. Deep Learning with Keras. Packt: 2017

8 Goodfellow, Bengio, Courville. Deep Learning. MIT Press: 2016

9 Bott. Microsoft Azure: Everything you need to know about Redmond's cloud service [online]. URL: https://www.zdnet.com/article/microsoft-azure-everything-you-need-to-know. Accessed 27 October 2019.

10 Microsoft. Cognitive Services and machine learning [online]. URL: https://docs.microsoft.com/fi-fi/azure/cognitive-services/cognitive-services-and-machine-learning. Accessed 27 October 2019.

11 Basak, Shah, Lehman, Abraham, Stirrup. Hands-On Machine Learning with Azure. Cognitive Services/bots. Packt: 2018

12 Microsoft. What is Azure Machine Learning? [online]. URL:https://docs.microsoft.com/fi-fi/azure/machine-learning/service/overview-what-is-azure-ml. Accessed 27 October 2019.

13 Microsoft. Azure Notebooks Documentation [online]. URL:https://docs.microsoft.com/en-us/azure/notebooks. Accessed 27 October 2019.

14 Keras. Keras Documentation [online]. URL:https://keras.io. Accessed 27 October 2019.

15 ONNX. ONNX Homepage [online]. URL:https://onnx.ai. Accessed 27 October 2019.

16 Microsoft.ONNX and Azure Machine Learning: Create and accelerate ML models [online]. URL:https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-onnx. Accessed 27 October 2019.

17 Lee, Qiu, Yu, Lin, Rexnord Technical Services. Bearing Data Set [online]. URL:https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository. Accessed 27 October 2019.

18 Flovik. How to use machine learning for anomaly detection and condition monitoring [online]. URL:https://towardsdatascience.com/how-to-use-machine-learning-for-anomaly-detection-and-condition-monitoring-6742f82900d7. Accessed 27 October 2019.

19 Microsoft. Deploy models with Azure Machine Learning [online]. URL:https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-deploy-and-where. Accessed 27 October 2019.

20 Keen. Deploying Neural Network models to Azure ML Service with Keras and ONNX [online]. URL:http://benalexkeen.com/deploying-neural-network-models-to-azure-ml-service-with-keras-and-onnx. Accessed 27 October 2019.

21 Microsoft. Consume an Azure Machine Learning model deployed as a web service [online]. URL:https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-consume-web-service. Accessed 27 October 2019.

22 Microsoft. MLOps: Manage, deploy, and monitor models with Azure Machine Learning [online]. URL:https://docs.microsoft.com/bs-latn-ba/azure/machine-learning/service/concept-model-management-and-deployment. Accessed 27 October 2019.

Initialization script

```
'''
Import all the necessary libraries and define parameters, as well as
utility functions used in the other scripts.
'''

#imports
import pandas as pd
import numpy as np
import os
import json
import requests
import onnxmltools
import onnxruntime as onnxrt
from azureml.core import Workspace
from azureml.core.model import Model as AzureModel
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.image import ContainerImage
from azureml.core.webservice import AciWebservice, Webservice
from keras.models import load_model, Model
from keras.optimizers import RMSprop
from keras.layers import LSTM, Dense, Input
from sklearn.preprocessing import MinMaxScaler
from sklearn.externals import joblib

# Parameters
n_features = 4
data_length = 20000
n_timesteps = 50
batch_size = 100
np.random.seed(1234) # Fix random seed for reproducibility

# Utility function definitions

'''
Read a folder full of CSV-files into a numpyarray.
'''
def load_data(directory):
    container = []
    files = os.listdir(directory)
    files.sort()
    for i, file in enumerate(files):
        print(file, i+1, '/', len(files))
        data = pd.read_csv(os.path.join(directory, file),
           sep='\t',header=None)
        data = np.array(data, dtype=np.float32)[:data_length, 1:]
        container.append(data)
    container = np.array(container).reshape(len(con-
tainer)*data_length, n_features)
    return container, files
```

```
'''
Reshape the input to 3D. X is the 2D array (samples, channels).
'''
def reshape_input(input_data):
    X = input_data
    len_req = batch_size * n_timesteps
    if len(X) < len_req:
        raise Exception('Input array too small, size {}, need at least
{}'.format(len(X), len_req))
    if not len(X) % n_timesteps:
        X = X[:int(np.floor(len(X)/n_timesteps))*n_timesteps]
    if not len(X) % batch_size:
        X = X[:int(np.floor(len(X)/batch_size))*batch_size]
    input_reshaped = X.reshape(int(len(X)/n_timesteps), n_timesteps,
n_features).astype(np.float32)
    return input_reshaped
```

Training script

```
'''
Define and train the neural network model.
'''

# Load data
training_data = load_data('Data/training_data')

# Scale data to [0,1] with respect to training data
scaler = MinMaxScaler(feature_range=(0, 1))
training_data = scaler.fit_transform(training_data)
joblib.dump(scaler, 'Outputs/scaler.save') # Save scaler

# Define the NN model
input_layer = Input(batch_shape=(batch_size, n_timesteps, n_features))
lstm = LSTM(16, activation="tanh", dropout=0.2, recurrent_dropout=0.2,
return_sequences=True, stateful=True)(input_layer)
lstm = LSTM(4, activation="tanh", dropout=0.2, recurrent_dropout=0.2,
return_sequences=True, stateful=True)(lstm)
lstm = LSTM(16, activation="tanh", dropout=0.2, recurrent_dropout=0.2,
return_sequences=True, stateful=True)(lstm)
output_layer = Dense(4)(lstm)
model = Model(input_layer, output_layer)
optimizer = RMSprop(lr=0.001, rho=0.9)
model.compile(optimizer=optimizer, loss='mse')
model.summary()

# Train the NN model
X_train = reshape_input(training_data)
history = model.fit(X_train, X_train, batch_size=batch_size,
epochs=50, verbose=1, shuffle=False, validation_split=0.25)

# Save the NN model
model.save('Outputs/LSTMAE.h5')
```

Deploy script

```
'''
Deploy the neural network to the Azure cloud.
The entry script must be provided.
'''

# Convert keras model to ONNX
model = load_model('Outputs/LSTMAE.h5')
onnx_model = onnxmltools.convert_keras(model)
onnxmltools.utils.save_model(onnx_model, 'Outputs/LSTMAE.onnx')

# Create a workspace
ws = Workspace.create(name='vibrationanalysis',
                      subscription_id=subscription_id,
                      resource_group=resource_group,
                      create_resource_group=True,
                      location=location
                     )

# Register the model
azureml_model = AzureModel.register(model_path = 'Out-
puts/LSTMAE.onnx', model_name = 'LSTMAE', workspace = ws)

# Create enviroment file with the required dependencies
dependencies = CondaDependencies()
dependencies.add_pip_package("numpy")
dependencies.add_pip_package("azureml-core")
dependencies.add_pip_package("onnxruntime")
with open("environment_config.yml","w") as f:
    f.write(dependencies.serialize_to_string())

# Create image configuration
image_config = ContainerImage.image_configuration(execution_script =
"entryscript.py", runtime = "python", conda_file = "environment_con-
fig.yml")

# Create container image
image = ContainerImage.create(name = "lstmaemodelimage", models = [az-
ureml_model], image_config = image_config, workspace = ws)
image.wait_for_creation(show_output = True)

# Deploy as web service
aciconfig = AciWebservice.deploy_configuration(cpu_cores = 1,
memory_gb = 1)
service = Webservice.deploy_from_image(deployment_config = aciconfig,
image = image, name = 'vibration-analysis', workspace = ws)
service.wait_for_deployment(show_output = True)
with open('Outputs/URI.txt', "w") as f:
    f.write(service.scoring_uri) # Save scoring uri
```

Entry script for the web-service

```
import json
import sys
import onnxruntime
import numpy as np
from azureml.core.model import Model

def init():
    global model_path
    model_path = Model.get_model_path(model_name = 'LSTMAE')

def run(raw_data):
    try:
        data = np.array(json.loads(raw_data)['data'],
dtype=np.float32)
        session = onnxruntime.InferenceSession(model_path)
        first_input_name = session.get_inputs()[0].name
        first_output_name = session.get_outputs()[0].name
        result = session.run([first_output_name], {first_input_name:
data})[0].tolist() # numpy arrays cannot be serialized.
        return {"result": result}
    except Exception as e:
        result = str(e)
        return {"error": result}
```

Test script

```
'''
Test the functionality of the deployed web-service.
'''

# Load and preprocess data
validation_data, timestamps = load_data('Data/validation_data')
validation_data = validation_data.reshape(int(len(valida-
tion_data)/data_length), data_length, n_features)
scaler = joblib.load('Outputs/scaler.save')
# Preprocess data
validation_data = [scaler.transform(data) for data in validation_data]
# Load previously saved uri
web_service_uri = ''
with open('Outputs/URI.txt') as f:
    web_service_uri = f.readline()

'''
Break array into batches and perform inference.
'''
def web_service_predict(data, web_service_uri):
    data_reshaped = reshape_input(data)
    samples = data_reshaped.shape[0]
    data_arr = data_reshaped.reshape(int(samples/batch_size),
batch_size, n_timesteps, n_features)
    response_arr = np.array([send_request(data, web_service_uri) for
data in data_arr])
    reconstructed_data = response_arr.reshape(data.shape)
    return reconstructed_data

'''
Send a request to infer on one batch of data.
'''
def send_request(data, web_service_uri):
    data_json = json.dumps({"data": data.tolist()})
    headers = {'Content-Type':'application/json'}
    response_json = requests.post(web_service_uri, data_json, head-
ers=headers)
    return json.loads(response_json.text)['result']

'''
Calculate the reconstruction error
'''
def score_output(reconstructed_data, input_data):
    y = np.array(reconstructed_data)
    X = np.array(input_data)
    rec_err = np.abs(X - y)
    mean_err = np.mean(rec_err)
    return mean_err
```

```
predictions = []
for index in np.arange(0, len(validation_data), 10):
    data = validation_data[index]
    time = timestamps[index]
    reconstruction_error = score_output(web_service_predict(data, 
web_service_uri), data)
    predictions.append(reconstruction_error)
    print("time: {}, reconstruction error: {:.4f}".format(time, recon-
struction_error))
```