Anomaly Detection in the Elasticsearch Service

August 2019

AUTHOR:
Jennifer Andersson, BSc in Physics
Uppsala University, Sweden

SUPERVISOR:
Dr. Ulrich Schwickerath

OPENLAB SUMMER STUDENT PROJECT 2019

ABSTRACT

The Elasticsearch Service is a distributed search and analytics engine widely used across CERN. Currently, issues in the service are resolved manually after being detected through internal monitoring by service managers. However, the number of clusters and metrics are large which makes them difficult to track, and issues are often discovered and reported by users. This is time consuming and disturbs the workflow of the service users. In light of this, the main objective of this project is to develop a model capable of identifying anomalies in the Elasticsearch Service clusters, in order to predict and eliminate service issues before they cause problems. This is done by analyzing the history of cluster data using machine learning methods. In this way, a single metric signaling service issues can be obtained and used to alarm service managers of upcoming issues. In 2017, a deep neural network model was developed for this purpose. However, several issues were identified with the model, the most severe being convergence issues in the autoencoder. In this project, a revised autoencoder based on long short-term memory neural networks (LSTM's) is developed, tuned and evaluated. Finally, it is used on new Elasticsearch Service cluster data. The final model shows improved convergence compared to the previous model, and is able to detect real service issues based on the anomaly scores obtained. By combining the anomaly scores with those obtained by a model simply predicting the cluster state as a moving average of preceding states, the rate of false positives is reduced. The conclusion is that that a combined model, reporting anomalies based on a combination of the anomaly scores obtained by the LSTM based model and the moving average model, is the most sensitive to real service issues.

ACKNOWLEDGEMENT

Foremost, I want to express my sincere gratitude to my supervisor, Ulrich Schwickerath, for his constant support and encouragement during the course of this project. His motivation and enthusiasm have been contagious, and his guidance invaluable in helping me advance my knowledge and progress with the project. I am very grateful to have been given the opportunity to work on this project under his supervision; it has been a both challenging and rewarding experience that I will take with me on future endeavors.

I would also like to thank Pablo Saiz and Paulo Alexandre Canilho for answering all my questions and bringing valuable feedback and insights to the project.

Furthermore, my completion of this project would not have been possible without the CERN Openlab summer student team, to which I offer a special thanks for giving me this opportunity and for organizing the summer student program in a way that leaves nothing more to ask. I also want to thank all my fellow CERN Openlab summer students for being a part of making this summer a wonderful and unforgettable experience.

TABLE OF CONTENTS

1. INTRODUCTION

The number of machine learning applications has exploded in recent years, with successful examples emerging in a variety of domains. One important application area is anomaly detection, a branch of machine learning aiming to detect data samples significantly deviating from the majority of data samples with common characteristics. Examples of anomaly detection involves identification of medical problems and fault or error detection in different contexts. Another example is the detection of anomalous user behavior, for example extracted from cluster data. In this project we explore the possibility of predictive anomaly detection in cluster data extracted from the CERN Elasticsearch Service.

The Elasticsearch Service is a distributed search and analytics engine for IT-purposes, widely used in over 160 use cases across CERN. Currently, issues in the service, typically cluster degradation due to anomalous cluster usage, are often reported by other users before being manually resolved. Due to the large number of clusters and metrics, it is difficult for service managers to keep track of everything and detect all issues before they cause problems for the users. The procedure is time consuming and disturbs the workflow of the service users. To amend this, anomaly detection prior to cluster degradation, with alarming to the service managers, may prove useful. The idea is to detect service issues before they cause problems, by analyzing the history of cluster data generated by the Elasticsearch Service. In this way, a single metric signaling service issues can be obtained.

The main objective of this project is to develop a model capable of identifying anomalies in the Elasticsearch Service cluster data using machine learning methods. This is done by replacing a previously developed autoencoder by an autoencoder based on long short-term memory neural networks. This is a revised approach extending a previous project that began in 2017, involving the development of a semi-supervised model consisting of a deep autoencoder and a classifier. The model was designed to detect anomalous cluster behavior and perform post-classification of anomalous cluster states. Unfortunately, the model showed alarming issues such as convergence issues, preventing generation of useful output. Here, a revised model of the deep autoencoder is developed using long short-term memory neural networks (LSTM's), to better account for the time dependencies in the cluster data and resolve the convergence issues. In this report, a discussion of the architecture of the new model is provided, along with a description of the previous model. The project involves data preprocessing, implementation, tuning and analysis of the LSTM based model as well as evaluation of the initial results. Finally, the model is used on online data from the Elasticsearch Service and attempts to detect real service issues based on the model predictions are made. The performance of the LSTM-based model is also compared to the performance of a non-machine learning model based on a simple moving average, and anomaly scores for several service clusters are displayed in real-time. In the end, the goal is to obtain a model which can generate a single metric, an anomaly score, signaling cluster issues before they occur. This metric can be used to alarm service managers about upcoming service issues.

2. ANOMALY DETECTION USING MACHINE LEARNING

a. ANOMALY DETECTION

Anomaly detection in machine learning can be considered a classification problem. A model is trained on training data, where an anomaly can be defined as data patterns deviating from the expected behavior. Other terms commonly used are outliers or novelties; all pointing to data samples deviating from the characteristics of what is referred to as normal data samples. To detect such data, many different techniques can be utilized, several of which are covered in an overview of anomaly detection research written by Chandola, Banerjee and Kumar (2009). The authors discuss several different approaches and application areas of anomaly detection, such as medical anomaly detection, cyber-intrusion detection and industrial damage detection. Early overviews of anomaly detection using neural networks were published in 2003 by Markou and Singh. They point out a few challenges with neural network-based methods compared to statistical methods, such as the computational cost of training and retraining the network. Anomaly detection using neural networks forms the basis of this project, and this section aims to provide an overview of the theoretical background to the neural networks required to understand the anomaly detection model implemented.

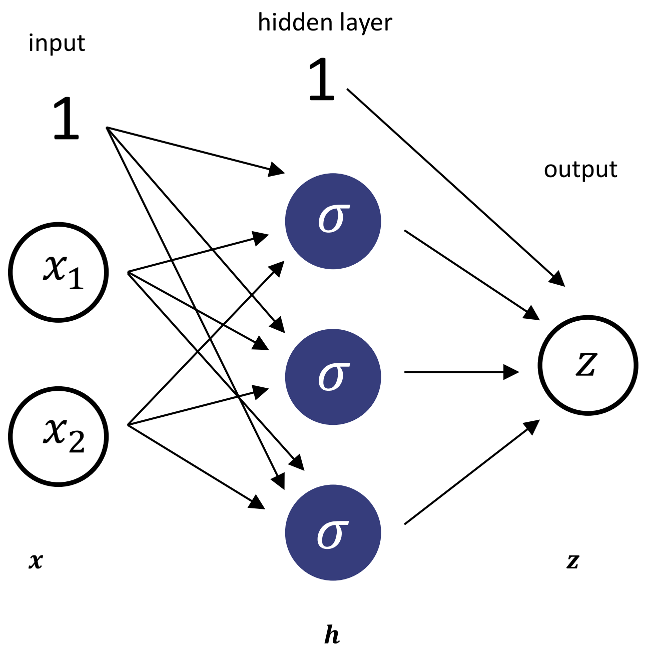
Anomaly detection using machine learning can be divided into three subclasses; unsupervised anomaly detection, supervised anomaly detection and semi-supervised anomaly detection (Chandola, 2009). They differ in what kind of training data they require.

- **Supervised:** Supervised anomaly detection requires labelled data sets including both normal and anomalous data samples. A model is then trained to classify new, previously unseen data. If the classes can be labelled accurately, which is a challenge in itself, this approach still raises the issue of unbalanced classes, as a common characteristic is that the number of anomalous data samples are small. This means that supervised machine learning is difficult in the context of anomaly detection, since anomalies often are rare by definition.
- **Semi-supervised:** Contrary to the labelled classes of supervised anomaly detection, semi-supervised anomaly detection only requires that the data samples of the normal class are labelled during training. This is very useful in cases where anomalies are rare, but also in cases where anomalies are hard to classify.
- **Unsupervised:** Unsupervised anomaly detection assumes anomalies to be rare, and that the data is dominated by normal samples. Unsupervised machine learning does not require labelled training data. Instead, the model is often trained to detect similar structures within the data, known as clustering.

When applying the trained model to new, unseen data, the anomalies detected are reported back to the user in some way. One option is to classify each data sample as either normal or anomalous according to the model output. A common way is to define an *anomaly score*. In this case, the anomaly score is calculated for each sample, so that the score of an individual sample determines the particular sample's likelihood of being an anomaly. For example, samples with scores above a certain threshold can be considered anomalies. This provides more flexibility, as the threshold can be adjusted depending on the use case and sensitivity to correct classifications, for example when false positives have to be avoided for safety-reasons. When neural networks are used for anomaly detection, the network is usually trained on normal training data in an unsupervised manner, learning the structure of the normal data to be able to recognize anomalies in the test data. In a one-class setting (where there is only one normal class), replicator neural networks can be used for anomaly detection. In this case, the number of output neurons is identical to the number of input neurons, and the normal input is reconstructed by the network. If it fails to reconstruct the input, the input is considered an anomaly. In this case, the anomaly score is defined by the reconstruction error.

There are several different anomaly detection techniques in machine learning. Some examples are classification-based techniques, others are nearest neighbour-based and clustering based techniques.

b. NEURAL NETWORKS



Intrinsically, a neural network is a nonlinear mathematical function that describes the output variable as a nonlinear function of the input variable, and can be used for both regression and classification problems. Figure 1 shows a schematic image of a simple feedforward neural network with one hidden layer. The equations governing a simple two-layer neural network can be found in Equation (1) and (2). Here, **X** is the input matrix, each row containing one input vector. **Z** is similarly the output matrix. **H** is a matrix containing the output of the hidden units, where each row is the output of the hidden layer for one input vector. σ is the activation function. In classification problems, the output matrix **Z** is pushed through a softmax transformation layer mapping the output to the class probabilities used for the classification.

$$\mathbf{H} = \sigma\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right) \quad (1)$$

$$\mathbf{Z} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} \quad (2)$$

Figure 1: Schematic image of a simple feedforward neural network.

During supervised training, the input and output are labelled data fed to the network. The weight matrix **W** of each layer is determined together with the bias vectors **b**. To train the network, a loss function is used to calculate the deviation between the output generated by the network and the true output. The parameters are then adjusted in order to minimize the loss through an optimization function such as stochastic gradient descent. Updating the weights, backpropagation is used to obtain the gradients of the loss function.

The two-layer neural network can be extended to a deep neural network with a large number of layers in a non-trivial way. The learning of such neural networks is referred to as deep learning. With each layer, however, the number of parameters that the network needs to learn increases. Since these models can become very flexible due to the number of parameters, regularization techniques are often required to avoid overfitting. Deep neural networks are also computationally expensive, and regular retraining should be avoided. An introduction to neural networks and deep learning can be found in any introductory text book (e.g. C.C. Aggarwal, 2018).

i. RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNN's) are designed to handle sequential inputs and can be viewed as a combined set of feedforward neural networks, where the output of each hidden state is fed back into the network. The advantage is that recurrent neural network can use information from previous decisions to make new decisions in the future. The power of this possibility is not difficult to realize, a simple example being text generation, where each new word will depend on the previous word generated, often with long time dependencies. A schematic image of an unrolled recurrent neural network layer can be found in Figure 2, where we view the recurrent neural network in terms of timesteps.

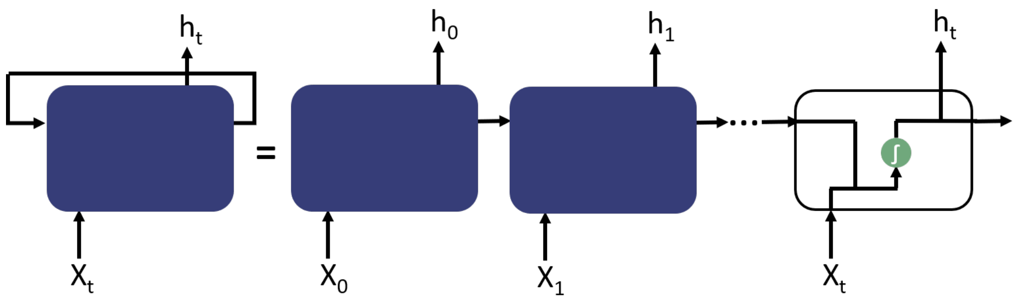


Figure 2: Unrolled RNN-layer. x_t is the input vector at timestep *t.* The output h_t of the hidden unit is fed back into the network resulting in a loop-structure. The activation function is marked in green, illustrating the simplicity of the architecture of each RNN-module.

The equations governing the recurrent neural network is similar to the equations governing the feedforward neural network. The equations of a network consisting of an input layer, a hidden layer and an output layer can be found in Equation (3) and (4), where $\boldsymbol{h}_t$ is the output of one hidden layer, σ_h is the hyperbolic tangent activation function and $\boldsymbol{x}_t$ is the input vector at time *t*. The weights $\boldsymbol{W}_c$, $\boldsymbol{W}_x$ and $\boldsymbol{W}_z$ are the parameters that need to be learned by the network. We have omitted the bias vectors in the formulation.

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{W}_c\boldsymbol{h}_{t-1} + \boldsymbol{W}_x\boldsymbol{x}_t) \qquad (3)$$

$$\boldsymbol{z}_t = softmax(\boldsymbol{W}_z\boldsymbol{h}_t) \qquad (4)$$

When training a recurrent neural network, the weights $\boldsymbol{W}_c$ of each hidden layer is shared for each timestep. As we shall see in the following section, this has a negative impact on the network's ability to learn. A simple and intuitive explanation of recurrent neural networks and long short-term memory neural networks can be found in the Colah's blog post *Understanding LSTM Networks.*

ii. THE VANISHING GRADIENT PROBLEM

The vanishing gradient problem occurs during training of deep feedforward neural networks when gradient information fails to propagate through the entire network. When the error gradient is propagated backwards through the network, the error signal typically decreases exponentially with each layer, to eventually vanish in early layers. This is referred to as the vanishing gradient problem. The reverse can also occur, where exponential growth of the gradient leads to exploding gradients. In recurrent neural networks, this problem becomes an even bigger issue since the network is unrolled for many timesteps. This can result in convergence issues as the vanishing gradient problem prevents the network from learning. Mathematically the problem presents itself by weight updates occurring to a smaller extent in early layers. Several techniques have been developed to tackle the vanishing gradient problem, such as alternate weight initialization schemes.

During gradient descent, when the loss is minimized with respect to the weights, backpropagation is the algorithm by which the gradients are determined. We will limit our discussion to the gradient with respect to the shared weights, as this is enough to illustrate the issue with vanishing gradients. The derivative of the loss $L(\boldsymbol{y}_t, \boldsymbol{z}_t)$ with respect to the shared weights is calculated as the sum of the corresponding derivatives of the loss at each timestep, as illustrated in Equation (5). $\boldsymbol{y}_t$ denotes the true output at timestep *t* and $\boldsymbol{z}_t$ is the output generated by the network. At timestep *k,* the derivative of the loss $L(\boldsymbol{y}_t, \boldsymbol{z}_t)$ with respect to the shared weights $\boldsymbol{W}_c$ can be determined as a sum-of-products according to Equation (6). Here, $\boldsymbol{h}_k$ denotes the output of the hidden state at timestep *k*, and $\boldsymbol{c}_k$ denotes the output that is fed back into the network. The second equality is obtained using the chain rule, due to the explicit and implicit dependence of $\boldsymbol{c}_k$ on $\boldsymbol{W}_c$, given Equation (3). The final term in (6) is the explicit derivative with respect to $\boldsymbol{W}_c$. The algorithm is referred to as backpropagation through time (BPTT).

$$\frac{\partial L(\boldsymbol{y}_t, \boldsymbol{z}_t)}{\partial \boldsymbol{W}_c} = \sum_t \frac{\partial L_t}{\partial \boldsymbol{W}_c} \qquad (5)$$

$$\frac{\partial L_k}{\partial \boldsymbol{W}_c} = \sum_t \frac{\partial L_k}{\partial \boldsymbol{h}_k}\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{c}_k}\frac{\partial \boldsymbol{c}_k}{\partial \boldsymbol{c}_t}\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{W}_c} = \sum_{t=0}^{k} \frac{\partial L_k}{\partial \boldsymbol{h}_k}\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{c}_k}\left(\prod_{i=t+1}^{\boldsymbol{k}} \frac{\partial \boldsymbol{c}_j}{\partial \boldsymbol{c}_{j-1}}\right)\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{W}_c} \qquad (6)$$

From Equation (6) we see that the derivative $\frac{\partial \boldsymbol{c}_j}{\partial \boldsymbol{c}_{j-1}}$ in the Jacobean matrix $\frac{\partial \boldsymbol{c}_k}{\partial \boldsymbol{c}_t}$ may lead to vanishing gradients, since the derivative of the activation function in (4) is less than unity. Thus, $\prod_{i=t+1}^{\boldsymbol{k}} \frac{\partial \boldsymbol{c}_j}{\partial \boldsymbol{c}_{j-1}} \to 0$ for large *k.* Here, the error at timestep *k* is transported back to previous timesteps. In this case, the value of the gradients of the loss decreases exponentially with *k-t*. The problem intensifies as the number of time steps increases, which is why recurrent neural networks are said to have a short-term memory, incapable of identifying long-term dependency correlations. The issue with exploding gradients occurs for similar reasons, when repeated multiplication of large $\frac{\partial \boldsymbol{c}_j}{\partial \boldsymbol{c}_{j-1}}$ leads to the gradient blowing up. Thorough mathematical proof is for example provided by Pascanu, Mikolov and Bengio (2013).

iii. LONG SHORT-TERM MEMORY NEURAL NETWORKS

In the late 1990's, a version of recurrent neural networks overcoming the vanishing gradient problem was proposed by Sepp Hochreiter and Jürgen Schmidhuber (1997). Long-short term neural networks, LSTM's, have since become widely popular, accounting for most of the great achievements accomplished by the use of recurrent neural networks. Like regular recurrent neural networks, LSTM's are designed to learn from sequenced data such as time series, but with an architecture that combats the vanishing gradient problem.

In the initial version, the vanishing gradient problem is tackled by attacking the decaying error backflow using constant error carousel units. With multiplicative gates learning to open and close access to the error flow, the units enforce constant error backflow.

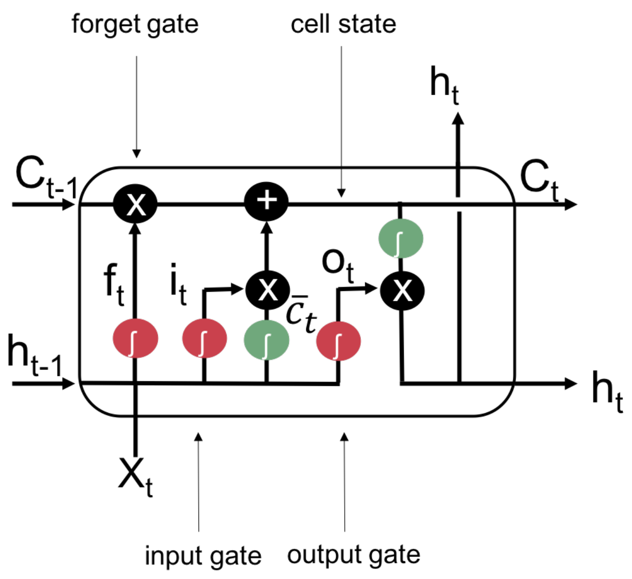


Figure 3: Illustration of the LSTM-architecture.

The equations governing the LSTM network are found in Equation (7) - (12). Notations are given in Figure 3, where green circles illustrate the hyperbolic tangent activation function σ_h and red circles illustrate the sigmoid activation function σ_s.

$$f_t = \sigma_s\,(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad (7)$$

$$i_t = \sigma_s(W_i \cdot [h_{t-1}, x_t] + b_i) \qquad (8)$$

$$\bar{C}_t = \sigma_h(W_C \cdot [h_{t-1}, x_t] + b_C) \qquad (9)$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \bar{C}_t \qquad (10)$$

$$o_t = \sigma_s(W_o \cdot [h_{t-1}, x_t] + b_o) \qquad (11)$$

$$h_t = o_t \cdot \sigma_h\,(C_t) \qquad (12)$$

The module replaces the simple RNN-module of Figure 2. Like before, the input vector is denoted by x_t and the output of the hidden state at time *t* is denoted by h_t. Each LSTM-unit is composed of a cell state, a forget gate, an input gate and an output gate. The gates provide four neural network layers, together regulating the information flow through the network. As shown by Equation (7), (8) and (11), the three gates consist of a familiar neural net layer with a sigmoid activation function, followed by a pointwise multiplication operation. The cell state C_t contains the information remembered by the network at time *t.* At the forget gate, how much of the information should be passed through to the cell state is determined, i.e. what should be forgotten. The output of the gate is denoted f_t (7), and what should be forgotten is detemined by pointwise multiplication with the previous cell state. Next, the input gate determines what information should be added to the cell state. A hyperbolic tangent layer creates new cell state candidates $\bar{C}_t$ (9), which is combined with the output of the input gate i_t (8) and added to the cell state by pointwise addition. The new cell state (10) is pushed through a hyperbolic tangent layer to ensure that its values remain between -1 and 1. Finally, the output gate determines what to output, o_t (11), and the output of the hidden state h_t is determined by elementwise multiplication (12).

The forget gate contribution to the cell state (10) is key to understanding how the LSTM network solves the problem with the vanishing gradients. Plugging the cell state into (6), we realize that the forget gate contribution to the gradient results in non-vanishing gradients if the forget gate activations are greater than 0. As is evident from Equation (7) – (12), the number of parameters the network is required to learn has grown significantly compared to the regular recurrent neural network described in (3) and (4), leading to increased computational expense. In return, the vanishing gradient problem is solved, and the network is able to learn long-term correlations otherwise not possible.

Since the introduction of the LSTM recurrent neural network in 1997, several derivations of the LSTM network have been presented. An important extension was made two years later by Gers, Schmidhuber and Cummins (1999) with the inclusion of the forget gate, enabling the LSTM module to reset its state. Later extensions include for example the gate recurrent unit. This network simplifies the standard LSTM architecture by combining the hidden state vector with the cell state vector, and the input gate with the forget gate, forming a single update gate.

3. ANOMALY DETECTION AND DEGRADATION PREDICTION

a. OBJECTIVE

Issues frequently occur in the Elasticsearch Service, sometimes causing cluster degradation. Here, the Elasticsearch Service data is used to train and evaluate an LSTM-based neural network model to detect anomalous cluster behavior in order to prevent such cluster degradations, based on the idea that the history of data signals cluster breakdowns before they occur. At each time the clusters can be classified as one of three states:

- **Green:** Normal state; the cluster is fully operational.
- **Yellow:** Warning state; the service has allocated all primary shards, but some of the replicas could not be allocated.
- **Red:** Degradation state; some or all of the primary shards are not allocated and there is a cluster degradation.

The main objective of the project is to look at the history of the Elasticsearch Service cluster statistics and use this information to predict the present (and ideally future) cluster state, to be able to tell if a cluster is going to break down, i.e. to obtain a single metric that is able to predict future red or yellow states. This can be used to alarm service managers about probable service issues while the cluster is still green, so that problems can be addressed before the user is affected. To do this, we train artificial neural network models to find anomalies, using the available input metrics. In this way, we move from the current approach of solving issues after they are reported, to a proactive approach where we take action before the cluster breaks down.

b. THE ELASTICSEARCH SERVICE DATA

Currently, the Elasticsearch Service contains 30 clusters and offers over 160 use cases all over CERN. The clusters are node based with indices which are split into shards. The shards can be either primary or replicas. Each shard is a lucent inverted index and each node is a server running Elasticsearch. In short, the user adds documents and can search them by the index.

A long record of cluster data spanning two years, with samples every five minutes, is available to train and evaluate the model. The statistics include:

- **Cluster statistics:** e.g. disk and CPU usage
- **Node statistics:** e.g. max RAM and CPU usage
- **Apache statistics:** e.g. number of apache return codes and response time
- **Thread pool:** e.g. percentage of threads completed
- **Log information:** e.g. added nodes
- **Preprocessed data** e.g. averages, deltas

In total, the available data amounts to approximately 180 features, out of which 121 input parameters are selected. These are obtained at 207 360 timesteps amounting to two years of data for the longest living clusters. However, the exact amount of available data varies by the cluster.

c. SEMI-SUPERVISED MODEL

i. MODEL DESCRIPTION

Prior to the development of the current model, an initial feedforward neural network model was developed. The combined model consisted of two combined networks, as illustrated by the schematic image presented in Figure 4. The first part is a fully connected autoencoder, which is an unsupervised neural network that learns to predict anomalies, i.e. to filter anomalous (red or yellow state) candidates. It learns to abstract and reconstruct normal inputs by learning an encoding of the data. The autoencoder is trained on data with green cluster states (normal data) and performs a preselection of potentially abnormal states, prior to classification.

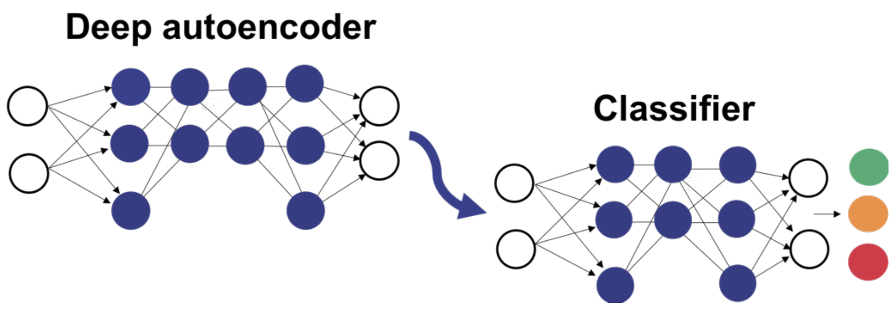


Figure 4: Schematic illustration of the original semi-supervised model.

If the root mean square error between the original inputs and the predicted, reconstructed inputs is below a certain threshold, the cluster status is classified as green. If it is above the threshold, it is fed into the degradation prediction classification network. This network is fully supervised, trained on anomaly inputs, i.e. data with future yellow and red states filtered by the autoencoder. Here, the final classification is done.

ii. CHALLENGES WITH THE SEMI-SUPERVISED MODEL

The implementation of the previous model showed that there are some major challenges with the approach described in the previous subsection. These include a high rate of false positives, i.e. where the model wrongly classifies the current cluster state as red or yellow. This is a major issue as a 95% prediction accuracy means that a cluster is wrongly classified as being degraded every five minutes. The model evaluation also showed convergence issues in both networks. Due to a lack of statistics, this network was mainly trained on yellow state candidates, but the rate of false positives remained at a high level.

Another conclusion regards the amount of data and consequently high dimensionality of the problem. The performance could possibly improve with increased understanding of the data characteristics, rigorous preprocessing of the data and improved feature selection. Another issue identified is that cluster characteristics evolve over time, and this behavior was detected as anomalies by the model. Consequently, the model had to be frequently retrained to adapt to changes over time. Since the autoencoder required a very deep network, the convergence issues due to the vanishing gradient problem became substantial. This made retraining even more difficult due to the necessity of constant retuning of parameters.

Moreover, it proved very difficult to generalize the model to more than one cluster per model, since the clusters show inherently different behavior. Therefore, one model per cluster was trained and deployed, increasing computational cost as several models need to be trained and retrained. Periodic anomalous cluster signatures were also falsely treated as anomalies by the semi-supervised model. These occur during normal circumstances, e.g. when new indices are repeatedly created at midnight, which the model should be able to learn.

d. REVISED MODEL

The basic idea of the new approach is to use LSTM's to improve model performance and address the convergence issues, which are diagnosed to be caused by the vanishing gradient problem. This allows for more frequent updates to follow the time evolution and enables the network to learn long-term time dependencies, which is crucial to time series forecasting. The model is implemented in Python using the neural-network library Keras on top of TensorFlow. Initially, only the autoencoder is revised, but in the future an improved classifier could be integrated with the autoencoder replacement to categorize the abnormal states. The main issue with the integration is finding an optimal way of classifying the cluster states, since the training of the classifier is supervised and requires annotated data during training.

The revised autoencoder learns to reconstruct the inputs and uses the samples from the previous *t* timesteps to predict the output at timestep *t+1*. Thus, it uses historic Elasticsearch Service data to predict the present.

i. TRAINING AND VALIDATION DATA

The data is divided into training and validation sets using a generator. From the total data, x chunks are used as training data and y chunks are used as validation data, where x and y are specified prior to network training. Chunking of the data means that training and validation data are taken in chunks distributed over the entire data set. In each chunk, sets of *t* samples are used to predict following sample. To optimize the data usage, the data sets of *t+1* samples used for training and validation is moving within the chunk, such that each chunk generates as many sets of *t+1* unique samples as possible, including overlapping sets. The number of unique sets are thus approximately equal to the difference between the chunk size and the number of timesteps *t*. This approach helps amend the problem of cluster time evolution detected in the previous model.

Figure 5 shows an example of the evolution of one input parameter over the course of two years, visualizing the importance of retraining if the network is unable to learn the natural evolution. The generator's division of data ensures that the network is trained and evaluated on data from the entire time period available.

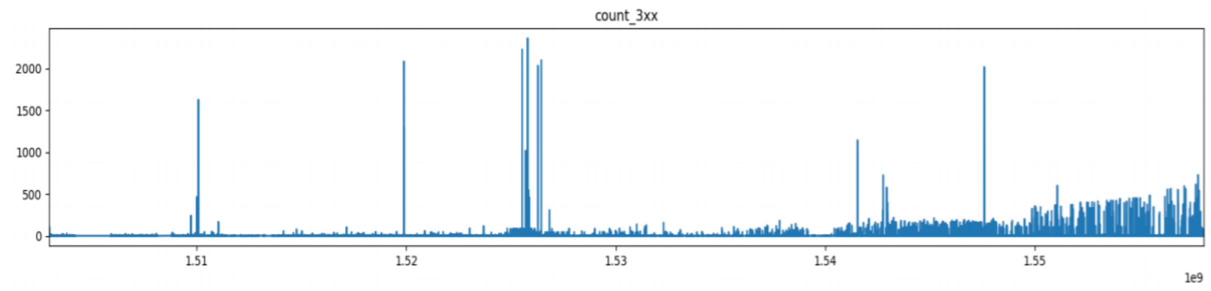


Figure 5: The time evolution of one input parameter (number of apache 300 return codes) over two years. The number of return codes is displayed on the y-axis. The x-axis shows the time, amounting to two years.

ii. DATA PREPROCESSING

Due to the high dimensionality of the problem, data preprocessing can have a significant impact on the model performance since it affects the network's ability to find correlations within the data. The chain of data preprocessing used includes removal of outliers through the z-score algorithm (optional, necessary when normalizing using maximum values), normalization through centralization, and taking the logarithm of the inputs depending on the skewness of the distribution to make the input parameter distributions more gaussian. If the skewness of the distribution of one input parameter is above a certain threshold, the input parameter is replaced by the logarithm of its value to reduce tails present in the histogram and increase the skewness of the distribution. Another option investigated is to use standardization instead of centralization. Standardization is less sensitive to outliers, which means that the time-consuming z-score outlier detection can be omitted. This is preferred, since the outlier detection removes a lot of training data. Different preprocessing schemes are investigated through analysis of the model performance, by training on differently preprocessed data. At the end of the preprocessing chain, the data is pushed through a (modified) sigmoid- or hyperbolic tangent function. Figure 6 and 7 shows an example of a raw input data parameter and a preprocessed input data parameter. Another attempt is to use an additional input in terms of the minute of the week to aid the network in detecting periodically recurrent anomalies in the input data.

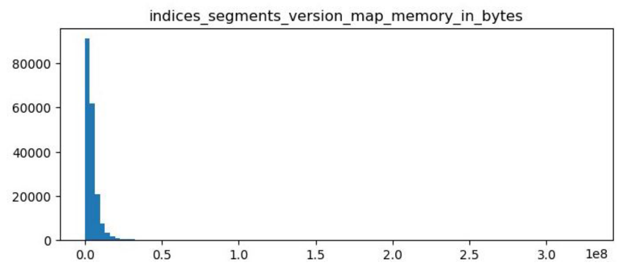


Figure 6: Histogram of the raw training data in terms of one input parameter.

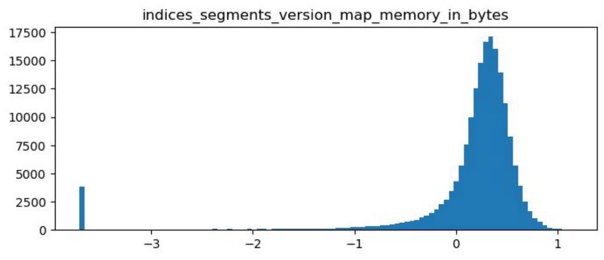


Figure 7: Histogram of the preprocessed training data in terms of one input parameter. Notice the remaining outlier.

4. MODEL EVALUATION

a. MODEL PERFORMANCE

The final network consists of five LSTM layers. The initial prototype involved 100 586 772 parameters. By addressing a configuration bug, this was reduced to a total number of 590 480 trainable parameters, 118 096 in each LSTM-layer. Initial evaluation of the model shows a significantly improved convergence compared to the previous model. Table 1 and 2 show the training and validation loss for the ALICE and CERT clusters, using different skewness thresholds for replacing the input by its logarithm. 288 timesteps are used to predict the state at the current timestep, i.e. a 24-hour history of data from the Elasticsearch Service is used to predict the present state. The SGD-optimizer is used, and the loss function is a simple mean squared error. The default decay- and learning rates are 0.000004 and 0.7, respectively. One third of the data is used for validation, and the remaining data is used for training. The evolution of the training and validation loss during 100 epochs of training can be found in Figure 8 and 9. In this case, no removal of outliers is carried out, standardization is used for normalization and the modified sigmoid function is applied to the inputs before they are fed into the network, to ensure that their values are in the proper range.

Cluster: ALICE			
Logging Threshold	**Validation loss**	**Training loss**	**Delta**
1000	0.0014	0.0016	-0.0002
15	0.0031	0.0026	0.0005
10	0.0026	0.0023	0.0003
7	0.0052	0.0024	0.0028
3	0.0035	0.0023	0.0012

Table 1: Training and validation losses after for different logging thresholds on a model trained on data from the ALICE-cluster.

Cluster: CERT			
Logging Threshold	**Validation loss**	**Training loss**	**Delta**
1000	0.0056	0.0019	0.0019
15	0.0081	0.0039	0.0039
10	0.0048	0.0005	0.0005
7	0.0099	0.0054	0.0054
3	0.0070	0.0026	0.0026

Table 2: Training and validation losses for different logging thresholds on a model trained on data from the CERT-cluster.

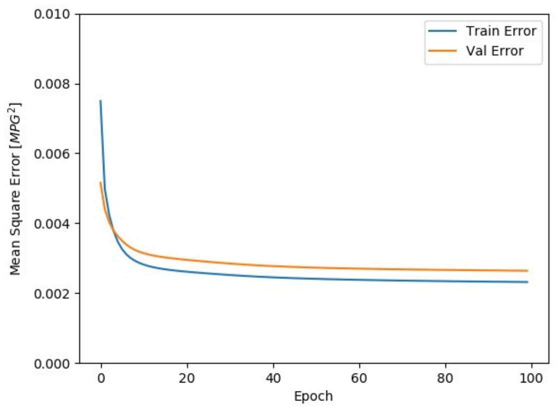


Figure 8: Training and validation loss during 100 training epochs in the ALICE-cluster. Logging is applied for input parameters with skewness larger than 10.

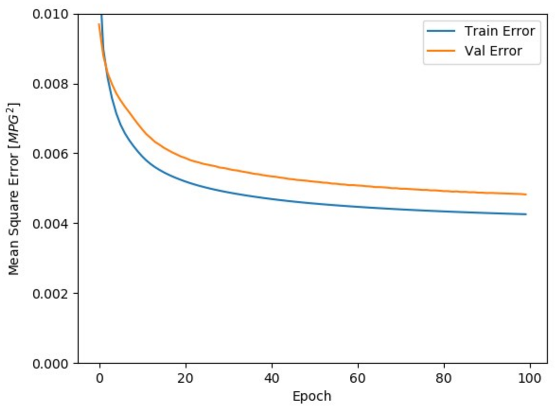


Figure 9: Training and validation loss during 100 training epochs in the CERT-cluster. Logging is applied for input parameters with skewness larger than 10.

As can be seen, the models trained for the ALICE- and CERT-clusters converge nicely and the training- and validation losses are low after 100 epochs. These are promising results compared to the performance of the previous model, which showed significant convergence issues. We also note that the model performance is cluster dependent. For the ALICE-cluster, the validation loss is the lowest using a logarithm threshold of 1000, where the original inputs are used. For the CERT-cluster the corresponding threshold is 10. A logarithm threshold of 0 means that the logarithm of the input data is used for all features. We notice that the optimal preprocessing chain may differ between clusters, which is reasonable since the clusters have proven to show inherently different behaviors. It does appear to follow a similar trend, and in both cases the thresholds 1000 and 10 yield the lowest rate of validation loss.

b. ANOMALY SCORE

To evaluate the model, a simple anomaly score based on the mean squared error between the reconstructed input generated by the network and the true output is calculated by running the model on previously unseen data, in this case the latest data from the Elasticsearch Service. A higher anomaly score indicates a higher probability that an anomaly has been detected, which in turn indicates an increased likelihood of service issues. The anomaly scores based on the network prediction are compared to the anomaly scores based on a prediction equal to a simple moving average, where the prediction at each timestep *t* is simply the average of the previous 288 timesteps. The identical preprocessing chain is used. This is a simple and computationally cheap non-machine learning approach which does not require any training data.

c. PERFORMANCE ON ONLINE DATA

Trained models are used to make predictions on online data to enable real-time anomaly detection. Once a day a one batch training step is carried out for each cluster, whereupon the weights are updated. The anomaly scores using the network prediction as well as the moving average prediction can be found in the Kibana dashboard presented in Figure 10, using a logarithm threshold of 10. This is an example of a dashboard presenting anomaly scores during the course of a few hours for a selection of clusters. Figure 11 shows a separate example of a visualization of the LSTM anomaly scores, and Figure 12 shows the corresponding moving average anomaly scores. Figure 13 shows the corresponding visualization of the anomaly score difference between the two models. The anomaly score ratio and average are also displayed, along with histograms showing the anomaly score distribution of each model. Example graphs are omitted here but can be provided in real time for selected clusters. A pie chart comparing the performance of the two models is also brought forth.

The dashboards presented show the evolution of the anomaly scores over time. An increase in the anomaly score indicates an anomaly. This can occasionally be observed in the anomaly score visualizations based on predictions generated by both the LSTM network and the simple moving average model. As we can see, the behaviors of the LSTM anomaly scores and the moving average anomaly scores are very similar, but not identical, over time. The network has learnt that the cluster behavior at a certain timestep is similar to an average over the preceding timesteps. In general, however, we note that the LSTM anomaly scores are higher than the moving average anomaly scores. This can be partly due to the fact that time stamps are incorrectly predicted by the LSTM network, but further investigation is required to establish the cause of the differences in behavior of the anomaly scores in detail.

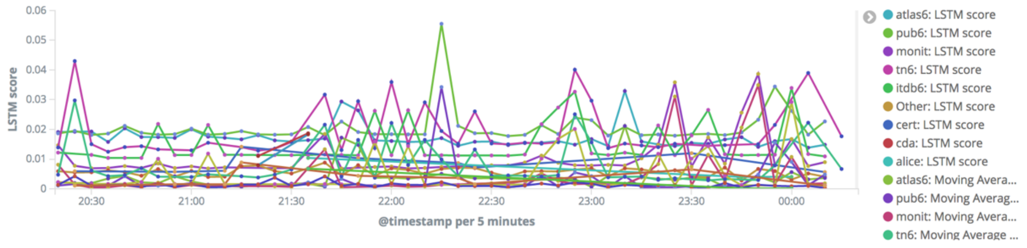


Figure 10: The LSTM and moving average anomaly scores for several different clusters.

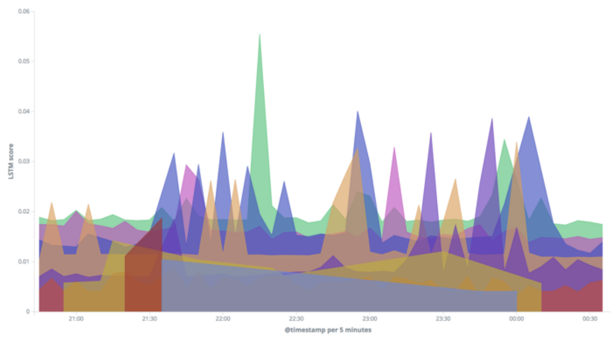


Figure 11: The LSTM anomaly scores for several different clusters. Different clusters are illustrated in different colors.

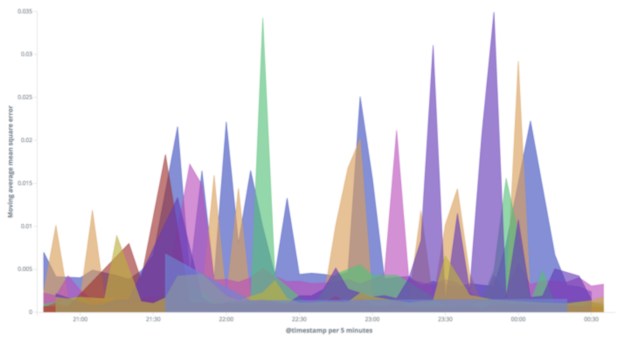


Figure 12: The moving average anomaly scores for several different clusters. Different clusters are illustrated in different colors.

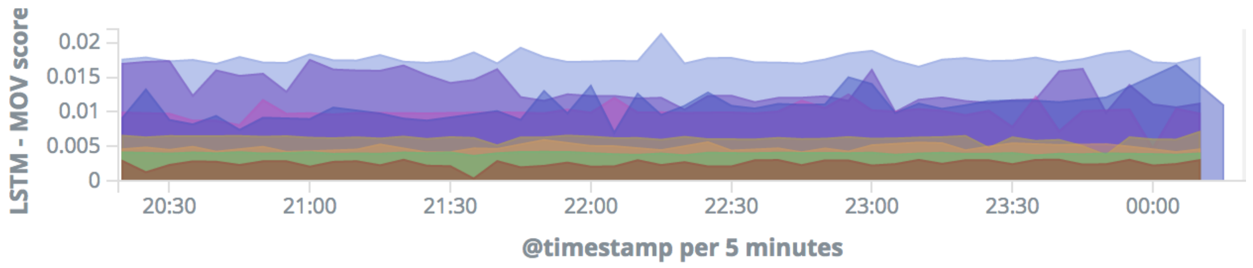


Figure 13: The difference between the LSTM and the moving average anomaly scores for several different clusters. Different clusters are illustrated in different colors.

Histograms over the distribution of the anomaly scores show that the detected anomaly count (instances with anomaly scores above a threshold) is lower during a fixed time period using the LSTM-based model. This could suggest a lower false positive rate and that the LSTM-based model can filter anomalies causing real issues better than the moving average model. This does, however, require confirmation by analysis of the events generating the high anomaly scores, since no classification of anomalies is available. It is difficult to map anomalies to service issues, and integration of an updated classifier with the replacement encoder would be beneficial.

Figure 13 shows the difference between the anomaly scores based on the LSTM predictions and the moving average predictions. If this artificial anomaly score is large where both models show peaks in the anomaly scores, i.e. where both models identify an anomaly, this seems to be a good indicator of real service issues as deduced by manual analysis of the detected anomalies. The reason behind this is subject to further investigations, but it indicates that the LSTM network is able to extract real anomalies better than the moving average model and that LSTM anomaly scores corresponding to real issues vary in size. In other words; if the anomaly score threshold is set to detect one real issue for each model, the LSTM-approach yields a lower rate of false positives than the moving average model. A large difference between the moving average and LSTM anomaly scores in some cases also appears to be a sign of overtraining, notably where the available training data is small. Another possible metric is to use the ratio between, or average of, the two anomaly scores. By analyzing the underlying events, it is concluded that a combination of the scores from the two models, using both the difference and the average, is the most efficient in identifying anomalies corresponding to events relating to real service issues. Further investigation can be carried out in order to find the optimal metric.

At the current deployment, a notification is sent if the difference between the LSTM and moving average anomaly scores exceeds 0.1. This threshold can be tuned to optimize the rate of false positives. Another notification is sent if the LSTM-anomaly score is four times the threshold. False positives have been noticed, but the reason for these instances have been identified as convergence issues due to a lack of training data in the particular clusters. Increasing the amount of training data by reducing the amount of validation data and retraining the network so far eliminates these kinds of false positives. Notification tuning is, however, still a work in progress. By looking at the anomaly scores produced, issues have been fixed before they cause problems, which is the main goal of the model deployment. One example is a stray data node with a /var partition filled up.

5. CONCLUSION

In conclusion, the convergence of the LSTM based model is good and the performance on previously unseen data from the Elasticsearch Service is promising. Thus, the convergence issue, which constituted the main issues with the previous model, has been solved using the LSTM-based approach. A better understanding of the input data is also accomplished, and work is finalized to run the model continuously on real-time incoming data. This includes upload and visualization of anomaly scores as well as deployment of an alarm system notifying service managers when an anomaly has been detected. The anomaly scores using the LSTM-based model for selected clusters are compared to the anomaly scores using a non-machine learning method, the simple moving average, and it is found that a combined prediction is the most sensitive to real service issues. Therefore, together with the moving average model, the LSTM-based model provides a single metric that can be used to predict anomalies in the Elasticsearch Service clusters before they cause problems. Analysis and tuning of the model parameters and notification thresholds are ongoing.

6. FUTURE WORK

To further improve performance, systematic tuning of the model hyper-parameters can be continued. Notification tuning based on the anomaly scores of the two models is also ongoing, and it would be beneficial to investigate cases where the moving average performs better than the LSTM-model, as well as other special cases. In general, continuation of the work involves gaining more experience of the model behavior and its performance on the current system. In terms of data preprocessing, its significant effect on the model performance has been shown and further systematic testing could be performed to find the optimal preprocessing chain. It is also possible to improve the model by feeding the network output back into the network in order to enable time series forecasting. In this way, future cluster states could be predicted.

Investigating the possibility of training a single model for all clusters is also of interest. A successful general model would greatly simplify the Elasticsearch Service anomaly detection. A next step is also to investigate the possibility of implementing a revised network for anomaly classification, to be integrated with the revised autoencoder. This does, however, require annotated data which demands additional work, since a strategy for this is currently not developed. Today, there is no immediately optimal way of classifying the anomalies and this is done manually.

Moreover, the model performance could be compared to other approaches in addition to the moving average. One example is the isolation forest; a tree ensemble method where outliers are explicitly detected, as opposed to the current approach which is based on the profiling of normal data. The LSTM-model approach could also be extended to other contexts, such as other services, to evaluate its general performance.

7. REFERENCES

Aggarwal, C. C. *Neural Networks and Deep Learning*. Springer International Publishing AG. 2018.

Chandola, V., Banerjee, A., & Kumar, V. Anomaly Detection: A survey. ACM Computing Surveys. Vol 41, no 3, 2009. doi: 10.1145/1541880.1541882

Gers, F.A., Schmidhuber, J. & Cummins, F. Learning to Forget: Continual Prediction with LSTM. *ICANN'99 International Conference on Artificial Neural networks*. Vol. 2, 1999: 850-855.

Hochreiter, S. & Schmidhuber, J.. Long Short-Term Memory. *Neural Computation.* Vol. 9, no. 8, 1997: 1735-1780.

Markou, M. & Singh, S. Novelty detection: A review – Part 2: Neural network based approaches. *Signal Processing*. Vol 83, no. 12, 2003: 2499-2521. Doi: 10.1016/j.sigpro.2003.07.019

Pascanu, R., Mikolov, T. & Bengio, Y. On the difficulty of training recurrent neural networks. *ICML'13 Proceedings of the 30th International Conference on International Conference on Machine Learning.* Vol. 28, 2013: III-1310-III-1318.