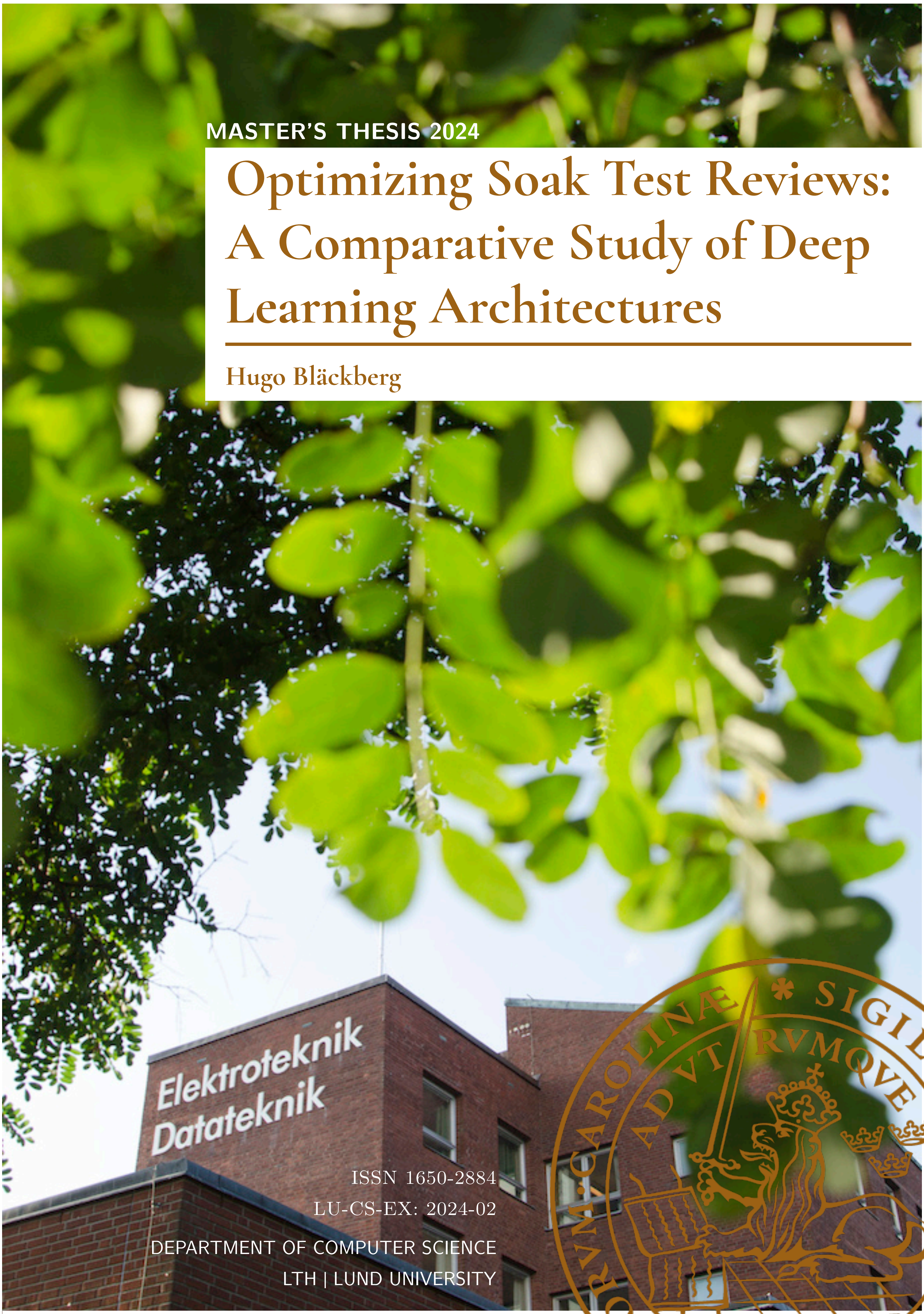
MASTER'S THESIS 2024

Optimizing Soak Test Reviews: A Comparative Study of Deep Learning Architectures

Hugo Bläckberg



ISSN 1650-2884
LU-CS-EX: 2024-02

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-02

Optimizing Soak Test Reviews: A Comparative Study of Deep Learning Architectures

Optimering av Soak Test-granskning: en jämförande studie av djupinlärningsarkitekturer

Hugo Bläckberg

Optimizing Soak Test Reviews: A Comparative Study of Deep Learning Architectures

Hugo Bläckberg
hu3266bl-s@student.lu.se

January 19, 2024

Master's thesis work carried out at Neo4j Sweden AB.

Supervisors: Patrik Edén, patrik.eden@cec.lu.se
Gustav Lindroth, gustav.lindroth@neo4j.com
Eric Sporre, eric.sporre@neo4j.com

Examiner: Markus Borg, markus.borg@cs.lth.se

Abstract

Soak testing, a subset of system testing, aims to assess the long-term health of a system by running for an extended duration, such as several days. Soak testing aims to expose any performance degradations that occur over a longer time than other forms of system testing would. The manual review of results from a soak test makes it a time-consuming process. Investigating whether machine learning can be applied to identify anomalies in the results could reduce the time spent on review. We explored different deep learning architectures for this purpose, including long short-term memory autoencoders, transformers, and convolutional neural networks. The long short-term memory autoencoder and transformer classify anomalies based on reconstruction loss and a threshold value. In comparison, the convolutional neural network and transformer-encoder were trained with target labels. All models performed subpar, with low accuracies, except the convolutional neural network with a balanced accuracy of 95% on the test set. Consequently, we propose a 1d convolutional neural network model that, with a high degree of accuracy, can classify a sub-sequence concerning a soak test as non-anomalous or anomalous.

Keywords: Machine learning, Anomaly detection, LSTM, Transformer, CNN, Soak testing, Graph Database

Acknowledgements

First and foremost I would like to thank my supervisor at LTH, Patrik Edén, for all the challenging and rewarding discussions. My work has greatly benefited from them, as much as I have enjoyed them. I would also like to extend my gratitude to Markus Borg for taking on the role of examiner.

Thank you to everyone at Neo4j for giving me the opportunity to write my thesis, especially Eric Sporre and Gustav Lindroth for welcoming me to the team and helping me tackle difficult topics related to my work.

Lastly I would like to thank my partner, Kathinka, for keeping me sane, and my parents, tasked with proofreading a topic totally foreign to them.

Contents

Chapter 1

Introduction

1.1 Background

System testing is an integral part of the software development life-cycle, used to understand the workings of the system as a whole. It serves as an evaluation to ensure that the system meets specified requirements and functions as expected. This is in contrast to testing that targets smaller individual areas, such as unit or functional testing. System testing is used at the end of the life-cycle to solidify a release after all other tests have concluded [1].

Neo4j is a graph database and software as a service provider that use system testing in their release process to catch problems with their products and services that only become apparent in an all-encompassing setting. One focus of system testing is on their graph database named after the company. This is to ensure scalability, that it performs without problems when the volume of data and user interactions increase. A subset of system testing that Neo4j utilize for their database is soak testing. Soak testing intends to target the long-term health of a system by running for a longer duration, such as several days. The soak tests aim to expose any performance degradations that happen over longer time than what other forms of system testing would. These degradations include memory leaks, increased garbage collection or any other problems in the system [2][3].

When a soak test is running the database outputs metrics for different events and processes related to how it is operating that are useful for debugging purposes. These processes and events include hardware related ones such as memory or cpu usage but also ones such as the total number of queries at a given moment. The verification process for these system tests is currently done manually. This involves looking at the metrics produced by the database to ensure that the database performed as expected throughout the test run. The verification process requires that the inspector has an understanding of all the processes that occur in the database. The inherit nature of system testing entails that the requirements are fuzzier compared to for example unit testing that give undisputed results. This added nuance to system testing of the database requires the ability to weigh different results in contrast to others, ultimately with respect to the performance of the system as a whole. This is something that human brains are very good at and where available algorithms fall short.

With the rise of machine learning and recent advances in neural networks it seems that we are improving on creating models that imitate the human brain. An effort is therefore justified to explore how well machine learning, specifically deep learning architectures, can meet the requirements concerning verification in system testing. The effort to automate this area would bring benefits such as relieving developers from extensive manual verification and could also increase the quality of verification as seen in areas such as medical imaging where machine learning algorithms performs at least as well as humans [4].

1.2 Research Objective

The objective of this thesis is to investigate how machine learning can be utilized to facilitate the verification process of the results from a soak test. This includes outright classifying a test as having passed or failed or receive feedback regarding behaviour that is anomalous. The effort was conducted through collection of metrics data from soak tests, identifying metrics relevant during a manual inspection in regards to classification as well as exploring, evaluating and comparing different deep learning architectures for the purpose previously stated. Evaluation of said architectures was conducted through measures such as sensitivity, specificity and accuracy related to classifying sub-sequences of soak tests as anomalous or part of a failing test. There exist multiple variations of soak tests at Neo4j that simulate different environments the database face in production. We limit our research by only investigating one of these: *administrative soak tests*.

1.2.1 Research Questions

With our research objective as a basis we formulated the following research questions:

RQ1: How well does the amount of anomalies detected in unsupervised analyses correlate with the need for further manual scrutiny?

RQ2: Do the anomalies detected help identify where in the time series that suspicious behaviours occured?

The research question above also extends into a further reaching one, namely:

RQ3: How accurately can a model classify if the results from a soak test constitute a pass or a fail based on the available data?

1.3 Scientific contribution

This thesis seeks to expand the knowledge for anomaly detection on operational data generated from system testing. Specifically it evaluates different state-of-the-art approaches to deep learning, that are prevalent in other areas, on operational data from a graph database. We propose a model utilizing a 1D convolutional neural network to, with a high degree of accuracy, classify system tests of a graph database on the basis of a pass or fail.

1.4 Related work

A previously conducted master's thesis at Neo4j [5] touched the same topics in identifying anomalies in time series data relating to the database in an operational state. The authors focused on trying to predict that the database was at risk of fault intolerance before it happened. They investigated if Histogram-based Outlier Score (HBOS) and an LSTM Autoencoder could be used for their purposes and compared the two. Although the authors were unsuccessful in their effort they lay out improvements to the data gathering that this thesis has from the start. The authors gathered data not from a controlled environment but from a real-life production environment. They state that future work should rather gather data in a controlled environment that is reproducible and therefore balanced. The data gathered within our project, i.e. each soak test run, posses the property that the environment that it is created within as well as the soak test itself are identical each run.

Another paper that employed the LSTM Autoencoder on time series approach is [6]. The authors achieved a high accuracy on detecting anomalous time series. The paper therefore serves as a good foundation for us when considering how to proceed with items such as deciding on a threshold value for our reconstruction loss.

The authors in [7] conducted a survey of popular deep learning architectures commonly used within areas such as computer vision, natural language processing and machine translation for the purpose of classification. The authors make the case for why 1D Convolutional Neural Networks (CNN) can be applied to multivariate time series in a compelling way, which is described in subsection 3.3.3. Other attempts on using CNNs for multivariate time series include [8][9][10]. This leads us to apply this type of architecture in this thesis.

With the recent rise in research on Transformers and in essence Attention-based models the authors in [11] demonstrate how Transformer models can be applied to anomaly detection on multivariate time series. They lay out a technique that is similar to anomaly detection using autoencoders where the model reconstructs the sequence and an anomaly is detected by comparing the reconstruction error to a threshold decided upon. The training and validation data only contained non-anomalous sequences which entails in their opinion that the model learned the latent space of normal heartbeats. This is an interesting approach since one could argue that the training- and validation data could also include anomalous sequences and that the purpose of the autoencoder would then be to distinguish these outliers from the rest. We therefore tried both approaches.

The authors in [12] provide a review on how anomaly detection using deep learning architectures is applied in the industry which offers inspiration for the various approaches our work can take on. They also offer guidelines for appropriate model selection as well as how to go about training that can aid us.

Chapter 2
Soak Testing

Soak tests is a software development methodology that aims to test the overall endurance of a system. The notion is different from load tests in the sense that they target the long-term health of the system by running for a longer duration, such as for several days. The soak tests aim to expose any performance degradations that happen over longer time than what typical load tests would. These degradations include memory leaks, increased garbage collection or any other problems in the system. Comparatively to a load test the soak test is run with a constant number of agents such as users or servers at a high load to the system to extensively find problems [2][3].

2.1 The Neo4j Graph Database

The graph database management system Neo4j, named after the company, lets user distribute a database over multiple physical machines, together called a *cluster*. This is to ensure data availability and faster access. A distributed system presents a lot of challenges in terms of how to keep the database updated across machines and handling disruption in the event of a machine going down. To explain how it works more deeply we define three notions: server, database and database management system (DBMS). Servers can be bare metal machines, virtual machines, or containers. The DBMS manages the servers and one or multiple databases. The DBMS assigns servers to a cluster that jointly hosts a database, each keeping a copy of it. A server can also host multiple databases.

2.1.1 Autonomous Clustering

To handle disruption to the cluster Neo4j utilizes a process called *Autonomous Clustering*. Within the cluster for a database the DBMS assigns the servers to one of two modes, either *primary* or *secondary*. An examples of how databases are allocated is shown in figure 2.1 [13].

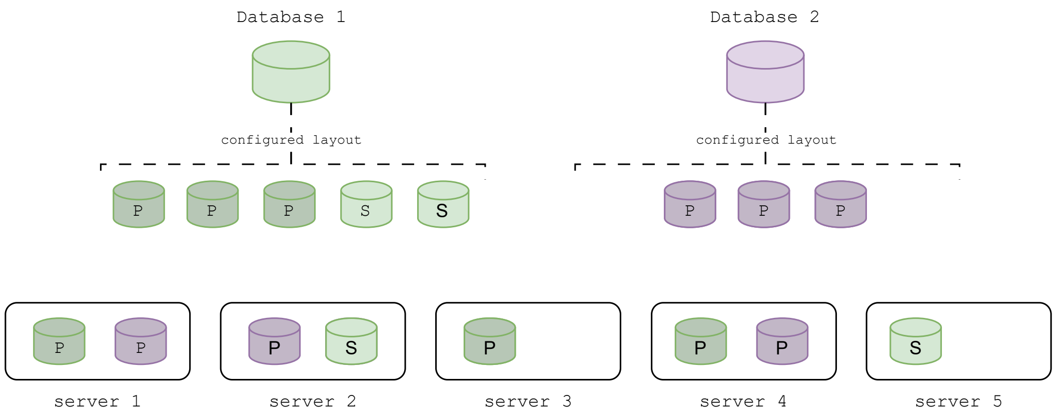


Figure 2.1: Example of a system with two databases, to the left a configuration of three primaries (P) and two secondaries (S), to the right a configuration of three primaries. Bottom shows allocation on different servers.

Servers who are *primaries* allow for both read and write transactions to the database through it. The *secondaries* only allow for read transactions to the database through them. Servers assigned as *primaries* are the point of contact for requests from clients to write to the database. A write transaction sent to one *primary* server is propagated to the rest of the *primary* servers following a protocol called Raft later mentioned. To keep each version of the database updated on the servers assigned as *secondary* they periodically poll a transaction log from *primaries* to find new write transactions and if found update their version of the database with the changes as well as adding them to their own transaction log. The more prevalent requirement from databases is a high capability of processing read transactions i.e. a larger number of *secondaries*. The idea of *secondaries* solely being responsible of keeping their copy of the database updated by pulling instead of being pushed to allows for a smaller number of *primaries* to be available in the cluster since the write transactions only go one way. The process is labeled as autonomous because of how the DBMS manages to keep a cluster intact in the case of a server abruptly shutting down. When a cluster lose connection to a member the DBMS automatically assigns another server present in other clusters to said cluster [13][14].

2.1.2 Raft Protocol

To further handle disruption to the cluster Neo4j implements the Raft protocol in addition to *Autonomous Clustering*. Raft is an algorithm that enables a cluster of computing systems to reach a *consensus* on a state they collectively manage. It works by first electing a server to be the *leader* of the cluster, the rest being labeled as *followers*. The *leader* is the entrypoint for changing the state of values and keeps track of changes through a log. The *leader* propagates updates to the log to the *followers* who in turn update their own version of the state. Meanwhile it periodically sends a heartbeat to the other members in the cluster to indicate its existence. If the *leader* fails to send a heartbeat that member is considered nonexistent and a new *leader* is elected by the remaining members [15]. The algorithm is only used for servers assigned as *primaries*. Consequently *secondaries* are not assigned as *leader* or *follower*.

2.2 Administrative test

The purpose of the administrative test is to see that the DBMS manages the clusters as expected when issued commands that affect the cluster such as change of modes or removing servers. The test creates three databases and keeps the total number of servers for each database at five throughout the test, two *Primaries* and three *Secondaries*. The test then randomly issues commands that alter the clusters concerning the three databases. We are going to reference them as *cluster commands*. The *cluster commands* and their resulting changes include:

- ***AddFollowerCommand***: Boots up a new server and assigns it as a *primary* and a *follower* in the mentioned cluster
- ***AddSecondaryCommand***: Boots up a new server and assigns it as a *secondary* in the mentioned cluster
- ***BackupCommand***: Instructs the cluster to make a backup of the database
- ***ReplaceFollowerCommand***: Removes a server assigned as a *follower* from the cluster and boots up a new server and assigns it at as a *follower*
- ***ReplaceLeaderCommand***: Removes the server assigned as the *leader* and boots up a new server and assigns it at as a *primary*
- ***ReplaceSecondaryCommand***: Removes a server assigned as a *secondary* from the cluster and boots up a new server and assigns it at as a *secondary*
- ***StopRemoveLeaderCommand***: Removes the server assigned as the *leader* from the cluster without replacing it. This forces the DBMS to rebalance to keep the desired number of *primaries* and *secondaries*. This also initiates an election process to elect a new *leader*
- ***StopRemoveSecondaryCommand***: Removes a server assigned as a *secondary* from the cluster without replacing it. This forces the DBMS to rebalance to keep the desired number of *secondaries*
- ***StopRemoveFollowerCommand***: Removes a server assigned as a *follower* from the cluster without replacing it. This forces the DBMS to rebalance to keep the desired number of *primaries*

In addition to issuing *cluster commands* the test also issues queries in the *Cypher Query Language* [16] to the database which we will reference as *cypher commands*. The *cypher commands* are issued in order to keep a workload that performs steady read and write operations as would be the case in a production database. The test is configured so that the number of queries sent to a database is constant at 500 transactions per second.

Chapter 3

Anomaly Detection and Machine Learning

3.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) is a subset of machine learning that draws inspiration from how biological brains gain knowledge from the environment. An ANN much like a brain is comprised of many connected neurons, referenced here as nodes, that collectively form a network. As with a neuron synapse, these connections have a weight associated with them that influences the strength of the signal from one node to another. The network receives an input and propagates the output from the nodes throughout the network to ensuing nodes to end up with an output. During the learning process each input is paired with a correct output that the network can use to calculate what's called a loss between the correct and actual output. Learning then occurs by adjusting the weights to be larger or smaller based on how high the loss was. Through repeated adjusting of the weights the actual output will move towards the correct output [17][18].

3.1.1 Perceptron

The perceptron consists of a single node with inputs $x_1, ..., x_k$ as a vector $\bar{x}$ each with an associated weight vector $\bar{w}$. The node performs a weighted summation of input vector $\bar{x}$ and outputs a new signal $\hat{y}$ as pictured in equation 3.1.

$$\hat{y} = \varphi\left(\sum_{k=1}^{K} w_k x_k + b\right) \tag{3.1}$$

An activation function $\varphi(a)$ is applied to the weighted summation to create the final output $\hat{y}$. As seen in equation 3.1, a bias term b is also added to the signal. The purpose of the activation function is inspired by the biological neuron where a strong input gives a larger output and a threshold behaviour exists to trigger the neuron. There exists different activation functions for various purposes. Two common ones are sigmoid and ReLU pictured in equation 3.2.

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \quad , \quad \text{ReLU}(x) = \max(0, x) \tag{3.2}$$

Another common one, softmax, takes a vector of values and adjusts them so that the sum equals to 1.

For each input $\bar{x}$ we have a corresponding expected output labeled y. The learning process that covers classification involves an optimization problem where the loss $E(\bar{w})$ between $\hat{y}$ and y is to be minimized. Through a process called gradient descent we can iteratively adjust the weights until we reach a satisfiable loss. There exists different ways of calculating loss that depends on the objective of our model. For regression the mean squared error (MSE) is used and for classification a formula called cross-entropy is used. For cross-entropy loss and binary classification where we only have two classes, 0 or 1, we calculate how much the actual output $\hat{y} \in [0, 1]$ differs from the expected output y. A logistic activation function $\varphi(a)$ turns the output from the weighted summation to a value within the range. To get the class prediction we round the value to the nearest integer. The perceptron is a core component in machine learning and is what in larger numbers make up a neural network [17].

3.1.2 Neural Network

A feed forward neural network is an evolvement of the perceptron that incorporates multiple connected nodes to form a network. The nodes are grouped into sections called layers of which there exists multiple. A feed forward neural network can have connections that

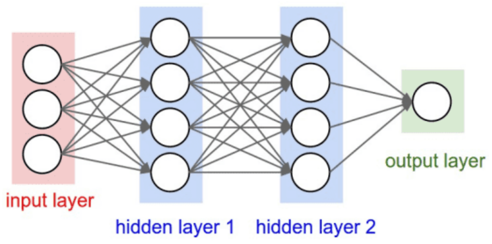


Figure 3.1: A fully connected neural network showing the input layer, two hidden layers and the output layer with their accompanying nodes (circles) and weights (lines) [19].

skip certain layers, such as a connection from an input layer directly to a second hidden layer. Comparatively, a neural network where each connection is only between a layer h_n and the next layer h_{n+1}, as pictured in figure 3.1, is called a multi-layer perceptron. The only requirement for the two types is that the connections go from left to right and that there are no feedback. The networks are structured to receive an input, propagate it through all the nodes in the network and produce an output. Gradient descent training occurs by comparing the actual output to the expected and backpropagating through the network and adjusting the weights [17][18].

3.1.3 Recurrent Neural Network

Recurrent neural networks (RNN) are a subset of neural networks that process an ordered sequence of values such as a sentence "He", "is", "happy", a sequence of words, or a time series of the temperature during a day. The network takes an input value $x(t)$ from the sequence $x(0), x(1), ..., x(T)$ one at a time and then applies the previous output $h(t)$ on the input with $\varphi(a)$ representing the activation function, as seen in equation 3.3.

$$\begin{aligned} y(t) &= \varphi(Vh(t)) \\ h(t) &= \varphi(Ux(t) + Wh(t-1)) \end{aligned} \tag{3.3}$$

This makes it so that every output is dependent on the previous input and subsequently the network keeps a memory of the previous inputs in the hidden node $h(t)$. If we "unfold" the network for a sequence as in figure 3.2 we can see this more clearly. We notice that the hidden

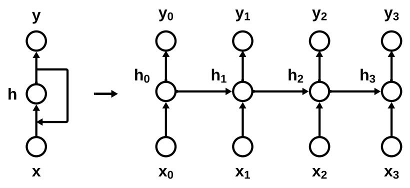


Figure 3.2: A folded RNN to the left with input node x, hidden node h and output node y. Hidden node has the previous value as input aswell. To the right, the RNN unfolded for each timestep with an input x_t and output y_t for each timestep t.

nodes weight W is shared across all of the steps in the sequence which enables the network to draw conclusions from every part of the sequence. A recurrent neural network is trained in the same manner as a feed-forward network through backpropagation by adjusting the weights to get the output to as closely possible match the input [7][17].

3.1.4 Long Short-Term Memory

One shortcoming with recurrent neural networks is that for very long sequences, since each sequence step depend on all previous, the gradient calculation causes values to vanish or explode slowing down training. Since RNN's can work poorly on longer sequences, the Long Short-Term Memory (LSTM) network was introduced [20]. The overall layout and idea of the network is the same apart from that instead of a hidden node as we had with an RNN we use a LSTM cell as pictured in figure 3.3. The output $h(t)$ is calculated based on a series

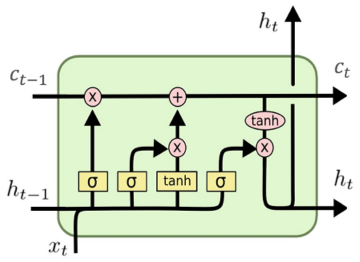


Figure 3.3: LSTM cell with the value x_t at time step t in the sequence, h_{t-1} value from the previous cell and c_{t-1} value from the previous cell as input. Output consists of the current h_t and c_t [21].

of operations including the input $x(t)$ and the output $h(t-1)$ from the previous LSTM cell. The cell also outputs $c(t)$ which is considered the internal memory of the cell. The LSTM deals with the issue of vanishing and exploding gradients by intentionally forgetting previous information with the network deciding on its own when in the sequence to forget. While this may seem counterintuitive when our objective is to form some sort of memory into how sequences are structured we only reset the long term memory. This is where the name comes from, we essentially end up with short-term memory on long sequences [7][17].

3.1.5 Convolutional Neural Network

A convolutional neural network (CNN) is another subset of neural networks that deals with data that has a grid-like structure such as an image with a 2D grid of pixels. They have seen a huge success of applications in areas such as image and video recognition. CNNs are able to capture spacial relations between inputs such as, in very simple terms, the occurrence of the color red forming a specific shape in images picturing apples. Through multiple layers, a network is able to distinguish different features inherent to the input in each layer. This means that one layer might focus on say the brightness of an image while another focuses on features like strong lines. In the most basic form the network performs a mathematical operation called *convolution* on the input $I(i, j)$, in this case an image, with the help of a *kernel* $K(m, n)$ as shown in equation 3.4, but expanding it to a sum.

$$H(i, j) = \sum_{m,n} I(i + m, j + n)\, K(m, n) \tag{3.4}$$

The values in the *kernel* $K(m, n)$, in the 2D case a matrix, influence the output $H(i, j)$ by accentuating different features of the input like those mentioned previously. The learning process then consists of adjusting these kernel values which we do using back-propagation. [7][17]

3.1.6 Transformer Architecture

A Transformer is a recent deep learning architecture proposed in [22] that tackles the downsides of existing recurrent networks (RNN, LSTM) with regards to taking a long time to train. Recurrent networks process one step at a time in a sequence which impacts the time to train a model when the sequences are especially long. The Transformer on the contrary takes the whole sequence as input which drastically cuts down training time. Transformers also utilizes a novel mechanism called attention, first mentioned in [23], implemented to improve how models take context in a sequence into account when predicting the next time step. Giving models the ability to understand context helps with tasks such as language translation where predicting the next word often depends on the context of the sentence [23][24].

General Attention Mechanism

Consider half a sentence, which is just a sequence of words, and we wish to somehow predict the next word in the sentence. A simple way would be to find the most common word in all published texts that comes after our previous word. A better way would be to take the context of the whole sentence into account. The attention mechanism allows for this by giving more weight to certain parts of the sentence when predicting our next word [23][24].

Transformer

The authors in [22] propose a variant of the attention mechanism, labeled Multi-Head Attention, for the Transformer. The full architecture of the Transformer is pictured in figure 3.4. The Transformer, originally proposed for natural language processing purposes, is com-

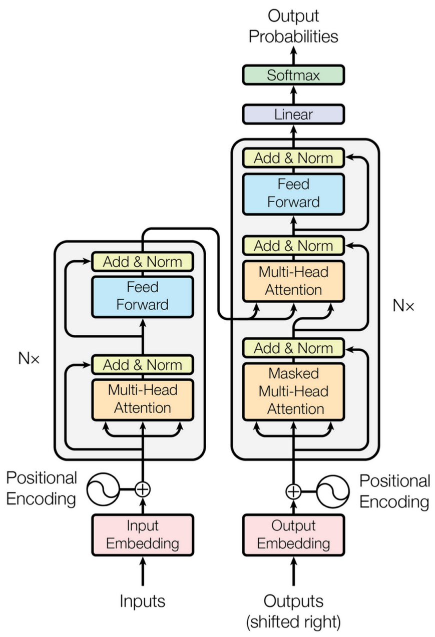


Figure 3.4: Transformer model with the encoder to the left and decoder to the right. Output from the encoder acts as input to the decoders middle part Multi-Head Attention layer. Target sequence also goes in to the decoder. Linear layer at the end as well as softmax produces probabilities for the target sequence [22].

prised of two components: Encoder and Decoder. It takes word embeddings of a sequence as input to the Encoder and target word embeddings as input to the decoder and outputs in this case probabilities for each target. The Transformer can also output a sequence if a different activation function than softmax is used. The Transformer architecture features a fully connected feed forward neural network in both the Encoder and Decoder.

3.2 Learning approaches

There exists multiple ways of training a machine learning model with respect to what type of data is available. Two main approaches are mentioned in this section: Supervised learning and Unsupervised learning.

3.2.1 Supervised Learning

Supervised learning is a concept that means to provide the model with an answer to a given input during training to increase learning. In the case of image recognition this constitutes giving the model an image of an animal, letting it predict the animal and afterwards giving it the actual answer. For supervised learning to be possible, a dataset is needed where for each sample there exists a label associated to the feature [18].

3.2.2 Unsupervised Learning

In contrast to supplying the model with labels during training, there exists an approach that seeks to let the model itself differentiate between patterns and label them. Some algorithms use a clustering approach that groups patterns together that share similarities while others seek to find a latent space representation that represents the majority of samples [18].

3.3 Time Series Anomaly Detection

3.3.1 Unsupervised Classification

An Autoencoder is a neural network model that consists of two parts: the encoder and the decoder. The size and number of layers of the encoder and decoder often have a mirrored symmetry, seen in figure 3.5, where the size of the layers decreases moving to the center, forming a butterfly design. The encoder takes an input and through the series of layers decreases

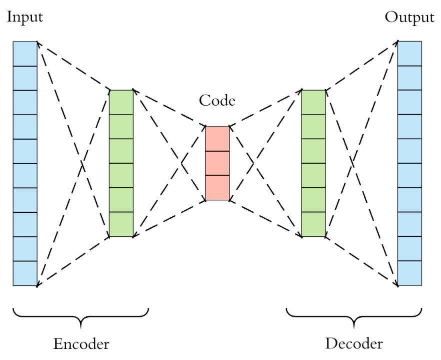


Figure 3.5: Depiction of the layers for an Autoencoder neural network model featuring an input layer, hidden layer, middlemost latent space, hidden layer and an output layer in order left to right [25].

the dimension of the data until reaching the middlemost layer, often called the latent space representation. The decoder takes this representation of the input and tries to reconstruct it through its layers to the original dimension. The goal of training is to minimize the loss between the input to the encoder and the reconstruction from the decoder. The loss is commonly defined as the mean squared error (MSE) or mean absolute error (MAE). In terms of anomaly detection the hope is that the model learns properties of the data that constitute normality. When given an anomalous input the representation loss will then be much greater signaling the presence of an anomaly [18][17]. By calculating the distribution of reconstruction losses on a dataset one can identify a threshold value that distinguishes normal from anomalous samples. The threshold can then be used for binary classification [6].

Training can employ a supervised approach where the set of samples include only those considered normal. The model then learns a representation of normality and reacts through a poor reconstruction when given an anomalous sample. Training can also employ an unsupervised approach if the set of samples are not labeled. The notion is that since an anomaly is an outlier it therefore does not occur more often than non-anomalous patterns, such that a representation of normality can still be achieved.

Employing an Autoencoder on sequence data can be achieved by using LSTM layers instead. The dimension is then reduced by decreasing the number of hidden nodes $h(t)$ in each subsequent layer [6].

3.3.2 Supervised Classification

Supervised classification using deep neural networks has gradually become a widely used machine learning method due to its performance in areas such as computer vision and natural language processing. Other supervised classification approaches such as support vector machines, random forests and decision trees have historically been widely used for supervised classification [26]. In advent of the successes with neural networks, research has also been done with supervised time series classification. One of the research areas is convolutional neural networks (CNN) that have gained popularity due to many successful implementations in various fields such as computer vision. The idea behind using CNNs for time series revolves around the notion that a convolution can be seen as a filter that moves along the time series in one dimension. Because we are using time series the input is 1-dimensional (time) unlike 2-dimensional as with images. A general form of how applying a filter on a time series is given in equation 3.5:

$$H(t) = \varphi\left(b + \sum_{n} I(t+n)K(n)\right) \tag{3.5}$$

Where $H(t)$ denotes the result of a convolution at time step t applied on a univariate time series $I(t)$ of length n, a kernel $K(n)$ and a final activation function $\varphi(a)$. Through the use of multiple convolutions we can create multiple versions of the same time series that accentuate different features inherent to it and therefore in theory learn more discriminative features [7]. The usage of channels in CNNs also gives us the added benefit of being able to handle multivariate time series where each time series is on its own channel [27]. To enable classification the result of the convolutional layers acts as input to an MLP whose last layer in the case of binary classification outputs a value in the range 0,1 through a sigmoid operation. The output value represents a prediction of which class the input is closest to. Consequently a prediction of 0.7 would categorize the input as pertaining to the class 1. The result of the convolutional layers is flattened such that an 4x4 image gives an input to the MLP of size 16 $(4 * 4 = 16)$ [17]. The MLP can feature multiple non-linear layers with activation functions such as ReLU but can also be as simple as a perceptron. The process of training the model is identical to a feed-forward network where the network is given the time series as input and compares the predicted output (i.e the predicted class 0/1) to the expected output [7]. In case of binary classification the loss function of choice would be binary cross-entropy. This is then followed by back-propagation in order to update the weights accordingly [7].

3.3.3 Classification Evaluation Metrics

Having derived a model the process of evaluating it presents a few options. With binary classification let us consider two classes, positive or negative, that a given sample belongs to. Classifying a sample as positive and it actually being positive is defined as a true positive (TP). Similarly if the sample is positive and is classified as a negative this counts as a false negative (FN). Visually this is shown in figure 3.1. Trivially the blue diagonal with the true

		Predicted	
		P	N
Actual	P	TP	FN
	N	FP	TN

Table 3.1: Confusion matrix with actual class on the y-axis and predicted class on the x-axis. Blue diagonal represents number of correct classifications, green diagonal represents number of incorrect classifications.

positives and true negatives represents the number of correct classifications from a set of samples and the green represents the number incorrect classifications [28]. The resulting table is called the confusion matrix and lets us identify irregularities with classification such as predicting one class with higher frequency. Observing the ratio of correctly classified samples is an important metric for evaluation. Sensitivity represents the ratio of correctly classified positive samples out of the total number of positive samples. Specificity represents the ratio of correctly classified negative samples out of the total number of negative samples. Formulas for these are seen in 3.6[28].

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP+FN}} \qquad \text{Specificity} = \frac{\text{TN}}{\text{TN+FP}} \tag{3.6}$$

Another important metric is Accuracy, defined as the ratio between correctly classified samples and the total amount of samples, formulated in 3.7.

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}} \tag{3.7}$$

An imbalanced set of samples, where samples of one class outnumber the samples of the other class, affect the Accuracy since it uses values from both the positive and negative row. When working with an imbalanced set of samples it is therefore recommended to instead focus on Sensitivity and Specificity. Another option is defined as Balanced Accuracy that takes the latter metrics into account to produce a better measure of accuracy for imbalanced sets of samples. The formula is specified in 3.8.

$$\text{Balanced Accuracy} = \tfrac{1}{2}(\text{Sensitivity} + \text{Specificity}) \tag{3.8}$$

Chapter 4
Method

4.1 Metrics Selection

In order to achieve a model that can successfully identify anomalies in the test data, the most straightforward way is to mimic how manual inspection is carried out. We therefore identified the most significant metrics that during a manual inspection tell us if the test has failed or contains anomalies. We identified them through interviews with people within the company and by attending inspections. In total we identified six metrics. The following sections describe the metrics we identified as most significant during manual inspection.

Throughput

Throughput is a metric that states the amount of reads and writes a server hosting a database processes in a given amount of time. We were able to separate the metric into two, namely one for the read throughput and one for write throughput. A drop in throughput tells us that the machine is not processing as many read or writes as it should, because of the load balancing property mentioned previously in section 2.2, and is a stark sign that the test has failed.

Cypher commands

As mentioned previously in section 2.2 the test generator issues cypher commands to the database. If the database does execute the command this is recorded as a successful command. If the database fails to execute the command in any manner this is recorded as a failed command. Each command has both a sequence of successful and failed counts. The number of commands were 24 so we ended up with 48 sequences in total. A decrease in the number of successful commands and subsequently an increase in failed ones do indicate that the test may have failed.

Cluster commands

Cluster commands are as mentioned previously in section 2.2 instructions issued by the load generator to the cluster of machines as a whole to in some way alter the structure of the cluster. As with cypher commands we have sequences of both failed and successful commands issued during the test. The number of commands were nine so we ended up with 18 sequences in total.

Cypher replan events

The replan option for a cypher query gives the option to decide whether a query should be recompiled or not. Not recompiling a query could be beneficial when dealing with expensive computations and knowing that the data has not changed since the last time a query was sent. This metric states the total number of times a query has been re-planned for a database. If we see large and frequent spikes this could indicate an issue. Since the administrative test consists of three databases we ended up with the same number of sequences. We took the derivative to produce a sequence concerning number of cypher replan events per time step.

Check pointing time

Check pointing is an action where all pending updates from volatile memory to non-volatile memory are flushed. This is to minimize the time required for recovery after an improper shutdown or crash has occurred in a database. The metric states the total time, in milliseconds, spent in check pointing so far. We took the derivative to produce a sequence concerning total time spent check pointing per time step.

4.2 Data Gathering

Metrics gathered during the duration of the system test, and later used as documentation for the manual inspection, was archived at the end of the test. We proceeded by fetching all the archives and wrote a script to recursively unpack the archives and extract the relevant files concerning the most significant metrics we previously identified. The time series sequences were formatted in a file format used by the monitoring tool called Graphite. The files contained the same time series but with different durations between each measurement: 1m, 5m, 10m. We thereafter used a script to dump the contents of the files into text files with the time series sequences intact. Using a Python script the contents of the text files were imported into a pandas *DataFrame* with each filename represented as a column and each time step represented as a row. Graphite conveniently enough stored the UNIX timestamp for when each "measurement" was received so we were able to match up all the rows in every column. The *DataFrame* was thereafter converted into a csv file and stored. In total we ended up with 36 csv files that were later used to build the dataset.

4.3 Data Formatting

The nature of the test meant that the number of possible features to the model would change with every occasion. One example of this is that the total number of machines spun up, and that at some point were part of a cluster, differs between runs. This is due to the way the test operates, as detailed in section 2.2. This entails that the number of Throughput, Check pointing and Cypher replan event sequences is different from each test run. To address this we decided to use dimensionality reduction that allowed us to from a variable amount of sequences get a single representation of the sequences. We also performed transformations on few selected metrics. We ended up with 16 features consisting of time series.

4.3.1 Dimensionality Reduction

Below is an overview of the metrics that differ in number between test runs and how a lower dimensional representation was constructed.

Throughput

At first glance a way to concatenate all the throughput sequences would be to add them all together. However due to the load generator adjusting the throughput to the other machines when a drop occurs elsewhere we would end up with a sequence that does not contain any information that a drop in throughput in a machine has occurred. To preserve the information in the sequence that a drop in throughput in one of the machines had occurred we calculated the variance in throughput over all the sequences at a given timestep.

Cypher & Cluster commands

We did not face the same issue with the sequences of cypher and cluster commands so we proceeded with dimensionality reduction by adding together the values at each timestep of each command to form one sequence. Subsequently we ended up with two sequences: one with counts of all successful commands and one with counts of all failed commands

4.3.2 Transformations

In accordance to how the team handles these metrics and uses them to perform manual inspection we did additional transformations on some metrics. This includes taking the derivative of sequences for Cypher replan events and Check pointing times.

4.3.3 Normalization

The training set was normalized using min-max normalization. Each feature was normalized independently of the others because of the difference in scale between them. The validation- and test-set was normalized in the same way but using the min, max values previously calculated on the training set.

4.3.4 Labeling

Since the previous soak tests had undergone manual inspection the team were aware of which test instances that were classified as passed or failed. Through manual inspection the team were able to identify at which point (unix timestamp) in the time series that a problem first became apparent. With the knowledge of where a problem first appeared an additional column was added to each csv file and each measurement was labeled with a 1 representing a pass and a 0 representing a fail. Some leeway of approximatively -2 hours were added with the notion that the problem might be apparent earlier and that a model might learn what happens before a problem occurs and in a sense see into the future.

4.4 Creating the datasets

To enable flexibility in our working process we implemented a custom *Dataset* class derived from the base class in PyTorch. The class imported all the available CSV files into a single *DataFrame* and returned three datasets considered for training, validation and testing containing the input features and target labels. All measurements were divided into sequences of a length specified when instantiating the class. The last sequence at the end of a test instance might not have enough measurements to satisfy the length specified so these sequences was padded with -1, intentionally since no time series sequence contain negative values. All the sequences were thereafter labeled using the notion that a whole sequence is a fail if it contains any measurement deemed a fail. Since the distribution of test instances that failed or passed was uneven, randomly sampling sequences for the three datasets could lead to an even greater imbalance. The *Dataset* class therefore utilizes stratified sampling to the extent that a train dataset that contains 70% of all sequences contains 70% of all sequences considered passed and 70% all sequences considered failed. The dataset consisted of 36 test instances of which 25 and 11 were considered to have passed and failed, respectively.

4.5 Anomaly detection

4.5.1 LSTM Autoencoder

We implemented an autoencoder architecture with hidden LSTM cells instead of regular hidden nodes that you will find in a typical autoencoder. The implementation was written using PyTorch. An overview of the architecture that we found to work best can be seen in table 4.1. We found that using the mean squared error (MSE) between the input sequence and reconstruction sequence during gradient descent provided the lowest loss compared to mean absolute error (MAE). As an optimizer we used NAdam. We trained the model using minibatches of a specified size and for each epoch calculated a MSE loss on the whole validation set. We used the loss on the validation set to adjust the hyperparameters of the model to counteract overfitting and bring down both the training and validation loss. The model was trained with various sequence lengths and we found that a sequence of 60 measurements, i.e. 60 minutes, provided the lowest loss. To allow us to classify a sequence as anomalous we proceeded with deciding on a threshold of the reconstruction loss that would

Layer (Type)	Component	Input size	Output size
LSTM	Encoder	16	16
LSTM	Encoder	16	3
LSTM	Decoder	3	16
LSTM	Decoder	16	16

Table 4.1: LSTM Autoencoder architecture in order from top to bottom. Each row specifies a layer in the model with the accompanying input and output size as well as what part, encoder or decoder, the layer pertains to.

allow us to differentiate the two. To decide on a threshold we used Optuna for optimization [29]. We computed the reconstruction error on the training set using the mean absolute percentage error (MAPE) that provided better results compared to using the MSE. The idea behind using MAPE was that since each metric has a different scale that looking at the error percentage-wise would handle cases where a metric is generally much larger than the rest better and therefore not skew the result. For each suggested threshold by Optuna we calculated the accuracy by labeling a sample as anomalous if the MAPE was greater than the threshold and comparing the labeling to the true label for the sample. The computed accuracy was balanced so as to not skew the results due to there being more samples labeled as normal than anomalous in the dataset. The optimization procedure ran for 1.000 iterations until it was stopped. We evaluated the model on a test set by computing the TP, TN, FP, FN from classifying the samples using the threshold and comparing the predicted and actual label.

4.5.2 Transformer

Another model was implemented in PyTorch using the transformer architecture consisting of a encoder and a decoder as described in [11]. The model was trained with a seq2seq approach where the encoder and decoder was fed an input sequence and the decoder produced an output sequence. During training the MSE loss between the input and output sequence was calculated and the objective was to minimize it. NAdam was used as an optimizer. To optimize the hyperparameters, pictured in Table A.2 in Appendix, we used Optuna to train the model and evaluate the hyperparameters with respect to the validation loss from the last epoch of training. We proceeded by deciding on a threshold for the reconstruction error, that would constitute a sequence as anomalous, using Optuna in the same manner as for the LSTM Autoencoder. Ultimately the model was evaluated on a test set by computing the TP, TN, FP, FN from classifying the samples, using the threshold, and comparing the predicted and actual label.

4.6 Classification

4.6.1 Transformer-Encoder

A relevant research area is how Transformers can be used for text classification and more recently for time series classification [11]. One approach is to train the encoder part of the Transformer as detailed in subsection 3.1.6 [30][31]. We wanted to explore if our already trained Transformer model used for seq2seq could be re-utilized for supervised classification purposes. The encoder pertaining to our transformer was disconnected from the decoder and a fully connected layer was added to the output layer from the feed-forward-network of the encoder. The fully connected layer has 16 input features, one output and a sigmoid operation. Training occured only on the fully connected layer with the other weights intact using the same training set as we used for the transformer. The length of the input sequence that was used to train the transformer was also used in training the classifier. We used NAdam as optimizer and binary cross-entropy for the loss function. We evaluated the model on the test set by computing the TP, TN, FP, FN from classifying the samples and comparing the predicted and actual label.

4.6.2 CNN

From the notion that CNNs can be applied to multivariate time series as mentioned in subsection 3.3.3 we implemented a one-dimensional CNN using PyTorch with the architecture seen in figure 4.1. The model takes a vector sequence $t_1, t_2, t_3, ..., t_k$ as input. Each feature

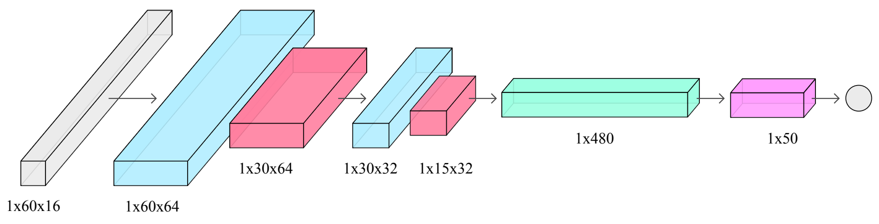


Figure 4.1: All layers ordered from left to right, input to output. Convolutional layers with padding same and ReLU (blue), max pool layers (red), flattened layer (green) and linear layer + sigmoid (purple). Width of block specifies number of channels. Depth of block specifies sequence length.

for a time series is inputed as a channel to the model. Consequently the first layer has 16 input channels. The models layers consists of a repetition of a 1d convolutional layer, ReLU activation function and a 1d max pooling layer. The convolutional layers pad the output so the input and output dimensions are the same. We experimented with the use of multiple repetitions and found that two repetitions gave the best validation accuracy. The first convolutional layer expands the number of channels to four times the amount. We found that first increasing the number of channels in the first layer worked best compared to decreasing them as we do with the other convolutional layers. Our notion is this creates more trainable parameters in the model. The output from the last 1d Max Pool layer is thereafter flattened and acts as input to a MLP with a sigmoid activation function at the end. We used NAdam as an optimizer and binary cross-entropy as the loss function. Hyperparameters of the model such as sequence length, kernel size, stride were manually optimized to find the choices that gave the lowest validation loss and balanced accuracy without overfitting. We evaluated the model on the test set by computing the TP, TN, FP, FN from classifying the samples and comparing the predicted and actual label.

Chapter 5

Result

5.1 Anomaly Detection

LSTM Autoencoder

We found the optimal threshold for the reconstruction MAPE of the LSTM Autoencoder to be 0.2610. This value was reached after 10,000 iterations. The model that performed the best was trained with an initial learning rate of 0.002 and a batch size of 32 for a duration of 100 epochs. Looking at the confusion matrices for each dataset in Table 5.1 we see that there is a large discrepancy between the two classes and that the model is more likely to predict that a sequence is non-anomalous. The test set accuracy for correctly identifying a sample

		Predicted	
		P	F
Actual	P	1086	194
	F	286	265
		TPR: .79	TNR: .58

		Predicted	
		P	F
Actual	P	317	49
	F	73	85
		TPR: .81	TNR: .63

		Predicted	
		P	F
Actual	P	161	22
	F	46	33
		TPR: .78	TNR: .60

Table 5.1: Confusion matrix for training, validation and test set on the two classes "Pass" (P) and "Fail" (F) from left to right. Sensitivity (TPR) and Specificity (TNR) below each matrix. Darker color in a cell represents a higher concentration of predicted values in that class

labeled as normal and anomalous ends up at 87.98% and 41.77%, respectively. If we compare the balanced accuracies among the different datasets, training, validation, test, we see that they are quite similar, 66.47%, 70.20% and 64.88% respectively.

Transformer

Through optimization with the help of Optuna we managed to find the hyperparameters stated in Table A.2 after 48 iterations of training for 100 epochs. It is interesting to note the significant difference in the input sequence length for the Transformer compared to the LSTM Autoencoder with it being 8 and 60 respectively. We found with the Transformer that a shorter sequence length provided a lower reconstruction loss. The resulting model exhibited significant potential due to the small reconstruction loss on test samples. We calculated the average reconstruction loss on each of the 16 in total features to get a sense of if some features were harder to reconstruct and found that they were low across the spectrum as seen in the Table A.1. We found the most optimal threshold value of the reconstruction MAPE to be 0.6322 after 10,000 iterations of searching.

		Predicted	
		P	F
Actual	P	8508	1055
	F	1976	2106
		TPR: .81	TNR: .67

		Predicted	
		P	F
Actual	P	2432	300
	F	560	606
		TPR: .81	TNR: .67

		Predicted	
		P	F
Actual	P	1224	143
	F	318	266
		TPR: .79	TNR: .65

Table 5.2: Confusion matrix for training, validation and test set on the two classes "Pass" (P) and "Fail" (F) from left to right. Sensitivity (TPR) and Specificity (TNR) below each matrix. Darker color in a cell represents a higher concentration of predicted values in that class

Looking at the confusion matrix in Table 5.2 we see similar results to the LSTM Autoencoder. The balanced accuracy where we take into account the disparity between the number of "pass" and "fail" samples ends up at 67.54% on the test set. However if we look at the accuracy independently for each class the accuracy for correctly identifying a sample labeled as "pass" and "fail" ends up at 89.54% and 45.55% respectively. Note the different amount of samples in all the datasets compared to the LSTM Autoencoder is due to a shorter input sequence length and therefore the ability to get more samples from available data. If we compare the balanced accuracies among the different datasets, training, validation, test, we see that they are quite similar, 70.28%, 70.50% and 67.54% respectively. This at least indicates that no overfitting has occurred. The notion that the Transformer performed poorly due to being trained on a collection of both anomalous and normal samples was ruled out, as training the Transformer on only normal samples produced worse results.

Visualizing the reconstruction MAPE

By calculating the reconstruction MAPE for the LSTM Autoencoder and Transformer with the test set as input we can look at the distribution. This is pictured in figures 5.2, 5.1. We see that for both models the reconstruction MAPE from *fail* sequences is to a large degree the same as those for a *pass* sequence judging from the leftmost portions of the figures. We also see a large concentration of samples located right after 0.6 in both figures and that almost no sample produce a reconstruction MAPE larger than that. The large concentration at around 0.6 matches well with the threshold value we found for the Transformer using Optuna. The threshold for the LSTM Autoencoder found using Optuna is located within the distribution concerning normal samples. Probably, not enough anomalous samples exist outside the distribution, compared to the larger spike at 0.6 in figure 5.2, to move the threshold further

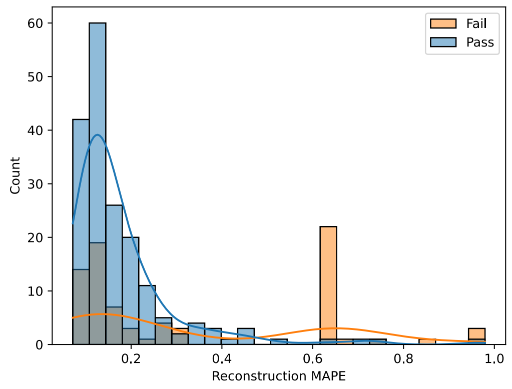


Figure 5.1: Distribution of reconstruction MAPE for all test sub-sequences labeled as pass (normal) or fail (anomalous) for the LSTM AE.

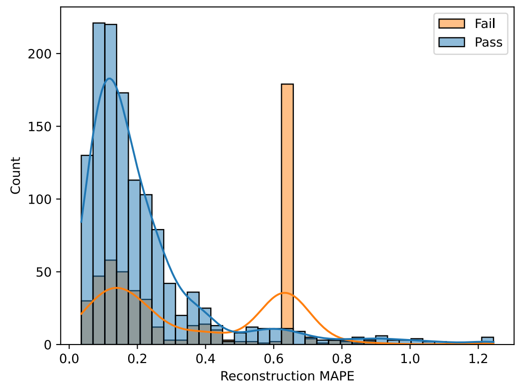


Figure 5.2: Distribution of reconstruction MAPE for all test sub-sequences labeled as pass (normal) or fail (anomalous) for the Transformer.

away. This might explain the lower accuracy of the LSTM Autoencoder compared to the Transformer.

5.2 Classification

Convolutional Neural Network

In Table 5.3 the confusion matrix for each dataset is featured for the CNN model. We see that the miss-classification is fairly low among all the datasets. The miss-classification of samples labeled as "fail" for the test set is particularly low with 6,3% miss-labeled. This success is even more impressive when you take into the fact that the training dataset had a much higher ratio of samples labeled as "passed" namely 70%.

		Predicted	
		P	F
Actual	P	1267	13
	F	15	536
		TPR: .99	TNR: .98

		Predicted	
		P	F
Actual	P	353	13
	F	8	150
		TPR: .98	TNR: .92

		Predicted	
		P	F
Actual	P	177	6
	F	5	74
		TPR: .97	TNR: .92

Table 5.3: Confusion matrix for training, validation and test set on the two classes "Pass" (P) and "Fail" (F) from left to right. Sensitivity (TPR) and Specificity (TNR) below each matrix. Darker color in a cell represents a higher concentration of predicted values in that class

Taking the balanced accuracy into account in Table 5.4 the accuracy stays consistent between each dataset which is a sign of a high grade of generalization and no occurrence of overtraining. There is a slight difference in accuracy between the two classes *pass* and *fail*, however this is expected because of the disparity previously mentioned. The model was trained with an initial learning rate of 0.0002 and a batch size of 64 for a duration of 500 epochs. We found that the model didn't produce satisfiable results until we increased the number of epochs.

Dataset	Accuracy (P)	Accuracy (F)	Balanced Accuracy
Training	0,9898	0,9728	0,9813
Validation	0,9645	0,9497	0,9569
Test	0,9672	0,9367	0,9520

Table 5.4: Accuracy concerning classifying a sub-sequence as pass or fail as well as balanced accuracy for training, validation and test set.

This is shown in figure 5.3 where we clearly see that the model takes a long time to reach a desired loss figure and even doing a large dip around 360 epochs.

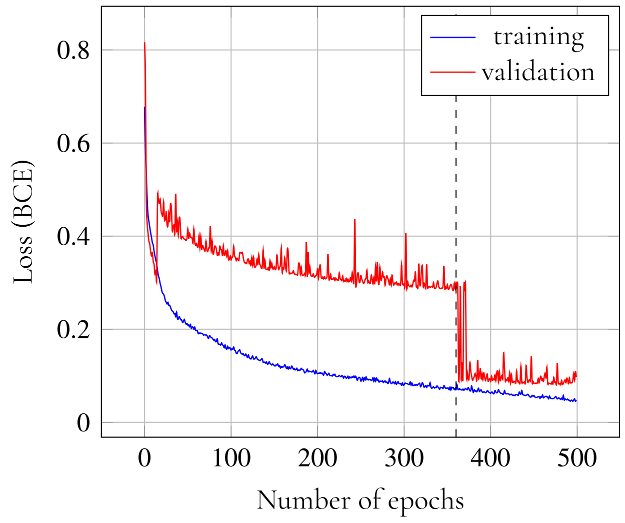


Figure 5.3: Instance of training and validation loss for the CNN model. Vertical line at 360 epochs. This specific instance was selected for evaluation.

Transformer-Encoder Classifier

The classifier consisting of the Encoder-part from the Transformer underperformed severly compared to the CNN based classifier. From the Table 5.5 we see that the Transformer-Encoder classifier is severly overfitted to the *pass* class classifying nearly all samples as belonging to that class. This accounts to a balanced accuracy of 50%, 50% and 50% on the

		Predicted	
		P	F
Actual	P	9563	0
	F	4082	0
		TPR: .70	TNR: NaN

		Predicted	
		P	F
Actual	P	2732	0
	F	1165	1
		TPR: .70	TNR: 1.0

		Predicted	
		P	F
Actual	P	1367	0
	F	583	1
		TPR: .70	TNR: 1.0

Table 5.5: Confusion matrix for training, validation and test set on the two classes "Pass" (P) and "Fail" (F) from left to right. Sensitivity (TPR) and Specificity (TNR) below each matrix. Darker color in a cell represents a higher concentration of predicted values in that class

training, validation and test set which is comparable to random assignment. Looking at the specificity, i.e. the proportion of actual fail samples correctly identified, it lands at close to zero which is reasonable looking at the confusion matrices where almost no fail samples are correctly classified. The model was trained for 100 epochs with a learning rate of 0.002, batch size of 32 and a sequence length of 8 time steps.

Chapter 6

Discussion

Reflections

Initially when starting out we imagined that the LSTM Autoencoder would produce satisfiable results in terms of accurately identifying an anomalous sequence. Mainly due to the prevalance in using autoencoders and LSTM networks for anomaly detection. We also imagined that supervised classification would perform subpar due to our small dataset. When developing the LSTM Autoencoder we started with a simple model of six layers with a bottleneck of 3 nodes while using MSE as a loss function. We found that using MAE provided a better result instead of MSE. We were however not satisfied since the model still suffered from overfitting. We found that MAE outperformed MSE when the complexity grew, and the other way around when the complexity shrank. Our notion with testing MAE came from the fact that it is less affected by outliers compared to MSE that accentuates outliers due to squaring the difference.

As stated previously we experienced an accuracy comparable to the LSTM Autoencoder when we initially started out with the CNN classifier. What changed this was the addition to the first layer that the number of out channels are 4x the number of input channels, i.e. from 16 to 64. This widely increased the accuracy of the model and one can imagine due to the increase of trainable parameters that the model is better at learning the data when dealing with a higher dimension. We started out with three repeated segments of a convolutional layer, ReLU and a max pooling layer as presented in subsection 4.6.2 and found the accuracy to be very good, however we noted that the validation loss had large spikes i.e. "wobbly", while still keeping a downward trend. Reducing the complexity of the model by removing one of the three repeated segments of layers removed this tendency. Our findings are in agreement with previous research, seen in [7][8][9][10], that demonstrate similar success in utilizing 1d convolutional neural networks for time series classification.

Based on the results from visualizing the reconstruction loss for the LSTM Autoencoder and the Transformer it becomes more clear why they both performed subpar. With the distributions concerning *pass* and *fail* sequences merging into one another it is hard to decide on a threshold that achieves a high accuracy on both classes. There could be multiple culprits for this result. Firstly the dimensionality reduction we performed on the original metrics during the data formatting stage section 4.3 could have removed certain aspects to the sequences that are prevalent when an anomaly is present. Secondly it might be a model issue where it does not have enough complexity in terms of trainable parameters to be able to draw conclusions from the input sequences. It is notable that the authors in [5] also did not achieve satisifable results with an LSTM Autoencoder, while attempting anomaly detection on metrics from the Neo4j database running in production. The authors attributed the reason that data was not gathered in a controlled environment, something that cannot be said for our case.

It is interesting to note how low of a reconstruction loss the Transformer model was able to produce and that the reconstructed sequences were nearly identical to the input sequences. This is in contrast to the poor result of the Transformer when trying to classify a sequence using the threshold. The possible culprits are the same as those stated before, namely that we have lost aspects of the data that indicate an anomaly or having to small of a dataset. This is evident due to the similarity of distribution of reconstruction loss for the Transformer and LSTM Autoencoder as pictured in figure 5.2. Contrary to our results previous research into Transformers and multivariate time series [11] showed promising results. In the paper the time series data consisted of electrocardiagram (ECG) readings. We hypothesise that the difference in accuracy between our work and theirs could be explained by ECG readings being more periodical than our data.

When attempting to find the most optimal threshold value for the reconstruction loss regarding the LSTM Autoencoder and Transformer models we calculated the loss on the set previously used for training the models, as mentioned in section 4.5. We initially calculated the reconstruction loss using mean squared error (MSE) on all 16 input features. Adding the notion that the scale of the values for the various features differed greatly meant that calculating the MSE would therefore skew the loss in favour of one or multiple features with a greater range of values while downplaying those with a smaller range. To counteract this we proceeded with using mean absolute percentage error (MAPE) to compute the reconstruction loss. MAPE counteracts the issue by instead looking at the percentage ratio that the deviation constitutes for each feature and then taking the average of all percentage ratios. The change had an impact on the resulting accuracy and when looking at the distribution of reconstruction losses showed improvement in the sense that the two classes were less merged. Judging from the results it would make sense to extend the use of MAPE to the loss function used during training of the models. Due to time constraints we were unable to investigate this.

When we tried different sequence lengths for the Transformer we found that the reconstruction loss was higher with longer sequences. The same sequence length as for the LSTM Autoencoder performed comparably. This goes against the notion that a Transformer should handle longer sequences better than a LSTM. The optimal sequence length ended up at 8 which is a far way from the LSTM Autoencoders sequence length of 60. Why this is the case is unclear.

Research Objective

Looking back at our first research question we demonstrated with the LSTM Autoencoder and Transformer that the amount of anomalous sequences detected did not correlate well with the sequences manually identified as in need of further scrutiny. This is due to the low accuracy of both models which were only able to correctly detect less than half of the anomalous sequences. Since the Tranformer and LSTM Autoencoder performed around the same one could argue that the models are not to blame but the data formatting, specifically the dimensionality reduction, or the data itself that inhibit the models from differing between non- and anomalous sequences.

The third research question argued whether a model can outright classify if results from a soak test constitutes a pass or fail. Regarding this we propose a model that with a high degree of accuracy can classify if a sequence spanning 60 time steps (60min) is considered *pass* or *fail*. This constitutes being able to outright classify if a soak test should be considered to have passed or failed by dividing the full time series into sub-sequences and use the model on them independently. By answering the first and third research question we also answer the second one, namely if we can identify where in the time series that anomalies are present. This answers itself since our models deals with sub-sequences of the full soak test and individually examines them for anomalies.

During the study appropriate measures were taken to address threats to validity. As mentioned in section 4.4 the dataset was split according to the train-validate-test paradigm that intends to infer validity and reliability of machine learning models. We also employ stratified sampling such that the amount of pass and fail sequences in each subset of the dataset is representative of the whole dataset, which adds to the validity and reliability. The study also details the hyperparameters and architectures used which greatly improves reproducibility. A threat to the validity is the fact that the models have only been evaluated on soak tests from the Neo4j graph database. Questions regarding generalization is therefore left unanswered.

Future Work & Applications

We propose a few directions that further work can take. Firstly, it would be interesting to apply the notion of an exploding layer, as seen in the CNN, to the LSTM Autoencoder and investigate if it improves the reconstruction loss. Secondly, due to the success in finding a threshold using MAPE compared to MSE it would be interesting to also use MAPE during training, i.e. to compute the reconstruction loss. This would probably decrease the importance that is laid on certain features. One could also imagine using another loss function that take the difference in scale between features into account.

The results demonstrate one potential use case that machine learning can have at Neo4j to facilitate the verification process. We propose a model that accurately can classify a soak test as having passed or failed as well as identify where during the duration of the test that an anomaly first occurred. Although results are promising further evaluation is needed by the company to ensure that the quality of released software does not decrease with an automated soak testing approach. A good approach is to first utilize the model alongside manual verification. The model could be integrated in the release process that already exists to automatically output predictions on the soak test and present them to the overseer. Although our models were trained on data gathered from an *administrative soak test* it would be interesting to apply the model to data gathered from other variations of soak tests used at Neo4j. This would showcase if the model learnt underlying generalizations that exist between tests.

Since our work is fine-tuned to adhere to the workings of the Neo4j graph database we imagine that the work has little bearing elsewhere in the industry. The nearest area that may benefit from our research is other providers of distributed databases that may find inspiration in our work.

Chapter 7
Conclusion

The objective of this thesis was to investigate how machine learning could be utilized to facilitate the verification process of the results from a soak test. This included outright classifying a test as having passed or failed or provide feedback regarding behaviour that is anomalous. In this effort we collected metrics data from soak tests and identified the most relevant metrics. We explored a wide variety of different deep learning architectures which included LSTM Autoencoders, CNNs and Transformers. We later evaluated our models against our objective by comparing metrics such as accuracy and reconstruction loss. We managed to develop models that leave room for improvement but also propose a CNN model that achieves a high degree of accuracy on classifying sequences as non- or anomalous.

Circling back to our first research question our findings demonstrate that the amount of anomalies detected in unsupervised analyses does not correlate well with the need for further manual scrutiny because of how few anomalous sequences were detected. In reference to our second research question our models can in theory help identify where in the time series that anomalous behaviours present themselves but falls short due to the high-degree of miss-classification. We do however, in reference to the third research question, propose a CNN model that accurately can classify if the results from a soak test constitutes a pass or a fail based on the available data. Regarding our second research question we can however with our CNN model help identify where in the time series that anomalous behaviours present themselves due to the high degree of accuracy and the inherent nature of the model looking at sub-sequences of the whole soak test.

References

[1] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software Testing Techniques: A Literature Review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182, 2016.

[2] Mihir Sardana, Tanupriya Choudhury, and Dev Kumar Chaudhary. Extensive review on software testing and pipeline testing softwares. In *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pages 246–251, 2017.

[3] Itti Hooda and Rajender Singh Chhillar. Article: Software Test Process, Testing Types and Techniques. *International Journal of Computer Applications*, 111(13):10–14, February 2015.

[4] Mattie Salim, Erik Wåhlin, Karin Dembrower, Edward Azavedo, Theodoros Foukakis, Yue Liu, Kevin Smith, Martin Eklund, and Fredrik Strand. External Evaluation of 3 Commercial Artificial Intelligence Algorithms for Independent Assessment of Screening Mammograms. *JAMA Oncology*, 6(10):1581–1588, 10 2020.

[5] Evelina Danielsson and Lisa Franzén af Klint. Predicting loss of fault tolerance in a cloud graph database, 2023. MSc thesis, Department of Computer Science, Lund University.

[6] Oleksandr I. Provotar, Yaroslav M. Linder, and Maksym M. Veres. Unsupervised Anomaly Detection in Time Series Using LSTM-Based Autoencoders. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, pages 513–517, 2019.

[7] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data mining and knowledge discovery*, 33(4):917–963, 2019.

[8] Ji Sung Lee, Hyeon Sung Cho, Kyo Il Chung, and Ji Sang Park. Feature Selection for Stock forecasting using Multivariate Convolution Neural Network. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1270–1272, 2020.

[9] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, 28(1):162–169, 2017.

[10] Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, and J Leon Zhao. Exploiting multi-channels deep convolutional neural networks for multivariate time series classification. *Frontiers of Computer Science*, 10:96–112, 2016.

[11] Abrar Alamr and Abdelmonim Artoli. Unsupervised Transformer-Based Anomaly Detection in ECG Signals. *Algorithms*, 16(3), 2023.

[12] Kukjin Choi, Jihun Yi, Changhwa Park, and Sungroh Yoon. Deep Learning for Anomaly Detection in Time-Series Data: Review, Analysis, and Guidelines. *IEEE Access*, 9:120043–120065, 2021.

[13] John Stegeman. Easily Run, Scalable, Fault Tolerant Graph Databases With Neo4j Autonomous Clustering. `https://neo4j.com/developer-blog/scalable-fault-tolerant-graph-databases-neo4j-autonomous-clustering/`. [Accessed 2023-11-10].

[14] Neo4j. Clustering / introduction. `https://neo4j.com/docs/operations-manual/current/clustering/introduction/`. [Accessed 2023-11-10].

[15] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[16] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD'18 Proceedings of the 2018 International Conference on Management of Data*, page 1433, Houston, United States, June 2018. ACM Press.

[17] Patrik Edén and Mattias Ohlsson. *Lecture Notes on Introduction to Artificial Neural Networks and Deep Learning*. Media-Tryck, 2022.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[19] Stanford University. CS231n Convolutional Neural Networks for Visual Recognition. `https://cs231n.github.io/convolutional-networks/#add`. [Accessed 2023-11-15].

[20] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[21] Olah, C. Understanding LSTM Networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Accessed 2023-11-15].

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[23] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[24] Navaneeth Malingan. Attention mechanism in deep learning. https://www.scaler.com/topics/deep-learning/attention-mechanism-deep-learning/. [Accessed 2023-12-04].

[25] Maha Alkhayrat, Mohamad Aljnidi, and Kadan Aljoumaa. A comparative dimensionality reduction study in telecom customer segmentation using deep learning and PCA. *Journal of Big Data*, 7:9, 02 2020.

[26] Zhaodong Wu, Jun Zhang, and Shengliang Hu. Review on Classification Algorithm and Evaluation System of Machine Learning. In *2020 13th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, pages 214–218, 2020.

[27] Raneen Younis, Sergej Zerr, and Zahra Ahmadi. Multivariate Time Series Analysis: An Interpretable CNN-based Model. In *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10, 2022.

[28] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 17:168–192, 1 2021.

[29] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[30] Sai Wu, Zhiyin Huang, and Hao Feng. Text Labels Classification Model based on BERT Algorithm. In *2023 4th International Conference on Big Data Artificial Intelligence Software Engineering (ICBASE)*, pages 329–332, 2023.

[31] Aaryan Jagetia, Umang Goenka, Priyadarshini Kumari, and Mary Samuel. Visual Transformer for Soil Classification. In *2022 IEEE Students Conference on Engineering and Systems (SCES)*, pages 1–6, 2022.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Appendices

Appendix A
Supporting tables

MSE	Metric
8.947875102991174e-05	All (Training)
0.0006488995277322829	All (Validation)
0.004095897078514099	database one throughput variance read
0.00229713530279696	cypher command fail count
0.0006401611026376486	cypher command success count
0.0004514287575148046	database one cypher replan events
0.0004140881064813584	database two cypher replan events
0.00033021011040546	database two check pointing duration
0.0002947737812064588	cluster command fail count
0.0002814968174789101	database one check pointing duration
0.00027126181521452963	database three check pointing duration
0.00025484495563432574	database one throughput variance write
0.00025009657838381827	cluster command success count
0.00020307455270085484	database two throughput variance read
0.0001932494924403727	database three cypher replan events
0.00016869704995770007	database two throughput variance write
0.00016436594887636602	database three throughput variance write
7.161071698646992e-05	database three throughput variance read

Table A.1: Individual MSE for each feature concerning the Transformer.

sequence length	8
learning rate	0.0003769
batch size	64
number of heads in the multi-head attention layer	1
number of encoder layers	3
number of decoder layers	3
number of nodes in feed forward network (encoder)	2901
encoder dropout rate	0.6114632
encoder layer norm value	0.0283474
number of nodes in feed forward network (decoder)	580
decoder dropout rate	0.5024840
decoder layer norm value	0.0005925
activation function	ReLU

Table A.2: Hyperparameters for `torch.nn.TransformerEncoder` and `torch.nn.TransformerDecoder` in [32] obtained using Optuna.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-12-18

EXAMENSARBETE Optimizing Soak Test Reviews: A Comparative Study of Deep Learning Architectures
STUDENT Hugo Bläckberg
HANDLEDARE Patrik Edén (LTH)
EXAMINATOR Markus Borg (LTH)

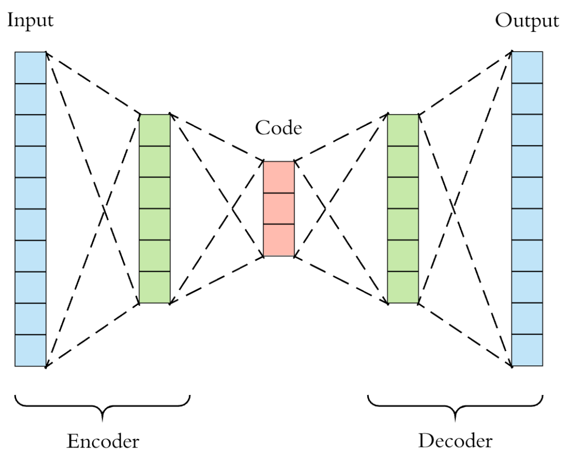
Underlätta analys av resultat från soak-testning med hjälp av maskininlärning

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hugo Bläckberg**

För att säkerställa att mjukvara uppfyller specificerade krav och fungerar som förväntat över en längre tid används soak-testning. Detta arbete har undersökt hur granskning av resultaten från soak-testning kan utföras med maskininlärning för att minska manuellt arbete.

Soak-testning är en metod för mjukvaruutveckling som syftar till att testa den övergripande robustheten hos ett system. Testerna riktar in sig på systemets långsiktiga hälsa genom att köra under en längre tid, till exempel under flera dagar. Detta exponerar eventuella prestandaförsämringar som sker under längre tid än vad typiska belastningstester skulle göra. Dessa försämringar inkluderar t.ex. minnesläckor eller ökad garbage collection. Resultaten från soak-testning kontrolleras idag manuellt, något som tar upp en stor del tid.

I mitt examensarbete har jag undersökt ifall maskininlärning kan användas för att identifiera avvikelser i resultatet från soak-testning, och i bästa fall klassificera resultatet som godkänt eller icke-godkänt. För detta ändamål utforskade jag olika arkitekturer för djupinlärning, mer specifikt LSTM autoencoders, transformers och konvolutionella neurala nätverk. Modellerna använder sig av samma metriker som finns tillgängliga vid manuell kontroll. LSTM autoencodern och transformern klassificerar avvikelser baserat på rekonstruktionsförluster samt ett tröskelvärde. Ett endimensionellt konvolutionellt neuralt nätverk och en transformer-encoder tränades därefter med etiketter från datan. Resultaten visar att alla



modeller presterade bristfälligt, med en låg noggrannhet, förutom det konvolutionella neurala nätverket som fick en balanserad noggrannhet på 95% på test-datan. Det kan finnas flera orsaker till resultatet, något som nämns i rapporten. Bland annat så kan konvolutionella neurala nätverk vara bättre lämpade för multivariata tidsserier eller så förlorades viktiga aspekter i datan under formateringen.