Andreas Wurm, BSc BSc

Predicting the Latency of MQTT Brokers Using Deep Learning

Master's Thesis

to achieve the university degree of
Diplom-Ingenieur
Master's degree program: Information and Computer Engineering

submitted to
Graz University of Technology

Supervisor
Assoc.Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf

Institute of Signal Processing and Speech Communication

Graz, September 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present Master's thesis.

Date	Signature

Acknowledgment

First and foremost I want to thank the Republic of Austria for giving me the chance to pursue my interests in Computer Engineering by providing free access to higher education.

I want to express my sincere thanks to my supervisor Assoc.Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf for his guidance and the excellent support throughout this thesis. I also want to thank Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig and Ing. Dipl.-Ing. Richard Schumi, BSc for their help with the underlying test system and the creation of the datasets used in this work.

Finally, I would like to thank my family and friends who were always there for me, providing help and support whenever I needed it.

Abstract

MQTT is one of the most widely used protocols in the Internet of Things. The performance of this protocol can be tested using statistical model checking integrated into a property-based testing tool. This approach utilizes a cost model to get predictions of the latency of the system-under-test. Multiple linear regression is used for the creation of the cost model at the moment.

In this work we replace the current cost model with deep learning methods. We analyze different datasets from various experiments and broker implementations in order to find the most suitable (recurrent) neural network. We compare different architectures of standard neural networks and gated recurrent units and evaluate the models on datasets created by the test system. We show that the results of the predictions improve significantly compared to the multiple linear regression. In particular, we obtain an R^2 value of 0.9152 compared to a value of 0.8697 with the current cost model, using a dataset created with limited CPU resources. Additionally, the effort of preprocessing, data cleansing and human interactions can be lowered to a minimum. Compared to the mean latencies of the system-under-test, a simulation on the model can achieve a speedup by a factor of up to 500.

Keywords neural network, deep learning, MQTT, Internet of Things, performance, latency, data analysis

Kurzfassung

MQTT ist eines der am weitesten verbreiteten Protokolle im Internet der Dinge. Die Performance dieses Protokolls kann getestet werden, indem Statistical Model Checking in ein Property-based Testing Tool integriert wird. Dieses Testwerkzeug verwendet ein Kostenmodell, um Vorhersagen über die Latenz des zu testenden Systems zu erhalten. Für die Erstellung des Kostenmodells wird derzeit eine multiple lineare Regression verwendet.

In dieser Arbeit ersetzen wir das aktuelle Kostenmodell durch Deep-Learning-Methoden. Wir analysieren unterschiedliche Datensätze aus verschiedenen Experimenten und Broker-Implementierungen, um das am besten geeignete neuronale Netzwerk zu finden. Wir vergleichen verschiedene Architekturen von neuronalen Netzwerken und Gated Recurrent Units und evaluieren die Modelle an Datensätzen, die vom Testsystem erstellt wurden. Wir zeigen, dass sich die Ergebnisse im Vergleich zur linearen Regression signifikant verbessern. Insbesondere erreichen wir einen R^2-Wert von 0.9152 verglichen mit einem Wert von 0.8697 des derzeitigen Kostenmodells, unter Verwendung eines mit begrenzten CPU-Ressourcen erstellten Datensatzes. Darüber hinaus kann der Aufwand für Vorverarbeitung, Datenbereinigung und menschliche Interaktionen auf ein Minimum reduziert werden. Verglichen mit der mittleren Latenzzeit des zu testenden Systems, kann die Simulation am Modell einen Geschwindigkeitsvorteil um einen Faktor von bis zu 500 erreichen.

Schlüsselwörter Neuronales Netzwerk, Deep Learning, MQTT, Internet der Dinge, Performance, Latenzzeit, Datenanalyse

Contents

1 Introduction **1**
1.1 Aim and Contributions . . . 1
1.2 Outline . . . 2

2 Background **3**
2.1 Statistical Model Checking and Testing . . . 3
2.1.1 Statistical Model Checking . . . 3
2.1.2 Property-Based Testing . . . 4
2.1.3 Stochastic Timed Automata . . . 4
2.1.4 Integration SMC into PBT . . . 5
2.2 MQTT . . . 5
2.2.1 Message Types . . . 6
2.3 Regression Analysis . . . 7
2.3.1 Multiple Linear Regression . . . 8
2.3.2 Model Evaluation . . . 8
2.4 Neural Network . . . 9
2.4.1 Neuron . . . 9
2.4.2 Architecture . . . 10
2.4.3 Activation function . . . 11
2.4.4 Training . . . 12
2.4.5 Regularization . . . 13
2.4.6 Model Selection . . . 14
2.5 Recurrent Neural Network . . . 14
2.5.1 Gated Recurrent Unit . . . 14
2.5.2 Training . . . 15

3 Cost Model Learning **17**
3.1 Test System . . . 17
3.1.1 Cost Model . . . 18
3.2 Log-Data Generation . . . 18
3.2.1 Implementations . . . 19
3.2.2 Datasets . . . 19
3.3 Features . . . 20
3.4 Analysis and Statistics . . . 21
3.4.1 Default Dataset . . . 21
3.4.2 Limited CPU Dataset . . . 25
3.4.3 Summary . . . 29
3.5 Preprocessing . . . 32
3.5.1 One-Hot Encoding . . . 32
3.5.2 Feature Scaling . . . 33
3.5.3 Target Scaling . . . 33
3.6 Learning . . . 33
3.6.1 Multiple Linear Regression . . . 33
3.6.2 (Recurrent) Neural Network . . . 34
3.7 Serving . . . 34

4 Experimental Results **37**
4.1 Default Dataset 37
4.1.1 Discussion 40
4.2 Limited CPU Dataset 41
4.2.1 Discussion 44
4.3 Summary 45

5 Conclusion **47**
5.1 Outlook 47

Bibliography **49**

1 Introduction

The Internet of Things (IoT) is growing in popularity. More and more small and passive devices, like sensors etc., are communicating over networks with each other. One way to establish a message exchange between those devices, without producing too much overhead and using too many valuable resources (like energy etc.), is the Message Queue Telemetry Transport (MQTT) protocol. It allows machine-to-machine communication with the use of a publisher-subscriber pattern. The broker, a central server in MQTT, is a key element in this infrastructure. It can be chosen from a wide range of different broker implementations. But how does one know if the chosen implementation is capable of fulfilling all the needs? It is not feasible to try out different systems in a real-life scenario. The performances of all eligible brokers have to be tested and compared with each other. To simplify this process of performance testing and comparison, Aichernig et al. [1] developed a new method of automatically testing the performance of different implementations, using Statistical Model Checking (SMC) integrated into a Property-Based Testing (PBT) tool and used it to check the response-times of a web application [2]. Their test system is capable of answering quantitative and qualitative questions and is therefore essential to identify the best performing software implementation. It allows to predict the performance on a model, rather than the real system. Then, the predictions are verified on the system-under-test (SUT), using a hypothesis test.

Aichernig and Schumi adopted this testing method for the specific use case of MQTT broker performance testing [3]. The system is capable of comparing multiple MQTT implementations in terms of latencies. It answers questions, like "What is the probability that the latency is below a certain threshold?". A substantial part of the test system is the cost model, which is used to estimate the costs of the SUT. Because this work is targeting the performance of different MQTT broker implementations, the costs we are interested in are the latencies. The latency is a measure of the time of an operation executed on the broker, from the request being sent, to the response being received. This work focuses on the cost model of the test system. In the test system a simple multiple linear regression is being used to create a cost model, which turned out to be not very accurate. Furthermore, using a linear regression as cost model takes a lot of effort in feature selection and preprocessing. This has to be done manually by analyzing coefficient matrices and other statistical parameters. The question is, if the whole process of cost model learning can be simplified and at the same time increase the quality of the prediction.

1.1 Aim and Contributions

The aim of this work is to improve the predictive accuracy of the cost model and to simplify the process of cost model learning, i.e. to keep the preprocessing at a minimum. Our deep learning cost model is capable of increasing the predictive accuracy of the currently used

multiple linear regression, which serves as a baseline for comparison. It is possible to achieve an R^2 value of over 0.9 without feature selection. We show that the processes of preprocessing, learning and predicting can be completely integrated into the test system.

The log-data, created in the test system, is used to select the neural network model. The datasets consists of 21 attributes and up to 300,000 samples which are carefully analyzed to find out what kind of machine learning technique should be used. To keep the process as simple as possible, we try to avoid as many manual preprocessing steps as possible. We leverage state-of-the-art methods in deep learning and take advantage of neural networks as well as recurrent neural networks to create a cost model which is capable of predicting the latency of an MQTT broker.

By now, there is no comparable test system available, which uses model-based testing combined with deep learning methods. Current work in the area of MQTT performance testing mainly focuses on different settings of the broker. To the best of our knowledge, there is no work that compares the performance of multiple broker implementations with each other. Our test system can speedup performance testing by a factor of up to 500 compared to mean latency on the SUT, which is a tremendous improvement. The method developed in this work can also be applied to compare various of other costs (like energy etc.) and the applications are therefore not limited to performance testing.

1.2 Outline

Chapter 2 explains the theoretical background and related work substantial for this thesis. The chapter starts with an overview of statistical model checking and property-based testing as well as how to integrate SMC into PBT to answer quantitative and qualitative questions. Then the MQTT protocol is described in detail. Afterwards an introduction to regression analysis is given before the main deep learning methods of this thesis, neural networks and recurrent neural networks, are explained. Chapter 3 introduces the test system and the cost model. It shows the generation of the log-data and the different test cases used in this thesis. The features and a statistical analysis of the data is done afterwards. The steps of preprocessing, learning and serving of the cost model are described and the neural network architectures used in this thesis are presented. Chapter 4 shows the experimental results and provides a detailed discussion. It shows the limitations of this method and provides a deep insight before Chapter 5 concludes this work and gives an outlook on future work.

2 Background

The following chapter presents the theoretical background and the current state-of-the-art techniques used in this work. Section 2.1 starts with statistical model checking and property-based testing, methods used in our test system around the cost model. Section 2.2 gives an overview of the MQTT protocol and its message types. Section 2.3 provides an introduction to regression analysis, multiple linear regression and the model evaluation. Afterwards, Sections 2.4 and 2.5 introduce the machine learning methods used in this thesis. They cover the structure of a single neuron as well as complex architectures and networks. Training of neural networks and methods to regularize them are shortly discussed.

2.1 Statistical Model Checking and Testing

This section is based on [2] and [3] as it is the underlying work of this thesis. For more detailed information the interested reader is referred to this work. The test system is described in Section 3.1.

2.1.1 Statistical Model Checking

Statistical Model Checking (SMC) [4] is a software simulation method which is capable of answering qualitative as well as quantitative questions of a stochastic model, formulated as properties. For example, SMC is able to answer questions like "What is the probability, the latency is below a certain threshold?". SMC creates samples using random walks on the model. Those samples are evaluated and verified to check if the property holds. The test system (see Section 3.1) uses a standard *Monte Carlo simulation*. The algorithm estimates the probability γ, that the property is satisfied. Multiple algorithms can be used to compute the number of samples n needed to get an answer with a certain probability. The test system uses the *Chernoff-Hoeffding bound* to calculate n using a lower limit for the estimation error ϵ. Sampling can be stopped when the stopping criterion, e.g. the number of simulations n, is reached. With the confidence $1 - \delta$, n is determined as

$$n \geq \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right). \tag{2.1}$$

The n Monte Carlo simulations produce the random variables $X_1, \ldots, X_n$, which result in $x_i = 0$ if the property does not hold or $x_i = 1$ if it holds. Then, the estimated probability is

$$\bar{\gamma}_n = \frac{\sum_i^n x_i}{n} \tag{2.2}$$

and the probability of the estimation error being below ϵ, is greater than the confidence:

$$\Pr(|\bar{\gamma}_n - \gamma| \leq \epsilon) \geq 1 - \delta. \tag{2.3}$$

Another algorithm used is the *Sequential Probability Ratio Test* (SPRT), which is a form of hypothesis testing [5]. It can be used to answer qualitative question, by deciding between a null hypothesis $H_0 : \theta = \theta_0$ or an alternative hypothesis $H_1 : \theta = \theta_1$. Sampling is done while $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$ with

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \sum_i^m \log \frac{f(x_i, \theta_1)}{f(x_i, \theta_0)}. \tag{2.4}$$

The variables α and β are the desired type I and type II errors. We accept H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ and H_1 when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$.

In the test system, SPRT is used to verify if a hypothesis, formed on the model, also holds on the SUT. Performance evaluation of software implementations is computationally very expensive. Using a model-based approach, and therefore running the Monte Carlo simulation on a model instead of the SUT directly, increases the speed of the process.

2.1.2 Property-Based Testing

Property-Based Testing (PBT) is a random-testing method. PBT automatically creates random inputs for the SUT and checks if the behavior of the SUT is as expected. The random inputs are created using data generators. A counterexample is formed if the SUT does not work as expected [6]. There are numerous PBT implementations, with QuickCheck being the first one [7].

PBT supports model-based testing, with models in the form of an extended finite state machine (EFSM) [8]. An EFSM consists of a finite set of states S, the initial state s_0, a finite set of variables V, a finite set of inputs I and outputs O, and a finite set of transitions T. It can be written as a tuple (S, s_0, V, I, O, T). A transition t is a tuple (s_s, i, g, op, o, s_t), where s_s is the source state and s_t the target state. The variables i and o are the input and output of the transition. The guard g is the predicate and therefore an expression that decides whether the transition is enabled or not. If the expression is always $True$, the transition does not have a guard. The operation op is the update function, which calculates new values for the variables.

In the test system, we execute a command on the SUT and one on the model, using the same input parameters. Valid input parameters are defined in the precondition. After execution of the command, the outputs and states of the SUT and the model are compared with each other in the postcondition.

2.1.3 Stochastic Timed Automata

Stochastic timed automata (STA) [9] are an extension of timed automata (TA) [10], which are used to simulate realistic timing behavior of computer systems [2]. The TA tuple (L, l_0, A, C, I, E) is extended by a probability density function (PDF) F for the sojourn and natural weights $W = (w_e)_{e \in E}$ to the STA tuple $(L, l_0, A, C, I, E, F, W)$. The remaining elements of the tuple is the finite set of locations L with the initial location l_0, a finite set of actions A, a finite set of clocks C, a finite set of invariants I and a finite set of transitions

(or edges) E. For a state (l, u) with the location $l \in L$ and the clock valuation $u \in C$, the sojourn time d is chosen with the use of the probability function f_l. This changes the state to $(l, u + d)$, where $u + d$ denotes the clock valuation change of $(u + d)(c) = u(c) + d$ for all $c \in C$. Then, an enabled edge e is selected from $E(l, u + d)$ and the probability of $w_e / \sum_{h \in E(l,u+d)} w_h$. The transition to the target location l' of e and $u' = u + d$ is performed afterwards.

In the test system, STA is used to add realistic timing behavior to the predictions of the model. We use a *semi-Markov process* as underlying stochastic process, because the clocks are reset at every transition, but exponential delays are not assumed.

2.1.4 Integration SMC into PBT

Aichernig and Schumi demonstrated that SMC can be integrated into a PBT tool [1, 11]. Which makes it possible to perform SMC of PBT properties and can return a qualitative or quantitative result, depending on the use case. The PBT tool is used as simulation environment to execute the SMC algorithm. In the test system, the integration is used for an performance evaluation with the model and for testing the SUT.

2.2 MQTT

MQTT (Message Queue Telemetry Transport) is a messaging protocol widely used in many modern IoT and machine-to-machine applications. MQTT is located in the application layer of the TCP/IP stack. The latest protocol version is 3.1.1 and was released in December 2015 [12]. The protocol consists of one central server, referred to as broker, and multiple clients. It follows a publisher-subscriber pattern. Topics, clients can subscribe or publish to, are structured hierarchically. The hierarchical layers are being separated by a forward slash (e.g. *sensor/temperature/area1* with *sensor* being the top and *area1* the bottom of the hierarchy). Clients can be both, publisher or subscriber. It is also possible to use wild-cards to subscribe to multiple topics at the same time (e.g. *sensor/+/area1* subscribes to all sensors within *area1*). The different message types (control messages) used in this protocol to establish a connection and to communicate are: `connect`, `connack`, `publish`, `puback`, `pubrec`, `pubrel`, `pubcomp`, `subscribe`, `suback`, `unsubscribe`, `unsuback`, `pingreq`, `pingresp`, and `disconnect` and will be further described in Section 2.2.1.

When publishing a message to a topic, the broker receives the message and distributes it to all the clients subscribed to the same topic. Figure 2.1 gives an overview of the publisher-subscriber pattern used in MQTT. The clients *Device 1* and *Device 2* are both subscribed to a topic, the client *Sensor* is publishing to and therefore are both receiving the messages in this topic. The protocol uses a 2-byte header with fields indicating the message type, the QoS level, the retain flag as well as the length of the payload. The retain flag can be set to keep the latest message on the broker, which will be delivered immediately to all newly subscribing receivers. Only the newest message for the corresponding topic, with the retain flag set to true, will be kept by the broker. Other MQTT features like last will and testament (LWT) or keep alive, which involve the message types pingreq and pingresp are not covered, because they are not relevant for this thesis.

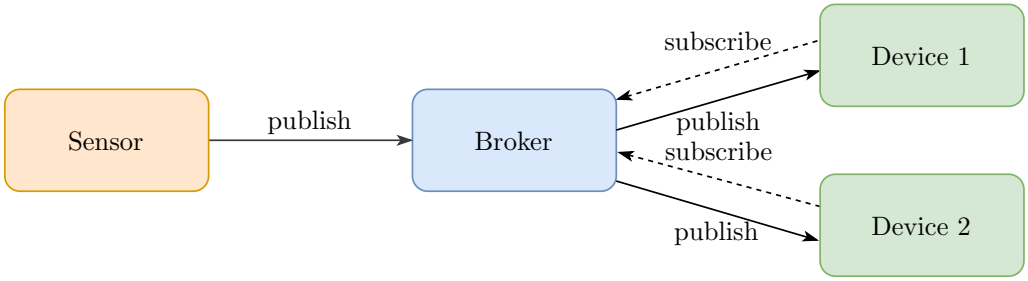


Figure 2.1: Overview of the MQTT publisher-subscriber pattern.

Quality of Service The protocol supports three levels of Quality of Service (QoS): At most once (QoS 0), at least once (QoS 1) and exactly once (QoS 2). The level *QoS 0* delivers the message only one time and does not need an acknowledge from the receiving clients. *QoS 1* messages will be delivered at least once but it could also happen that they get delivered more than one time. The level *QoS 2* indicates that the message will be received exactly once. To ensure the QoS, the pubrec, pubrel and pubcomp control messages are sent after the publish, depending on the level of service.

2.2.1 Message Types

This section covers only the message types that are relevant for this thesis. Other control messages which for example would require QoS 2 are not used in our experiments.

Connect The message type `connect` is the first message that has to be sent after a client has initiated a TCP/IP connection with the broker. If the connection was successfully established, the server answers with a `connack` message. The `connect` message has several options, like *username*, *password*, *clientId* or a *cleanSession* flag. The *cleanSession* flag indicates that the client wants to use a persistent session, i.e. the broker stores all messages missed by the client and delivers them after a reconnect if the flag is set to false. This flag is also dependent on the QoS level and therefore only relevant for QoS 1 or 2. If the flag is set to false and the client had a clean session before, the broker creates a new session for that client.

Disconnect Disconnect is the counterpart to the `connect` message. It is sent to the broker to end an existing connection.

Subscribe The control message `subscribe` is necessary to subscribe to topics and therefore to receive messages, published by other clients. The message contains a list of topics and the corresponding QoS levels. A successful subscription will be acknowledged by the broker with a `suback` message.

Unsubscribe The unsubscribe message is used to remove existing subscriptions. The payload contains a list of topics the clients wants to unsubscribe from. If the request was successful, the broker acknowledges it with a `unsuback` message.

Publish The publish message is used to send messages. It contains the fields *topicName*, *QoS*, *retainFlag*, and the payload.

2.3 Regression Analysis

Regression analysis is a statistical method, used to describe relations between one or more independent variables $\boldsymbol{X}$ to the dependent variable $\boldsymbol{y}$, also referred to as *target* variable. This analysis can be used to make predictions or forecasts of target values based on various input features (or a history of them). The simplest form of the regression analysis is the *linear regression*. It tries to fit a line or hyperplane through data, while keeping the occurring error as small as possible [13]. Regression analysis is a form of supervised learning, which means that, unlike unsupervised learning, the real target values are known in the training phase. The regression consists of the following variables and parameters:

$\boldsymbol{X}$ is an $n \times m$ matrix, consisting of n samples and m independent variables (the features). A single sample is denoted as $\boldsymbol{x}_i = [x_{i,1}, x_{i,2}, \cdots, x_{i,m}]^T$.

$\boldsymbol{y} = [y_1, y_2, \cdots, y_n]^T$ is the dependent variable of size n (we assume a single target variable).

$\boldsymbol{\beta} = [\beta_0, \beta_1, \cdots, \beta_n]^T$ of size n are the regression parameters.

Note, that for the sequel, bold and uppercase letters such as $\boldsymbol{X}$ denote matrices and the transpose is denoted by a superscript T. Lowercase bold letters such as $\boldsymbol{y}$ denote vectors and lowercase non-bold letters such as m denote scalars.

The regression model $f(\cdot)$ tries to explain the dependent variable with the independent variables, the regression parameters and the error term $\boldsymbol{\epsilon}$, i.e.

$$\boldsymbol{y} = f(\boldsymbol{X}, \boldsymbol{\beta}) + \boldsymbol{\epsilon}. \tag{2.5}$$

Residual The residual $\boldsymbol{e}$ is the difference between the observable true value $\boldsymbol{y}$ and the estimated value $\hat{\boldsymbol{y}}$ by the regression model: $\boldsymbol{e} = \boldsymbol{y} - \hat{\boldsymbol{y}}$

Error The error term $\boldsymbol{\epsilon}$ on the other hand is a random variable which is the unobservable change of the (unobservable) true value. This could be for example errors in measurement, noise etc.

RSS The residual sum of squares (RSS) is the sum of the squared residuals, i.e.

$$\text{RSS} = \sum_i^n (y_i - f(\boldsymbol{x}_i))^2 = \sum_i^n (y_i - \hat{y}_i)^2 = \sum_i^n e_i^2. \tag{2.6}$$

TSS The total sum of squares (TSS) is the sum of the squared difference of the target variable and its mean value $\bar{y}$, i.e.

$$\text{TSS} = \sum_i^n (y_i - \bar{y})^2. \tag{2.7}$$

2.3.1 Multiple Linear Regression

To explain the dependent variable $\boldsymbol{y}$ with multiple independent variables $\boldsymbol{X}$, multiple linear regression [14] is used. It is defined as

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_m x_{i,m} + \epsilon_i \tag{2.8}$$

and when using all data samples[1] as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}. \tag{2.9}$$

Ordinary Least Squares

One method of estimating the unknown regression parameters $\boldsymbol{\beta}$ of the multiple linear regression is to use ordinary least squares (OLS). To get an estimation of the unknown regression parameters, OLS minimizes the residual sum of squares. Ignoring the error term, it can be written as

$$\arg\min_{\beta} \mathrm{RSS} = \arg\min_{\beta} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) = \arg\min_{\beta} \boldsymbol{e}^2. \tag{2.10}$$

We obtain the regression parameters by using the Moore-Penrose inverse as

$$\boldsymbol{\beta} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}. \tag{2.11}$$

2.3.2 Model Evaluation

To evaluate and compare results of different models, metrics like R^2, mean squared error (MSE) or rooted mean squared error (RMSE) can be used. We introduce these metrics in the sequel.

$\mathbf{R^2}$ Also referred to as coefficient of determination, R^2 is a measure for the goodness of a fit. A value of 1 denotes a perfect fit, while a value of 0 indicates a bad fit and occurs when the model always estimates the mean of the dependent variable. It is also possible for values to be negative, when the data is fitted worse than with a mean estimator. It is computed as

$$\mathrm{R}^2 = 1 - \frac{\mathrm{RSS}}{\mathrm{TSS}} = 1 - \frac{\sum_i^n (y_i - \hat{y}_i)^2}{\sum_i^n (y_i - \bar{y})^2} = 1 - \frac{\sum_i^n e_i^2}{\mathrm{var}\{y\}^2} \ . \tag{2.12}$$

MSE The mean squared error is the average of the squared sum of the residuals. A value of 0 indicates a perfect fit of the model. There is no upper boundary. The MSE is determined as

$$\mathrm{MSE} = \frac{1}{n}\mathrm{RSS} = \frac{1}{n}\sum_i^n (y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_i^n e_i^2 \ . \tag{2.13}$$

[1] Note that for $\boldsymbol{X}$, a column of ones is added to include the bias.

RMSE The rooted mean squared error is the square root of the MSE, i.e.

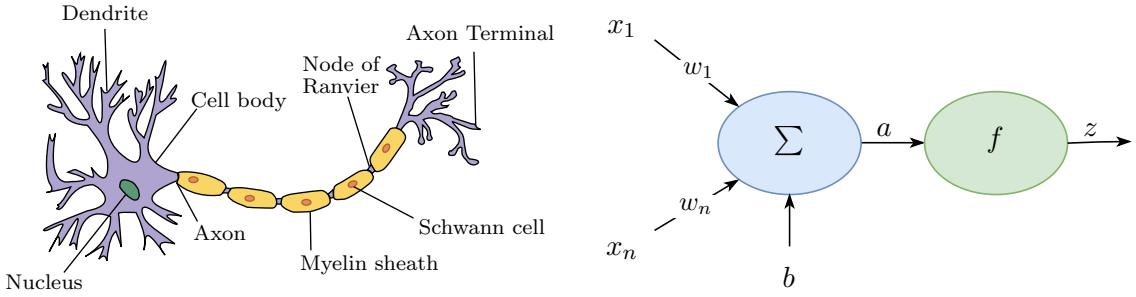
$$\text{RMSE} = \sqrt{\text{MSE}}. \tag{2.14}$$

2.4 Neural Network

Artificial neural networks (ANN) were inspired by biology, where multiple individual neurons together form a complex network. This network, better known as the brain, is capable of learning and processing things [15, 16]. The multilayer perceptron (MLP) is one type of artificial neural networks. It consists of neurons arranged in multiple layers with connections between them. Without recurrent connections between the neurons, it is also called feed forward network [17]. Neural networks are used in a wide field of applications, for classification or prediction tasks. They are used for signal processing, speech recognition, image classification, self-driving cars, image optimization, etc. [18–20]. *Deep learning* denotes the use of a neural network with multiple layers which are capable of processing complex data [21–23].

2.4.1 Neuron

An artificial neuron is the smallest building block of a neural network. Its principles are based on the biological neuron, which is shown in Figure 2.2a. The dendrite therefore corresponds to the input of the neuron, the axon to the output and the cell body forms the activation function. A connection between two biological neurons is referred to as synapse.



(a) biological neuron (adopted from [24]) *(b) artificial neuron*

Figure 2.2: Structure of an artifical neuron compared to a biological neuron.

The artificial neuron on the other hand is shown in Figure 2.2b. It consists of one or more inputs x_i and a corresponding weight w_i for each of it, a bias b and an *activation function* (also called transfer function) $f(\cdot)$. An artificial neuron can be mathematically described as

$$a = b + \sum_{i}^{m} x_i w_i = \boldsymbol{w}^T \boldsymbol{x}, \tag{2.15}$$

where

$$\boldsymbol{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad \text{and} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}. \tag{2.16}$$

After applying the activation function $f(\cdot)$, we obtain the output (or activation) z of the neuron:

$$z = f(a) = f(\boldsymbol{w}^T \boldsymbol{x}) \tag{2.17}$$

2.4.2 Architecture

Multiple neurons connected together, form a neural network. Those connections can also be structured in layers to form an MLP:

Input Layer is the first layer of a neural network and processes the input of the network.

Hidden Layers are the layers between input and output layers. A neural network can consist of multiple hidden layers, which are increasing the, so called, depth of the network and therefore also its complexity.

Output Layer is the last layer of a neural network and provides the output of the network. For classification tasks, this layer usually has multiple neurons, while in prediction and forecasting it consists of only one neuron with linear activation function.

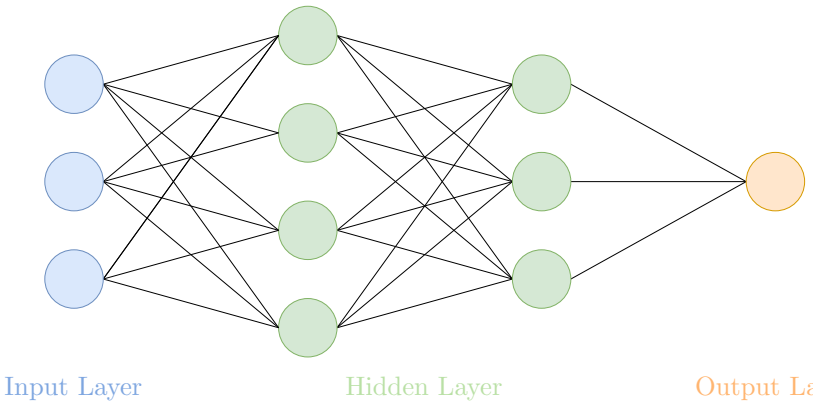


Figure 2.3: Architecture of a neural network, structured in input layer, hidden layers and output layer.

Figure 2.3 shows a neural network with three inputs, two hidden layers with 4 and 3 neurons and an output neuron. This architecture can also be denoted as 4-3-1.

2.4.3 Activation function

To establish non-linearity in the network, a (non-linear) activation function is used. Depending on the goal of the task, various activation functions can be used. Activation functions are monotonic functions and have to be derivable so they can be used in back-propagation (see Section 2.4.4). In the following, we introduce three widely used activation functions.

Sigmoid The sigmoid activation function, commonly known as the logistic activation function, is characterized by the S-shape and shown in Figure 2.4a. It can range from 0 to 1 and is defined as

$$f(a) = \sigma(a) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2.18}$$

and the derivative is defined as

$$f'(a) = \sigma(a)\left(1 - \sigma(a)\right). \tag{2.19}$$

TanH The tanH activation function is the hyperbolic tangent function. It can range from -1 to 1. Like the logistic function, the tanH function is a sigmoid function too. Figure 2.4b shows the hyperbolic tangent function. The activation function is defined as

$$f(a) = \tanh(a) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.20}$$

and the derivative of this function is defined as

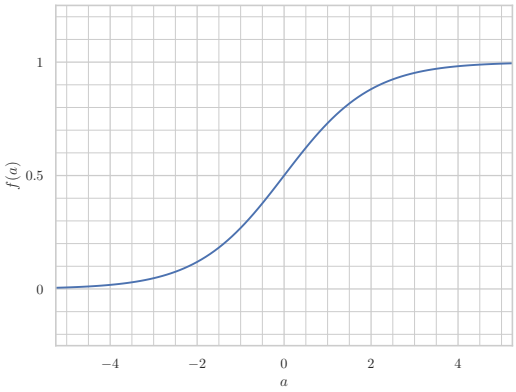
$$f'(a) = 1 - \tanh^2(a). \tag{2.21}$$

ReLU The ReLU is the rectified linear unit activation function. It can range from 0 to infinity. The ReLU function leads to better results than comparable activation functions [25] and is widely used in modern applications [26]. There are various extensions to this activation function. Figure 2.4c shows the ReLU activation function, which is defined as

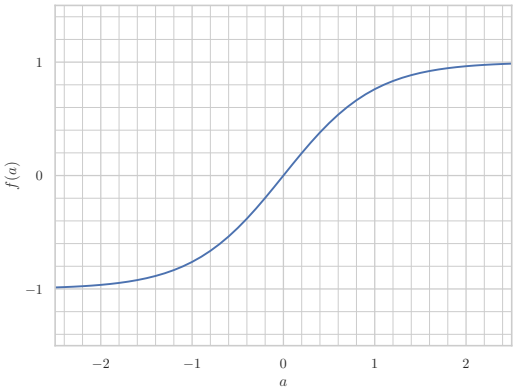
$$f(a) = \max(0, a) = \begin{cases} 0 & \text{for } a < 0, \\ a & \text{for } a \geq 0, \end{cases} \tag{2.22}$$

and the derivative of the ReLU function is defined as

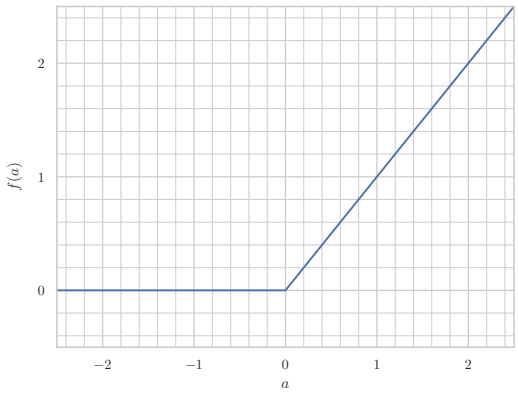
$$f'(a) = \begin{cases} 0 & \text{for } a < 0, \\ 1 & \text{for } a \geq 0. \end{cases} \tag{2.23}$$



(a) Sigmoid (logistic) activation function



(b) TanH activation function



(c) ReLU activation function

Figure 2.4: Comparison of three widely used activation functions.

2.4.4 Training

Training is the process of learning the unknown parameters of a neural network. One method to train a neural network, is backpropagation (also referred to as backprop), which is a supervised learning method. The goal of the training is to find the parameters that minimize the loss function J (also referred to as error or cost function). Minimizing the loss function cannot be solved analytically and therefore an optimization algorithm has to be applied. The *gradient descent* algorithm is used for optimization, to find the minimum of a function. However, it is not guaranteed that the global minimum is found and not just a local minimum. As cost function we use the MSE (defined in (2.13)), but also other functions like the mean absolute error (MAE) or the cross entropy can be used.

Backpropagation The backpropagation algorithm is split into a forward step and a backward step. The forward step processes the input through the network and calculates the output, similar to the prediction. The backward step on the other hand, calculates the error by comparing the output of the network to the target value, propagates it back and updates the parameters of the network. The MSE loss function J we are using, is

multiplied by a factor $\frac{1}{2}$ to simplify the future derivation:

$$J_{\text{MSE}} = \frac{1}{2n}\sum_{i}^{n}(z_i - y_i)^2 = \frac{1}{2n}\sum_{i}^{n} e_i^2. \tag{2.24}$$

The derivative of the squared error function J, with w_{jk} being the weight between the neurons j and k, is:

$$\frac{\partial J}{\partial w_{jk}} = \frac{\partial J}{\partial z_k}\,\frac{\partial z_k}{\partial a_k}\,\frac{\partial a_k}{\partial w_{jk}} = \delta_k z_j, \tag{2.25}$$

where

$$\delta_k = \begin{cases} \frac{\partial f_k(a_k)}{\partial a_k}\; e_k & \text{if k is an output neuron,} \\ \frac{\partial f_k(a_k)}{\partial a_k}\; \sum_l \delta_l w_{kl} & \text{if k is a hidden neuron.} \end{cases} \tag{2.26}$$

Then, gradient descent is applied to update the weight w_{jk}. With the learning rate η, the change of the weight is:

$$\Delta w_{jk} = -\eta \delta_k z_j. \tag{2.27}$$

The standard gradient descent update step, can also be extended with other parameters as in the *ADAM* (Adaptive Moment Estimation) optimizer [27]. When training the network, the input can be fed into the network in different ways:

Batch learning uses the entire dataset in one batch.

Mini batch is the dataset split into multiple smaller batches with the same size.

Stochastic Gradient Descent (SGD) or online learning uses a batch size of one.

2.4.5 Regularization

Overfitting occurs if a neural network memorizes the training set or parts of it. It is characterized by a low training error but a high test error. The opposite is called underfitting and is characterized by high training and test errors. Underfitting mostly occurs if the network is too shallow or the chosen architecture is too simple. Overfitting can be prevented by the use of various regularization techniques like:

Early Stopping stops the learning process when the error on the validation set increases more than a certain threshold.

Dropout randomly stops using various neurons of the network, so other neurons have to increase their ability in generalization [28].

L_1/L_2 Regularization (also referred to as weight decay) adds e.g. L_2 norm of the weights as an additional term to the objective function, so that large weights are penalized:

$$J = \text{MSE} + \lambda ||\boldsymbol{w}||_2. \tag{2.28}$$

2.4.6 Model Selection

In order to choose the best model for the machine learning task, the entire dataset is split into a training set, test set and validation set. The different models are trained using the training set. Afterwards the validation set is used to compare the trained models with each other and the model with the lowest error is chosen. Then, the test set is used to obtain the final performance of the network.

2.5 Recurrent Neural Network

A variant of the neural network which is capable of *memorizing* the past of a sequence is the recurrent neural network (RNN) [23]. A recurrent cell gets its output fed back as input and usually consists of multiple gates that decide how much of the (new) input should be kept, how much of the (old) internal cell state should be preserved and what should be replaced by new information. RNNs are very useful for sequential tasks like speech recognition [29] or language translation [30, 31].

A simple recurrent neural network with the output vector $\boldsymbol{h}_t$ and the activation function σ can be described as:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{b}). \tag{2.29}$$

The $h \times h$ matrix $\boldsymbol{U}$ and the $h \times m$ matrix $\boldsymbol{W}$ are the weights and $\boldsymbol{b}$ is the bias vector of size h. The number of features in the input is denoted with m and the number of recurrent units is denoted with h. Figure 2.5 shows the simplified structure of an RNN cell, before and after unrolling.

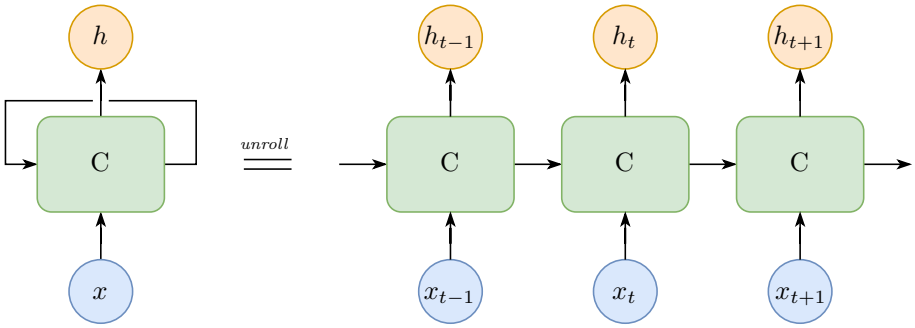


Figure 2.5: Schematic representation of a recurrent neural network and its unrolled version.

2.5.1 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) [32] is one type of RNN and aims to solve the *vanishing gradient problem*, which is commonly present in recurrent neural networks [33] and describes vanishingly small gradients, that prevent the weights from being updated. It is similar to a long short-term memory (LSTM) cell, introduced by Hochreiter and Schmidhuber [34], but uses fewer parameters and no output gate. The performance of GRUs is slightly better than the performance of LSTMs [35]. It consists of a reset gate and an update gate. As suggested by Kyunghyun et al. the activation functions are a sigmoid

function, denoted as σ, and the hyperbolic tangent. The operation $\odot$ is the Hadamard, or entrywise, product. Update gate vector $\boldsymbol{z}_t$ and reset gate vector $\boldsymbol{r}_t$ decide how much of the past information from previous times steps should be kept, and what should be updated, i.e.

$$\boldsymbol{z}_t = \sigma\left(\boldsymbol{W}_z \boldsymbol{x}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1} + \boldsymbol{b}_z\right), \tag{2.30}$$

$$\boldsymbol{r}_t = \sigma\left(\boldsymbol{W}_r \boldsymbol{x}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r\right), \tag{2.31}$$

and

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W} \boldsymbol{x}_t + \boldsymbol{U}\left(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}\right) + \boldsymbol{b}\right). \tag{2.32}$$

The output vector is computed as:

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \tilde{\boldsymbol{h}}_t + \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1}. \tag{2.33}$$

Figure 2.6 shows the inner structure of a GRU with its reset and update gates.

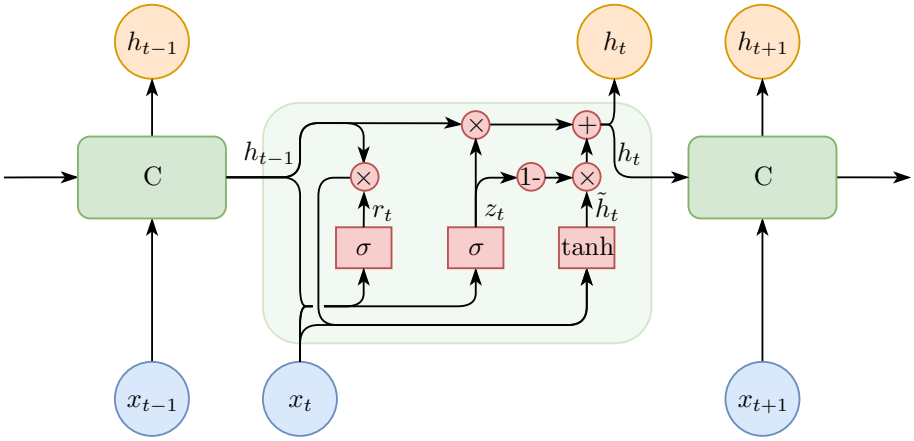


Figure 2.6: Simplified representation of a GRU (inspired by [36]).

2.5.2 Training

Similar to an MLP, a recurrent neural network like the GRU is trained using backpropagation. For RNNs, the process is called *backpropagation through time* and essentially, the network is unrolled (unfolded) and transformed to a standard feed forward network [37].

3
Cost Model Learning

This chapter gives an overview of the test system developed by Aichernig and Schumi [3] in Section 3.1 and shows how the cost model is embedded in it. Next, Section 3.2 describes the generation of the log-data and Section 3.3 presents the features of the datasets. Afterwards, Section 3.4 statistically analyzes the different datasets and compares them with each other. We show all the necessary preprocessing steps in Section 3.5 as well as a detailed description of the learning techniques we use in Section 3.6. Then, Section 3.7 shows how the trained cost model is integrated into the test system in a process called *serving*.

3.1 Test System

Aichernig and Schumi developed and implemented[2] a test system based on the integration of SMC into PBT. This test system is described in detail in the publication "How Fast is MQTT? Statistical Model Checking and Testing of IoT Protocols" [3]. The publication further exploits the results developed within TRUCONF[3] (trust via cost function driven model based test case generation for non-functional properties of systems of systems), but is a publication of the Dependable Things project[4]. In this test system, log-files are generated while testing an MQTT broker (the reference SUT) using model-based testing techniques. During testing, the SUT is treated as a black box. Multiple threads with clients are running concurrently and are sending messages to the broker. The latencies of the messages are recorded in log-files, which are then used to learn a cost model. Originally a multiple linear regression is used to create the model. Combined with user profiles, the cost model is afterwards being used to create a timed model (STA, see Section 2.1.3). Later, Monte Carlo simulations of the model are being conducted in virtual time i.e. a fraction of real time. The predicted latencies of the models are compared with the real time latencies of the SUT in a hypothesis test (SPRT, see Section 2.1.1) and the test system decides which hypothesis should be accepted and which one should be refused. Figure 3.1 gives an overview of the test system by Aichernig and Schumi.

The test system was implemented in C# and using Visual Studio 2012 with the .NET framework 4.5, NUnit 2.64 and FsCheck 2.92.

[2] `https://github.com/schumi42/mqttCheck`
[3] `http://truconf.ist.tugraz.at`
[4] `https://www.tugraz.at/projekte/dependablethings`

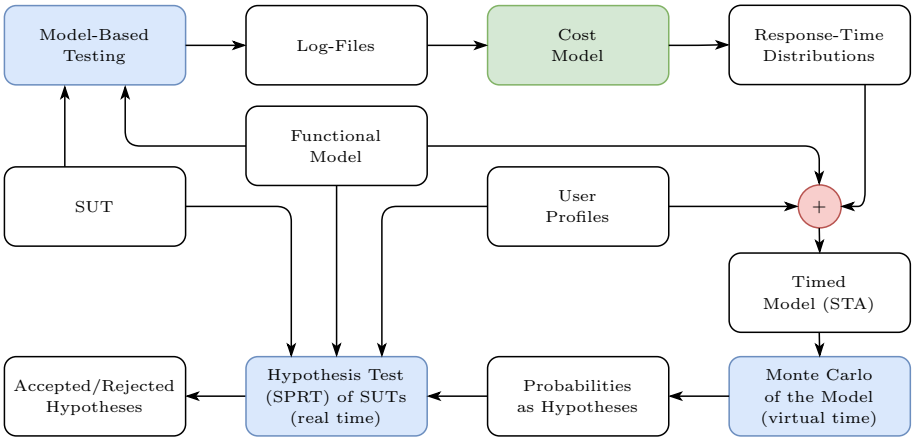


Figure 3.1: Overview of the test system by Aichernig and Schumi (based on [38]).

3.1.1 Cost Model

As seen in the overview of the test system (Figure 3.1), the green box represents the cost model we are going to develop and improve in this thesis. Originally, Aichernig and Schumi used a multiple linear regression to create a cost model. This work replaces the linear regression with deep learning methods, in order to achieve better results while saving more effort in preprocessing.

3.2 Log-Data Generation

As described in Section 3.1, the data we are using in training is log data, generated while randomly testing the SUT using model-based testing. In this work, the performance of multiple MQTT broker implementations and usage scenarios will be evaluated. The generated datasets are sorted by time and split into comma separated values (CSV) files per *Client ID*. The data is generated using approximately equally distributed message types. The test system is running on a Microsoft Windows server (2008 R2) with 2.1 GHz Intel Xeon E5-2620 v4 CPU, 8 cores and memory of 32 GB. In our experiments, the machine runs the clients as well as the broker. Therefore it is likely that the processes are influenced by each other. However, there is no disturbance because of network latencies in this setup. For some experiments, the broker is executed in a virtual machine using Oracle VM Virtual Box (see Section 3.2.2). Table 3.1 shows a simplified excerpt of the log data. Note that some feature names are left out or shortened to make it more readable. All features of the dataset are listed and described in detail in Section 3.3.

ActiveReq	TotalSubs	Msg	TopicSize	MsgSize	Subs	Receivers	Latency
18	754	connect	0	0	0	0	106.36
18	737	subscribe	14	0	0	0	98.88
18	700	disconnect	0	0	0	0	0.19
19	733	publish	14	76	2	2	228.11
18	742	unsubscribe	14	0	0	0	78.46

Table 3.1: Simplified example of the generated log-data. Detailed feature names are presented in Section 3.3. In particular ActiveReq and TotalSubs correspond to #ActiveRequests and #TotalSubscriptions. Subs and Receivers correspond to #Subscribers and #PublishReceiver.

3.2.1 Implementations

There are several implementations of MQTT brokers. This work evaluates the performances of three of the most widely used implementations for its experiments: *EMQ*, *Mosquitto* and *VerneMQ*. For all experiments, the default settings of the brokers are used. As MQTT client library, Aichernig and Schumi use M2Mqtt[5].

EMQ is the Erlang MQTT Broker, also referred to as eMQTT It is an open source MQTT broker written in Erlang. In the experiments, version 2.3.5 of the broker is used.

Eclipse Mosquitto is an open source MQTT broker written in Python and C [39]. In the experiments, version 1.4.15 of the broker is used.

VerneMQ is a broker developed by Octavio Labs in Erlang/OTP. The lastest release is version 1.4.1, which is used in this thesis.

3.2.2 Datasets

We identified multiple usage scenarios and therefore run a range of different experiments which led to two major groups of datasets, which are subject of this thesis:

Default The dataset originally used by Aichernig and Schumi in [3]. It consists of 100 test cases, each with a length of 50 messages and 3 – 100 active clients. The dataset has a length of approximately 300,000 samples.

IoT Environment This dataset is similar to the default one, except that we run the broker in a Virtual Box and limit its CPU resources (in percentage of the host machine's CPU). We used this setting to create a more realistic IoT environment, which often operate on very small, energy efficient devices and do not have a lot of computing power. For this work we experimented with limitations of 10%, 5% and 1% of the CPU resources.

[5] `https://m2mqtt.wordpress.com`

Other experiments, like increasing size of messages or topics, or experiments with many passive clients (clients which only subscribe to topics, but do not publish themselves) are not part of this thesis, but might be covered in future work.

3.3 Features

The generated datasets consist of 20 – 21 features (attributes), which are logged properties of the broker for every operation (or request) executed on it, captured on the client side. The target variable is the *duration* (latency) of the broker.

Time The date and time (in microseconds) when the log entry was created.

OperationIDForClient Incremental identifier of all the operations performed by one client.

GlobalOperationID Globally unique identifier of all operations executed on the broker.

Client Unique client identifier (also referred to as Client ID).

#StartActiveRequests Number of active requests on the broker at the beginning of the operation. Note that this feature was not present in earlier implementations of the test system.

#EndActiveRequests Number of active requests on the broker at the end of the operation. Also referred to as #ActiveRequests in earlier implementations of the test system.

#TotalSubscriptions Number of total subscriptions on the broker, which is the sum of all subscriptions to all existing topics.

Msg Message type of the operation (also referred to as *control message*) (cf. Section 2.2.1). Clients can `connect`, `disconnect`, `subscribe`, `unsubscribe` or `publish`. This is a categorical variable.

From Boolean value indicating the current state. Not relevant for our experiments.

To Boolean value indicating the next state. Not relevant for our experiments.

Retain Boolean value of the retain flag (cf. Section 2.2). Not relevant for our experiments and therefore always equal to *false*.

TopicSize Size of the topic, the client publishes to. Only set if the message type is `subscribe`, `unsubscribe` or `publish`.

CumulativeTopicSize Sum of all topic sizes on the broker.

MsgSize Size of the message published by the client. Only present if the message type is `publish`.

CumulativeMsgSize Sum of all message sizes on the broker.

#Subscribers Number of subscribers at the topic the client publishes to. Only present if the message type is `publish`.

#PublishReceiver Number of receivers of the published message. Only present when the message type is `publish`.

Success Boolean value indicating if the operation was executed successfully.

ExceptionMsg Field for error messages if any errors occurred.

PopulationSize Number of clients (population) of the current test case.

Duration Latency of the operation. The latency is measured in milliseconds from the time the client is sending the request, until it receives a response from the broker or in case of a `publish`, until all subscribers received the message.

3.4 Analysis and Statistics

We perform statistical analysis on the data in order to get a better understanding of the datasets and therefore to make it easier to find suitable cost models for them. Scatter plots and plots over time give us insights on how the data has to be processed and what deep learning method and hyper parameters should be used to achieve the best results. In the analysis we focus on the two major groups of datasets: The default dataset and the dataset with limited CPU resources.

3.4.1 Default Dataset

The default datasets consists of 254,752 samples for EMQ and 242,911 samples for Mosquitto. Note, that the default dataset consists of only 20 different features, compared to the 21 features of other datasets, because it was created with an older version of the test system. In this dataset, #StartActiveRequests is not present. However, it includes #ActiveRequests, which is equivalent to #EndActiveRequests of the newer datasets.

EMQ

For EMQ, the mean and standard deviation for the most relevant features are shown in Table 3.2. The statistical information for the target variable, the latency (duration), is given in Table 3.3. Overall the latency has a mean of 8.3884 ms and the standard deviation is 22.3957 ms.

		#ActiveRequests		#TotalSubscriptions		#Subscribers	
Msg Type	**count**	**mean**	**std**	**mean**	**std**	**mean**	**std**
connect	53255	22.7577	9.9333	279.2641	195.9930	–	–
disconnect	58177	19.6330	12.0123	276.3124	195.2427	–	–
subscribe	45032	22.8802	9.9221	275.5825	194.9149	–	–
unsubscribe	47198	22.8351	9.8820	278.4362	195.9785	–	–
publish	51090	22.9838	9.9007	275.7641	195.7054	1.5051	1.8507

Table 3.2: Mean values and standard deviation of the most relevant features for the EMQ default dataset.

Msg Type	min	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	max	mean	std
connect	0.7587	7.5436	20.6914	47.7472	410.8455	34.0358	38.0782
disconnect	0.0536	0.1399	0.1911	0.4130	197.1196	0.8186	4.1201
subscribe	0.1726	0.5622	0.8031	1.3087	227.1484	1.8164	5.5172
unsubscribe	0.1082	0.3579	0.5553	0.9810	270.9498	1.4148	4.6294
publish	0.1706	0.5744	0.8918	1.6208	200.9157	2.5094	7.4279
	0.0536	0.3885	0.7972	2.9910	410.8455	8.3884	22.3957

Table 3.3: Statistical values for the target variable (latency) in ms for the EMQ default dataset. Min *is the minimum latency, and* max *the maximum latency. The Q-values describe the quantile and the last two columns are the mean value and the standard deviation.*

Figure 3.2 shows the distribution of the latency (duration) as histogram. It is structured per message type and the red dotted line marks the mean value of the latency. Figure 3.3 shows the scatter plot for #ActiveRequests and #TotalSubscriptions, while Figure 3.4 shows the scatter plot for #Subscribers, MsgSize and TopicSize.

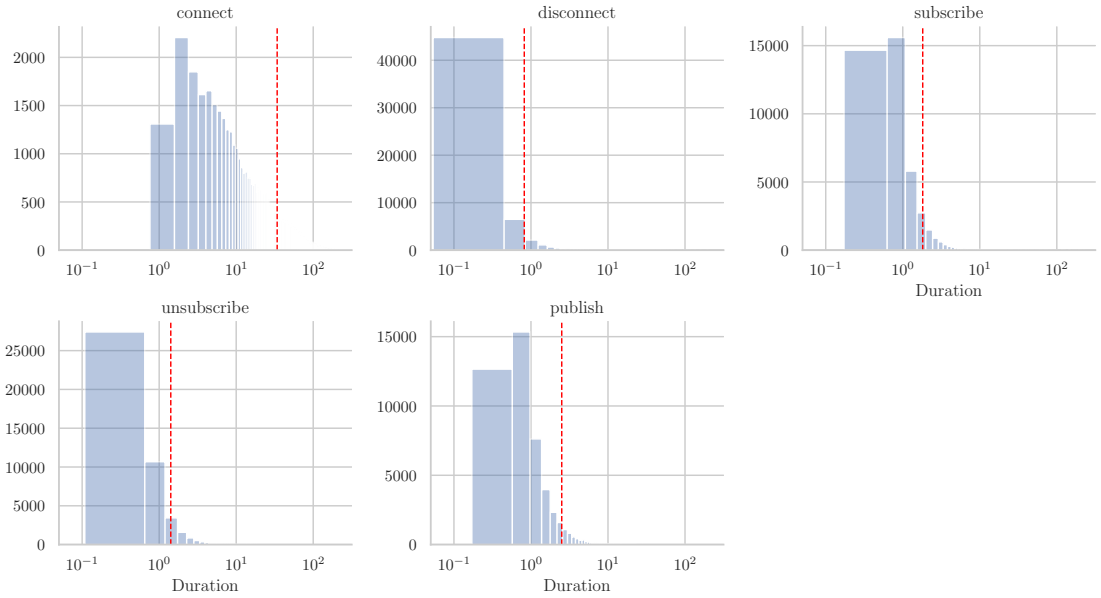


Figure 3.2: Distribution of the latencies per message type for the EMQ default dataset. The mean is displayed in red.

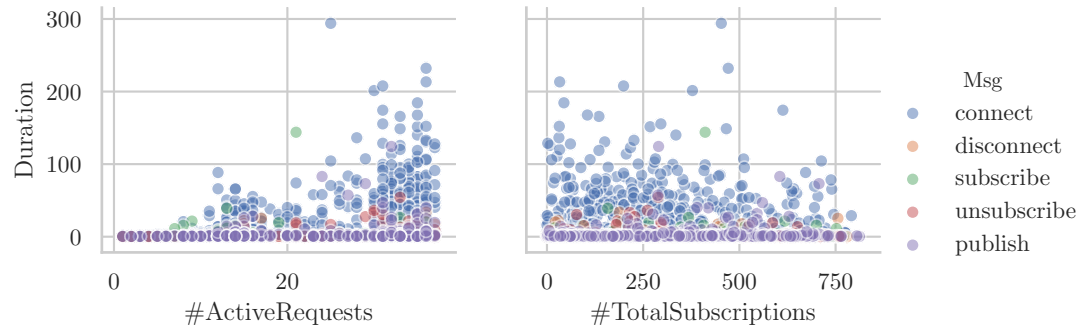


Figure 3.3: Scatter plot of #ActiveRequests and #TotalSubscriptions for a subsample of 1% of the EMQ default dataset.

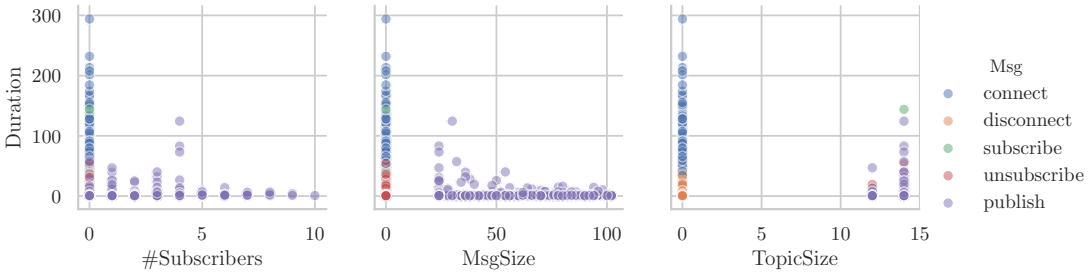


Figure 3.4: Scatter plot of #Subscribers, MsgSize and TopicSize for a subsample of 1% of the EMQ default dataset.

Mosquitto

For Mosquitto, the mean and standard deviation for the most relevant features are shown in Table 3.4. The statistical information for the target variable is given in Table 3.5. Overall the latency has a mean of 7.5457 ms and the standard deviation is 19.4222 ms.

		#ActiveRequests		#TotalSubscriptions		#Subscribers	
Msg Type	**count**	**mean**	**std**	**mean**	**std**	**mean**	**std**
connect	50886	22.3892	10.0489	270.3474	191.9621	–	–
disconnect	55580	19.2689	12.0329	269.4060	191.6482	–	–
subscribe	43013	22.4298	10.0958	268.1302	191.7639	–	–
unsubscribe	44930	22.4383	10.1194	270.8538	192.4423	–	–
publish	48502	22.5435	10.0900	268.7437	191.9409	1.5325	1.8529

Table 3.4: Mean values and standard deviation of the most relevant features for the Mosquitto default dataset.

Figure 3.5 shows the distribution of the latency (duration) as histogram. The scatter plots for Mosquitto are omitted, because they are similar to the scatter plots for EMQ

Msg Type	min	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	max	mean	std
connect	0.7626	4.5762	14.2127	38.7582	467.1957	27.8185	34.1847
disconnect	0.0541	0.1365	0.1682	0.2712	256.4340	0.6070	3.3153
subscribe	0.1496	0.7416	1.2263	2.1079	146.9460	2.5553	5.1801
unsubscribe	0.1062	0.7002	1.1514	1.9836	454.3836	2.3789	5.3354
publish	0.1535	0.9030	1.6422	2.9013	199.6367	3.4399	7.1008
	0.0541	0.4885	1.3199	3.7697	467.1957	7.5457	19.4222

Table 3.5: Statistical values for the target variable (latency) in ms for the Mosquitto default dataset. Min *is the minimum latency, and* max *the maximum latency. The Q-values describe the quantile and the last two columns are the mean value and the standard deviation.*

and no new insights can be gained.

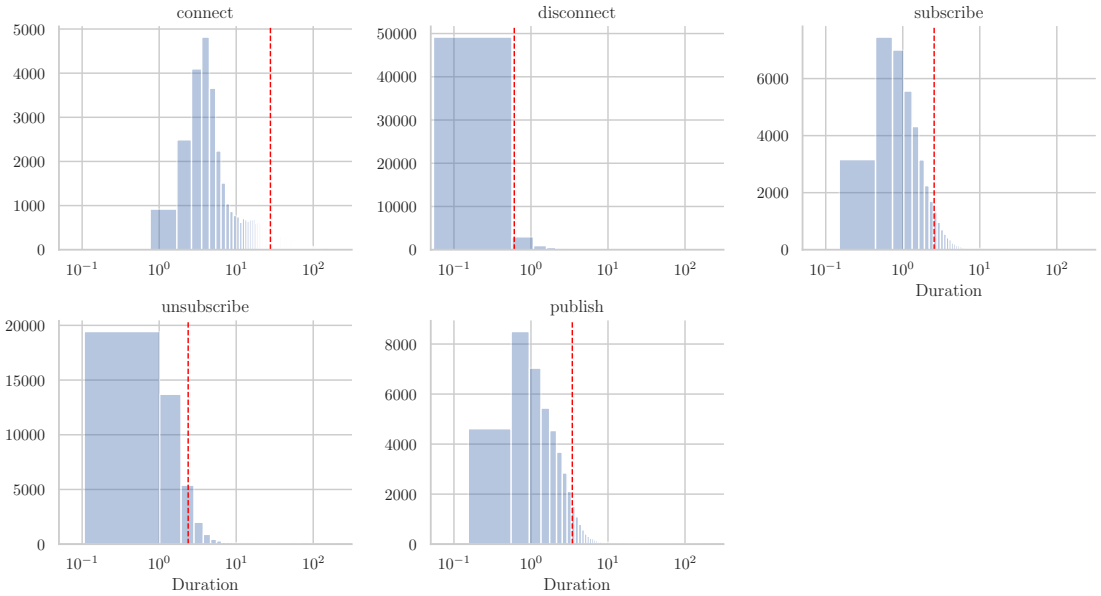


Figure 3.5: Distribution of the latencies per message type for the Mosquitto default dataset. The mean is displayed in red.

Discussion

The scatter plots of the default dataset do not give a lot of insights. There are higher latencies present, when the number of active requests increases, but the latency can also be on the same level with half of the active requests. The data is very noisy, has a high variance and is distributed all over the place. Especially the number of total subscriptions does not seem to have an influence on the latency at all. It is clearly noticeable, that the features #Subscribers and MsgSize are only dependent on the message type *publish* and are set to zero for the other message types. The feature TopicSize on the other hand is

only dependent on *subscribe*, *unsubscribe* and *publish*. It is also noticeable that the latency is not increasing, when the number of subscribers increases, which is not very intuitive. The latency also stays almost constant for increasing message or topic sizes.

As shown in the tables and the scatter plots, the control message *connect* takes on average much longer than the other message types. This is because the TCP/IP connection has to be initiated with a handshake first. Afterwards, the broker needs to create and store a session for the newly connected client.

When sorting the dataset by time or GlobalOperationId it is noticeable that #Active-Requests or #TotalSubscriptions are not raising in a constant way, they are jumping, which leads to the assumption that the samples in the dataset are not independent from each other. Putting them in groups of 10 and summarizing the features made a slight linear correlation visible between #ActiveRequests and the latency.

3.4.2 Limited CPU Dataset

The limited CPU datasets are created for limited CPU resources of 1%, 5% and 10% of the host system's CPU. They consist of around 300,000 samples, are structured in 21 different features and are created running the same experiments on the test system as for the default datasets. Note that for the limited CPU dataset we mainly focus on the EMQ broker and the 5% CPU limitation, in order to improve the readability of this work. For the limited CPU datasets we omit the mean and standard deviation for the features #ActiveRequests, #TotalSubscriptions and #Subscribers, as they stay almost constant over all experiments and message types (see Section 3.4.1).

1% CPU EMQ

For the 1% CPU EMQ dataset, the statistical information for the target variable is given in Table 3.6.

Msg Type	count	min	max	mean	std
connect	51918	1.9021	5695.4095	1616.1722	1039.1016
disconnect	56712	0.0677	45.2018	0.2443	1.0011
subscribe	43850	0.5592	4348.7390	629.3687	469.2520
unsubscribe	45914	0.3242	1577.8925	181.7533	146.5992
publish	49692	0.4607	5724.0788	350.7410	358.5365
	248086	0.0677	5724.0788	553.4135	797.3277

Table 3.6: Statistical values for the target variable (latency) for the 1% CPU EMQ dataset. Min *is the minimum latency, and* max *the maximum latency. The last two columns represent the mean value and the standard deviation.*

5% CPU EMQ

The statistical information for the target variable for the 5% CPU EMQ dataset is listed in Table 3.7. Figure 3.6 shows the distribution of the latency (duration) as histogram.

Figure 3.7 shows the scatter plot for #ActiveRequests and #TotalSubscriptions, while Figure 3.8 shows the scatter plot for #Subscribers, MsgSize and TopicSize. Note, that for EMQ, this figures are only shown for the 5% CPU EMQ dataset, as they are similar for the 1% and 10% CPU EMQ datasets.

Msg Type	**count**	**min**	**max**	**mean**	**std**
connect	62259	1.8080	5090.4812	1448.7444	839.5920
disconnect	67991	0.0658	42.8013	0.2481	1.0589
subscribe	52393	0.5436	2808.1109	575.1638	388.8421
unsubscribe	54587	0.3374	1434.2807	164.7632	121.4869
publish	59422	0.4954	4996.5954	321.7108	304.0088
	296652	0.0658	5090.4812	500.4498	685.6155

Table 3.7: Statistical values for the target variable (latency) for the 5% CPU EMQ dataset. Min *is the minimum latency, and* max *the maximum latency. The last two columns represent the mean value and the standard deviation.*

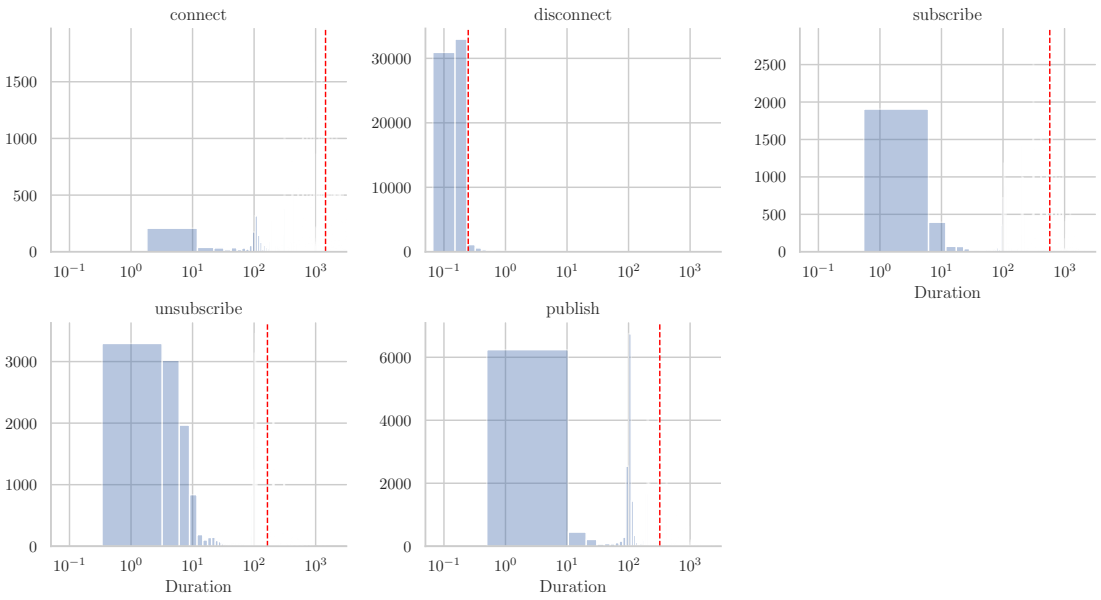


Figure 3.6: Distribution of the latencies per message type for EMQ with 5% CPU. The mean is displayed in red.

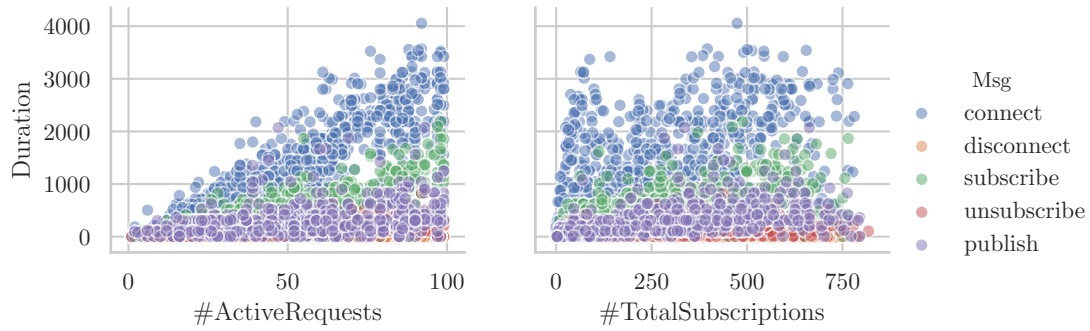


Figure 3.7: Scatter plot for a subsample of 1% for the 5% CPU EMQ dataset.

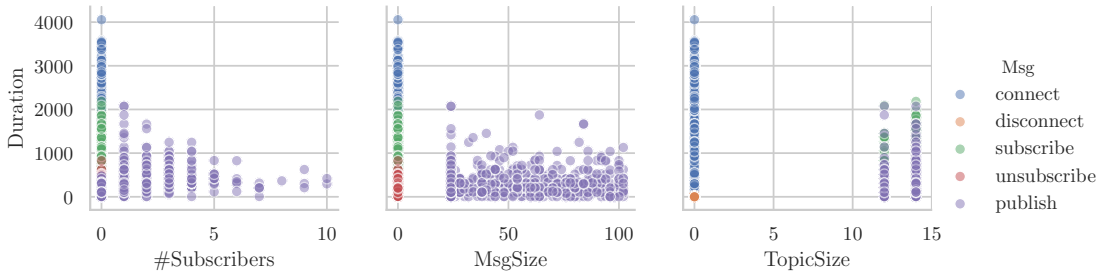


Figure 3.8: Scatter plot for a subsample of 1% for the 5% CPU EMQ dataset with features more relevant to the message type publish.

5% CPU Mosquitto

The statistical information for the target variable for the 5% CPU Mosquitto dataset is listed in Table 3.8. Figure 3.9 shows the scatter plot for the number of active requests and the number of total subscriptions.

Msg Type	count	min	max	mean	std
connect	54555	1.1419	3206.6305	185.7904	122.7880
disconnect	59580	0.0750	109.9941	0.4585	2.1792
subscribe	45913	0.2374	2180.4905	124.4450	132.3105
unsubscribe	48191	0.2135	2481.9387	127.1247	143.8745
publish	51847	0.2340	1875.4681	181.7770	162.0015
	260086	0.0750	3206.6305	120.8355	142.3915

Table 3.8: Statistical values for the target variable (latency) for the 5% CPU Mosquitto dataset. Min *is the minimum latency, and* max *the maximum latency. The last two columns represent the mean value and the standard deviation.*

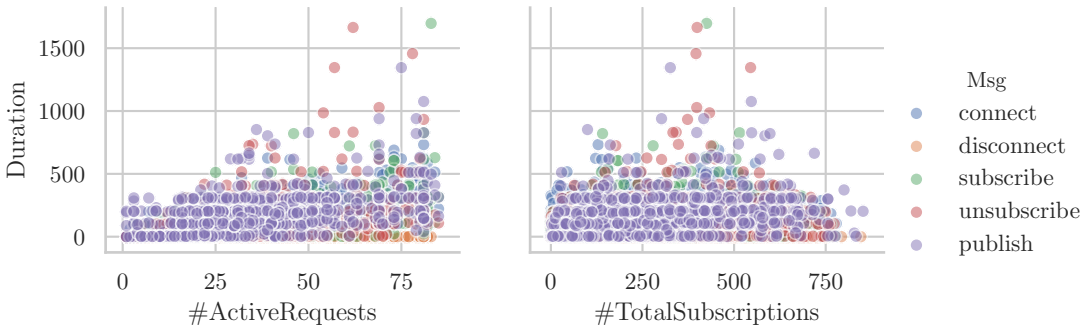


Figure 3.9: Scatter plot for a subsample 1% of the 5% CPU Mosquitto dataset.

5% CPU VerneMQ

The statistical information for the target variable for the 5% CPU VerneMQ dataset is listed in Table 3.9. Figure 3.10 shows the scatter plot for #ActiveRequests and #TotalSubscriptions.

Msg Type	count	min	max	mean	std
connect	51437	2.1347	23151.3111	3130.6190	2273.1320
disconnect	56228	0.0750	174.7420	0.2810	1.4671
subscribe	43313	0.6226	10007.3503	588.7484	314.6931
unsubscribe	46019	0.5753	3727.9271	199.0021	155.8423
publish	49594	0.5846	27035.3788	384.6339	864.0394
	246591	0.0750	27035.3788	870.9941	1622.8523

Table 3.9: Statistical values for the target variable (latency) for the 5% CPU VerneMQ dataset. Min *is the minimum latency, and* max *the maximum latency. The last two columns represent the mean value and the standard deviation.*

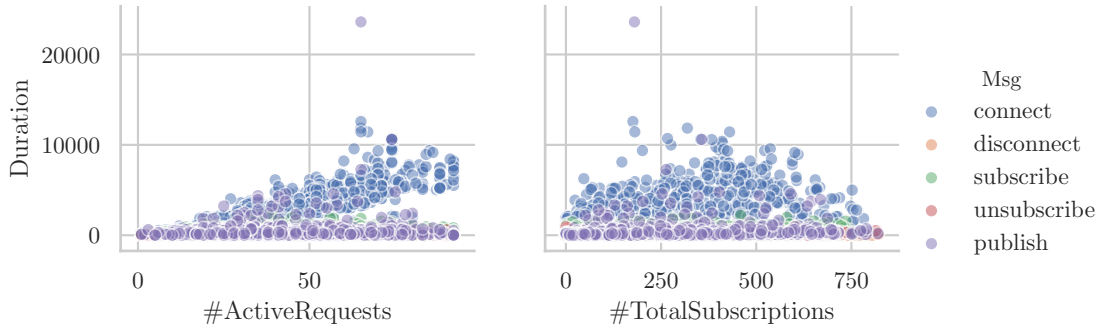


Figure 3.10: Scatter plot for a subsample 1% of the 5% CPU VerneMQ dataset.

10% CPU EMQ

For the 10% CPU EMQ dataset, the statistical information for the target variable is given in Table 3.10.

Msg Type	count	min	max	mean	std
connect	47229	1.6730	5222.9173	1415.9095	879.5023
disconnect	51601	0.0755	46.1624	0.2308	0.8521
subscribe	40078	0.4159	2694.2412	571.6901	395.8018
unsubscribe	42065	0.3696	1241.0303	165.5091	127.4656
publish	45263	0.3496	4885.5129	312.8729	303.0169
	226236	0.0755	5222.9173	490.2836	686.5436

Table 3.10: Statistical values for the target variable (latency) for the 10% CPU EMQ dataset. Min *is the minimum latency, and* max *the maximum latency. The last two columns represent the mean value and the standard deviation.*

Discussion

The mean values and standard deviation increase dramatically compared to the default dataset. The message types *connect* and *publish* are the control messages with the longest duration, while *disconnect* is still short compared to all other message types (except for the Mosquitto dataset). In the scatter plots for EMQ and VerneMQ we see a slight linear trend, when increasing the number of active requests. However, for the 5% CPU Mosquitto dataset there is no linear trend visible and the mean of the latency is much lower than for the other datasets. It seems that limiting the CPU resources did not eliminate most of the noise in this dataset.

Same as for the default dataset, increasing the number of subscribers, the message or topic size does not have a lot of impact on the latency. An increasing number of total subscribers has an impact on the latency too, but in the scatter plot, there is also a lot of noise present.

Note that the percentage values of the CPU limitations should not be taken too seriously. Repeated experiments have shown a high variance of the mean latencies for the limited CPU datasets. The reason for this is not only the randomness in the experiments. Also the virtualization with Virtual Box and inaccuracies in the CPU resource limitation might have an impact on the variance of the latencies.

3.4.3 Summary

Except for Mosquitto, we can see a linear trend for increasing numbers of active requests. To make the trend more visible, we created a line plot instead of the previous scatter plot. For the EMQ default dataset it is shown in Figure 3.11 and for the 5% limited CPU EMQ dataset it is shown in Figure 3.12. The limited CPU dataset clearly shows a linear trend for all message types, while the default dataset increases only slightly for the message type *connect*, with a high variance (shown as shaded area around the line). The number of total

subscriptions are very noisy, but also slightly increasing in the limited CPU dataset. The latency does not really increase with a higher number of subscribers, as seen on the right.

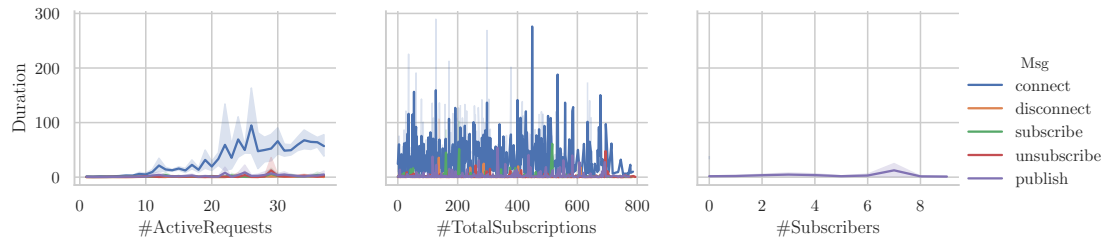


Figure 3.11: Line plot for the most relevant variables for the EMQ default dataset. The shaded area represents the variance.

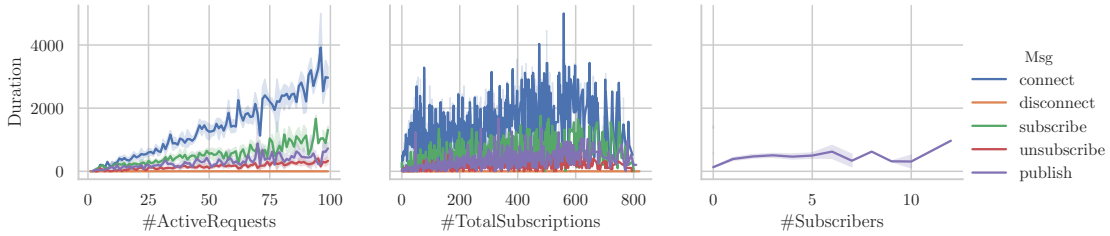
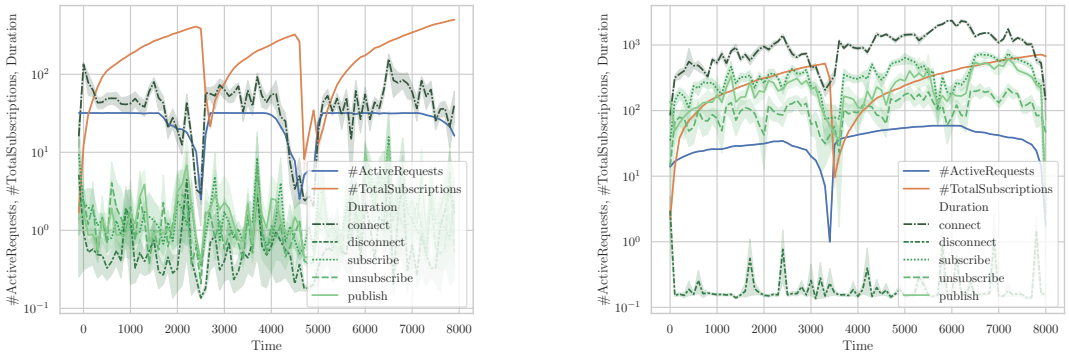


Figure 3.12: Line plot for the most relevant variables for the 5% CPU EMQ dataset. The shaded area represents the variance.

Figure 3.13 shows the features of the datasets over time. As seen in the increasing and decreasing number of active requests and number of total subscriptions, there are multiple test cases run after each other. So, the number of active requests and the number of total subscriptions are reset with every start of an experiment. One can also see that the latencies approximately follow the same pattern for the default dataset in Figure 3.13a, while the duration in Figure 3.13b for the 5% CPU EMQ dataset does not fluctuate that much and seems to be more stable.

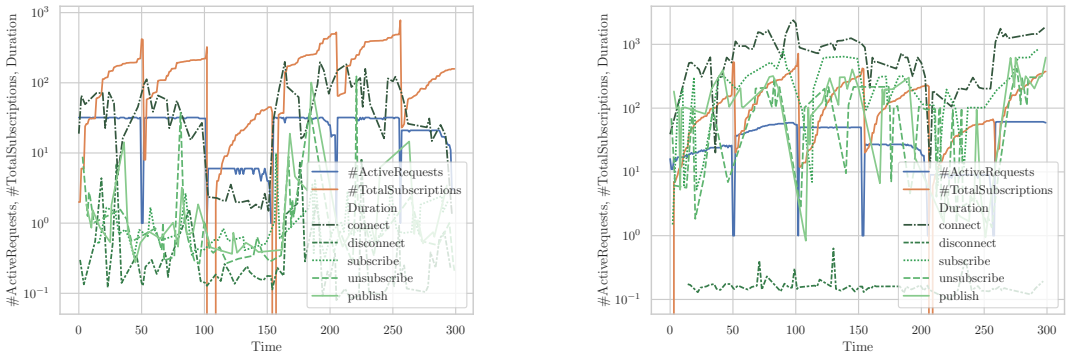
Figure 3.14 shows the data over time for one user (`client0`) only. As in the previous Figure 3.13, the different experiments run after each other. The duration in the 5% CPU EMQ dataset (Figure 3.14b) does not change as much as the data in the default dataset (Figure 3.14a).

There is definitely a time dependency in the datasets, which is probably caused by the testing server's scheduling mechanism. As shown in the figures, the variance of the duration is decreasing, if the CPU resources are limited to 5%. One can clearly observe this in Figure 3.13. Because of the limited CPU resources, the events processed on the broker take longer and timing effects caused by scheduling etc. on the testing server are not as significant anymore. However, for Mosquitto it seems that the noise is not decreasing when the CPU resources are limited as shown in Figure 3.15, because the latencies are lower than for VerneMQ or EMQ and therefore errors are more prominent.



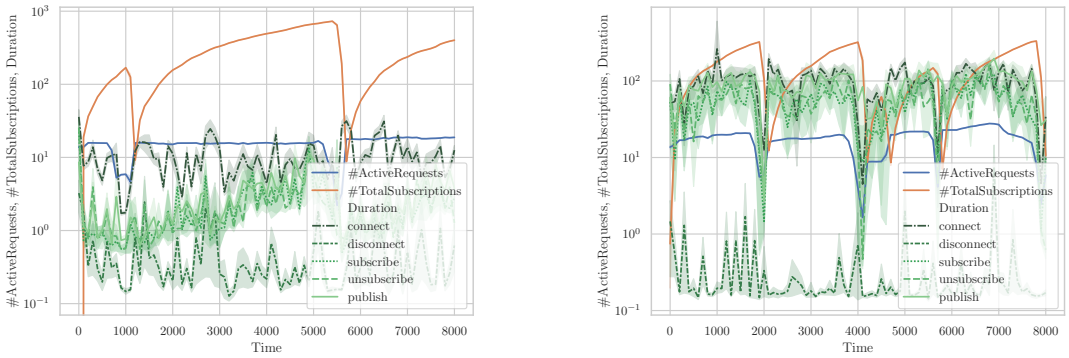
(a) default EMQ dataset *(b) 5% CPU EMQ dataset*

Figure 3.13: Most relevant features over time for the default dataset and the 5% CPU EMQ dataset.



(a) default EMQ dataset *(b) 5% CPU EMQ dataset*

Figure 3.14: Most relevant features over time for the default dataset and the 5% CPU EMQ dataset and one user (`client0`) only.



(a) default EMQ dataset *(b) 5% CPU EMQ dataset*

Figure 3.15: Most relevant features over time for the default dataset and the 5% CPU dataset for Mosquitto.

3.5 Preprocessing

Before we can use the datasets to train the neural networks, some preprocessing is performed in order to increase the quality of the data. The data has to be put in the required format, so it can be used as an input for the (recurrent) neural networks. We also filter out unsuccessful events, failed because of timeouts or other reasons.

For the recurrent neural network, the dataset is sorted ascending by *GlobalOperationID* and converted into a cubical form with features, time steps and batch size as dimensions as shown in Figure 3.16.

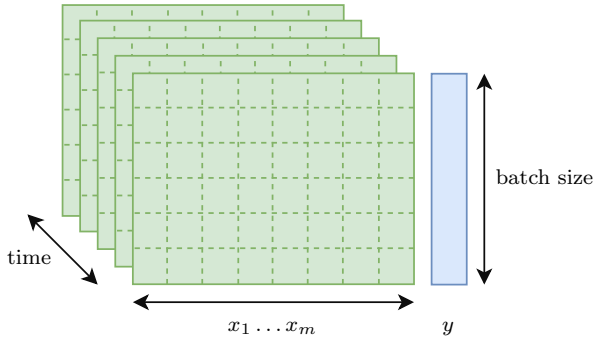


Figure 3.16: Converted dataset for recurrent neural networks. The features are displayed in green and the target variable is displayed in blue.

The dataset is shuffled and split into a training and validation set, using a ratio of 80% and 20% respectively. Note, that Aichernig and Schumi used all of the data for learning the linear regression. Afterwards, we apply one-hot encoding, feature and target scaling as described in the following sections.

3.5.1 One-Hot Encoding

Since the feature *message type* is a categorical variable, we have to use one-hot encoding to convert the categories into indicator variables. This means, that *Msg* is split up into the binary features Msg_connect, Msg_disconnect, Msg_publish, Msg_subscribe and Msg_unsubscribe as shown in Table 3.11.

Msg:	connect	disconnect	publish	subscribe	unsubscribe
Msg_connect	1	0	0	0	0
Msg_disconnect	0	1	0	0	0
Msg_publish	0	0	1	0	0
Msg_subscribe	0	0	0	1	0
Msg_unsubscribe	0	0	0	0	1

Table 3.11: One-Hot encoding of the categorical variable Msg *to the indicator variable in the left column.*

3.5.2 Feature Scaling

To keep the values of the features and therefore also the parameters of the neural network small, we use feature scaling. One method to scale features, is to standardize them, which means the features are scaled to zero mean and unit variance. An alternative to standardization is the use of a Min-Max scaler, which works slightly better for our data. The Min-Max scaler transforms the features to a scale between 0 and 1:

$$x_{i,\text{scaled}} = \frac{x_i - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})} \tag{3.1}$$

3.5.3 Target Scaling

Since the latency can range from fractions of a millisecond to over a second, we also scaled the target variable $\boldsymbol{y}$ for some of the experiments. We use the logarithm to scale $\boldsymbol{y}$:

$$y_{i,\text{scaled}} = \log(y_i) \tag{3.2}$$

3.6 Learning

Developing the cost model is the main part of this thesis and is also referred to as learning. In the original test system, multiple linear regression is used as cost model, which serves as a baseline for the deep learning methods we are going to apply in this thesis. We are using multiple different methods like (deep) neural networks or GRUs, depending on the dataset we are going to process.

3.6.1 Multiple Linear Regression

Aichernig and Schumi [3] were using a multiple linear regression to create a cost model in the original implementation of the test system. To achieve fairly good results, they had to perform a lot of feature analysis and preprocessing. The data was checked for any biases in the log-data with the use of scatter plots, histograms and correlation matrices. In a next step, they cleaned the data i.e. they removed outliers (top 5% latencies per message type) and log entries with error messages. In an extensive feature selection step, they only kept features that had an impact on the latency. Features with no additional information were not considered in the cost model. To do this, they used the Pearson correlation coefficient. They also found out, that some features are only relevant for certain message types and therefore set the features #ActiveRequests and #TotalSubscriptions to zero for some message types.

The linear regression was performed in R [40] with the `lm` function and using the Features *Msg*, *EndActiveRequests*, *TotalSubscriptions* and *Subscriptions*. The R Stats Package automatically splits the message type into dummy features (dummy encoding). It outputs a table of regression coefficients, standard errors, t-values and the significance of each feature. The output was written to a text file, which was then loaded by the test system. In order to use the linear regression as a baseline and to make comparison easier, we re-implement the R project of Aichernig and Schumi in Python, using scikit-learn [41].

3.6.2 (Recurrent) Neural Network

In this work, we compare various different architectures of (recurrent) neural networks. We implemented the models using Python 3.6[6] with Keras [42] and a TensorFlow [43] back end. The experiments were performed on a MacBook Pro with a Intel Core i5 2 GHz dual-core CPU and 8 GB of RAM running macOS 10.13.6. Some experiments are performed on a Linux GPU cluster, running Debian 4.9.82 and utilizing multiple NVIDIA Tesla K40c GPUs with 12 GB of RAM.

As input features we selected #ActiveRequests, #TotalSubscriptions, #Subscribers, Msg, TopicSize, MsgSize and #PublishReceiver. Other features such as Retain, From or To were omitted, because they were constant throughout all experiments or not relevant. Using one-hot encoding for the message type, sums the number of input features up to 11. We applied a Min-Max scaler on all of our features. Scaling the target using the logarithm was not an advantage for most of the experiments.

In all of our experiments we used the MSE (2.13) as loss function and an ADAM optimizer with a learning rate of 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a decay of 0.0.

Architecture

As neural network architecture, we used multiple densely connected layers with a ReLU activation function (2.22) and linear activation in the output layer. All the other settings were left to their default values (bias initialized with zeros, weights initialized using the Glorot uniform initializer [44]). As regularization in the standard neural network we used early stopping. We experimented using dropout with a rate between 0.1 and 0.2 applied after every hidden layer, but improvements could not be observed.

For the recurrent neural network, we used NVIDIA's cuDNN[7] GRU implementation for the recurrent units. This implementation does not offer dropout. Therefore, we used kernel and recurrent regularizers (weight decay) for some experiments, but no improvement could be observed. The architecture consist of multiple stacked GRU layers and one dense layer with linear activation as output layer. As activation function in the recurrent layers we used the hyperbolic tangent and the sigmoid function in the recurrent step, as suggested in [32]. All the other settings were left to their default values (bias initialized with zeros, weights initialized using the Glorot uniform initializer).

For the neural network as well as for the recurrent neural network we used 1 – 10 hidden layers with 6 – 512 neurons (or units) in the first hidden layer and a same or decreasing number of neurons in the subsequent layers. The architecture is denoted by the number of neurons per layer, split by hyphens e.g. 11-12-6-1 for an input layer of size 11, one hidden layer with 12 neurons, one hidden layer with 6 neurons, and an output layer with one neuron.

3.7 Serving

To use the cost model, written in Python, in the enclosing test system, written in C#, we had to implement a *serving platform*. We used a small Flask[8] server, which loads the

[6] `https://www.python.org`
[7] `https://developer.nvidia.com/cudnn`
[8] `http://flask.pocoo.org`

trained Keras model (saved in h5 format), does the preprocessing and responses with a prediction. Requests can be submitted via *HTTP GET* method e.g.

```
http://127.0.0.1:5000/predict?msg=publish&totalsubscriptions=42&
    subscribers=7&activerequests=10
```

to get a prediction for the message type *publish* with 42 total subscriptions, 7 subscribers and 10 active requests. The server then responses with the prediction in *JSON* format e.g.

```
{"prediction":5.261019706726074,"success":true}
```

for a predicted latency of approximately 5.2610 ms.

Another option, which turned out to be faster and easier to integrate, is to use the standard input and output of the Python script instead of a Flask server to communicate with the cost model.

4 Experimental Results

This chapter presents the experimental results we achieved in this work. It shows the architectures that performed best with the different datasets. We focused on the EMQ and Mosquitto default datasets as well as the 1%, 5% and 10% limited CPU EMQ datasets (see Section 3.2.2) for our comparison. At first we describe the results achieved with the default dataset and what challenges we were facing in Section 4.1. We tried standard neural networks as well as recurrent neural networks to cover the sequential dependencies we found out about in the statistical analysis. In Section 4.2, we perform experiments with standard neural networks using the limited CPU datasets. As a baseline for all of our experiments, we used the multiple linear regression from [3] with all its preprocessing steps as described in Section 3.6.1. To have a better comparison with unprocessed data, we ran the linear regression also on the datasets without any preprocessing. We tried different architectures, from shallow to very deep neural networks. At the end of this chapter, Section 4.3 gives a summary of the results achieved in our experiments.

4.1 Default Dataset

The results for the linear regression, with and without preprocessing, are listed in Tables 4.1. Note that the *MSE* of the linear regression with preprocessing cannot be used for comparison, because the preprocessing step removed all the outliers. Figure 4.1 shows

Dataset	**R^2**	**MSE**
EMQ	0.3906	309.3296
Mosquitto	0.3760	229.1834

(a) without preprocessing

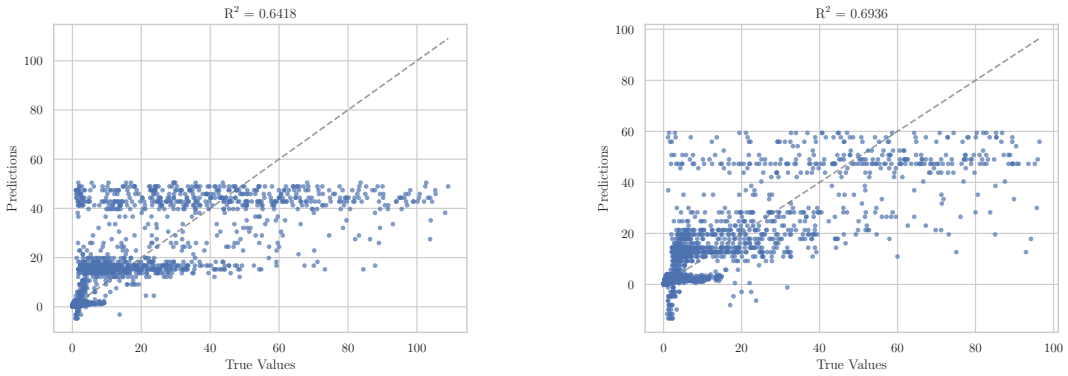
Dataset	**R^2**	**MSE**
EMQ	0.6418	93.5200
Mosquitto	0.6936	56.6065

(b) with preprocessing

Table 4.1: Results of the linear regression, with and without preprocessing, for the default datasets.

the results of the linear regression (with preprocessing) from Table 4.1b as scatter plot. It shows the true duration (latency) in the x-axis and the predicted duration in the y-axis.

The dashed 45° line represents the ideal result for the regression, where all the predictions are equal to the real values.



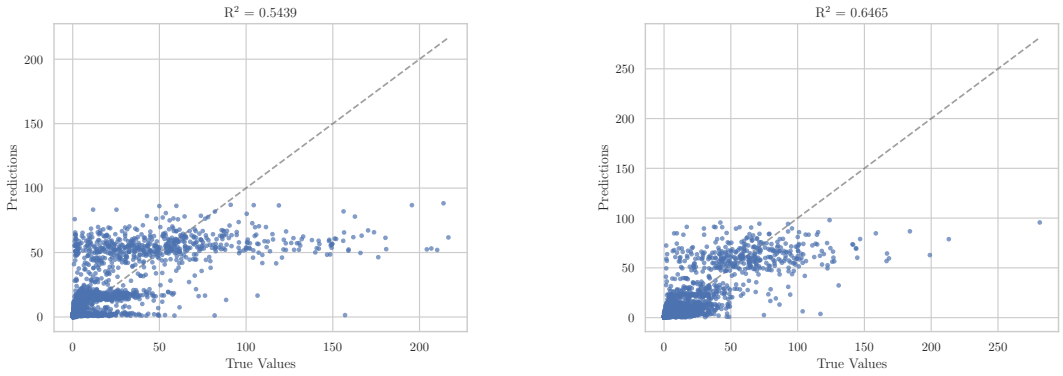
(a) EMQ *(b) Mosquitto*

Figure 4.1: True vs. predicted duration (latency) for the linear regression with preprocessing and the default datasets.

As a next step, we tried different neural network architectures, starting with very shallow neural networks like 11-6-1 to very deep neural networks like 11-265-128-64-32-16-8-1. The best results were achieved with the 11-265-128-64-1 network and are listed in Table 4.2 under the *NN* column and graphically in Figure 4.2. As noticed in the analysis in Section 3.4, we can observe a time dependency in the default dataset. To take those dependencies into account and process a sequence of data, we concatenated multiple time steps from one user and trained a 11-256-128-64-1 neural network (referred to as $NN_{sequence}$) as well as a 11-256-128-64-1 GRU. The best results for that experiments are listed in Table 4.2 and shown in Figure 4.3 for the neural network and Figure 4.4 for the GRU.

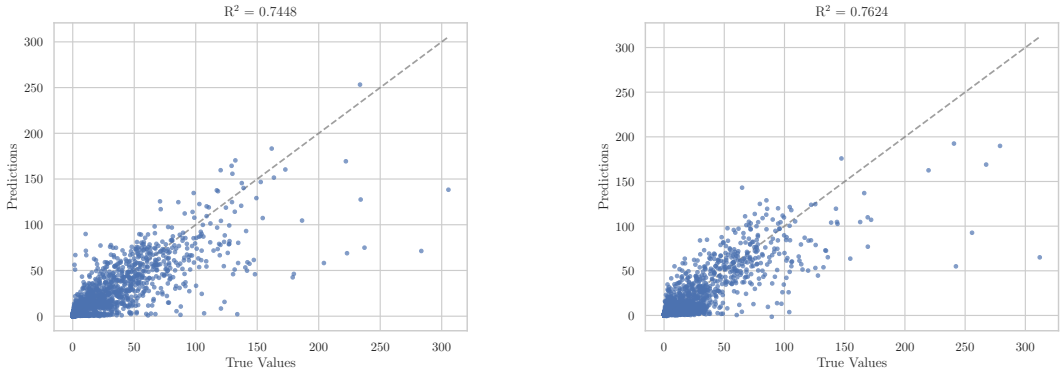
	NN		**$NN_{sequence}$**		**GRU**	
Dataset	**R^2**	**MSE**	**R^2**	**MSE**	**R^2**	**MSE**
EMQ	0.5439	231.5091	0.7448	124.8604	0.7606	117.1198
Mosquitto	0.6465	129.8170	0.7624	89.4789	0.7915	78.5293

Table 4.2: Results of the 11-256-128-64-1 neural network and GRU architecture for the default datasets.



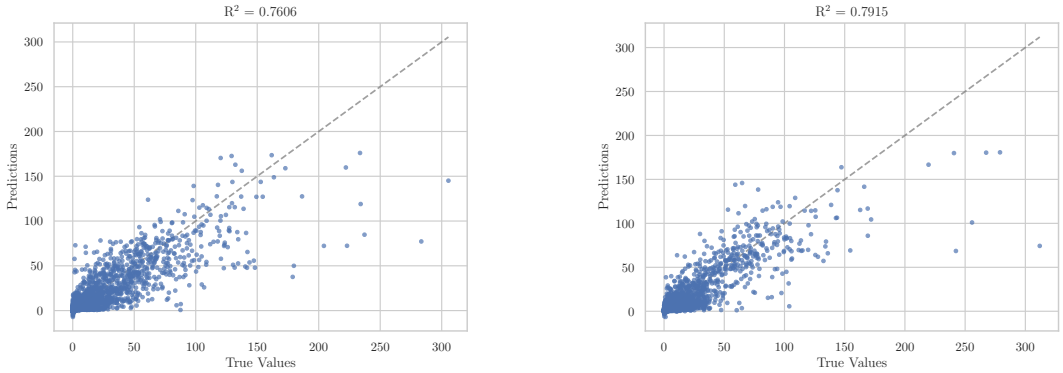
(a) EMQ (b) Mosquitto

Figure 4.2: True vs. predicted duration (latency) for the 11-256-128-64-1 neural network and the default datasets.



(a) EMQ (b) Mosquitto

Figure 4.3: True vs. predicted duration (latency) for the 11-256-128-64-1 neural network using the concatenation of three time steps as input feature vector on the default datasets.



(a) EMQ (b) Mosquitto

Figure 4.4: True vs. predicted duration (latency) for a 11-256-128-64-1 GRU using a sequence of length 3 on the default datasets.

4.1.1 Discussion

As seen in the analysis and statistics of the data (see Section 3.4) the dataset has a time dependency. This is the reason the recurrent neural network performed best for this dataset. Because the scheduling is also depended on the current user, the results turned out to be better after grouping the dataset by users and then sorting it than just sorting it by time.

Good results were also achieved by using a neural network with the time steps concatenated. Methods which did not consider the time dependencies, did not lead to good results as seen in the results. The standard neural network did not perform better than the linear regression with preprocessing. But it performed much better than the linear regression without preprocessing. As seen in the scatter plots, the preprocessing removed all the outliers and therefore most latencies above 110. That is the reason, why the axes in the plots in Figure 4.1 are limited to around 120, while the range of other plots is to around 225. The scatter plots also show, that the neural network and the linear regression did not perform very well for larger latencies.

Because the neural network did not perform better than the linear regression with preprocessing, we tried a lot of different architectures and parameters. We trained networks down to a depth of 8 layers, but there was not a lot of improvement. The 11-265-128-64-1 architecture was a good compromise between decent results and not too many parameters and therefore a better performance. However, architectures with less parameters, like 11-12-12-6-6-1, did not perform much worse with an R^2 value of around 0.5369.

Similar as for the architecture, we also performed experiments with a different amount of time steps for the GRU and the neural network with concatenated time steps. The best results were achieved with only three time steps and the dataset grouped by clients. We tried to increase the time steps up to 800, but the results of the network did not increase, while the training and prediction time of it drastically decreased. The results for different time steps, using the 11-12-12-6-6-1 neural network architecture and the EMQ default dataset are listed in Table 4.3.

Time Steps	$\mathbf{R^2}$	**MSE**
3	0.7339	130.1850
6	0.6971	148.9849
8	0.6902	157.3421

Table 4.3: Different time steps settings for the 11-12-12-6-6-1 neural network architecture and the EMQ default dataset.

Note that for the GRU, even though we used the same number of units and layers as for the neural network, the number of parameters is significantly higher because of the structure of a GRU unit. As shown in Section 2.5.1, one GRU unit consists of six weight matrices, while artificial neurons consist of only one. The GRU therefore needs around 4 ms for the prediction, which is twice as much time needed than the neural network. Compared to the mean latency of the default datasets (around 8 ms), this is a only a speedup of a factor of 2.

4.2 Limited CPU Dataset

The multiple linear regression results of the limited CPU datasets are listed in Tables 4.4. A scatter plot is shown in Figure 4.5 for the data with preprocessing.

Dataset	R^2	MSE
1% CPU EMQ	0.6852	146106.8164
5% CPU EMQ	0.7085	136517.9478
10% CPU EMQ	0.6852	455975.1158

(a) without preprocessing

Dataset	R^2	MSE
1% CPU EMQ	0.8697	47924.4792
5% CPU EMQ	0.8631	52431.6899
10% CPU EMQ	0.7526	181229.1207

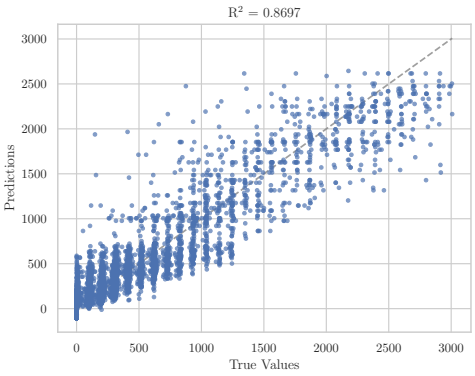
(b) with preprocessing

Table 4.4: Results for the linear regression with and without preprocessing for the limited CPU datasets.

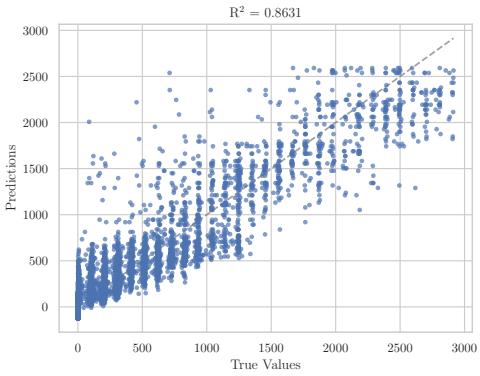
Because of limiting the CPU resources, dependency of samples over time was not as prominent anymore and therefore, sequential neural networks have not been applied for this datasets. In this experiment, we also tried different neural network architectures, but ended up taking the same as for the default dataset. This also shows that this architecture is able to generalize well between different use cases. The results for the 11-265-128-64-1 architecture are listed in Table 4.5 and the scatter plot can be seen in Figure 4.6.

Dataset	R^2	MSE
1% CPU EMQ	0.9152	39348.0030
5% CPU EMQ	0.9141	40214.0486
10% CPU EMQ	0.7737	243292.5299

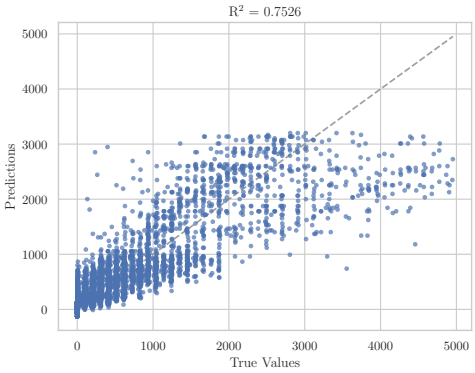
Table 4.5: Results of the 11-256-128-64-1 neural network architecture for the limited CPU datasets.



(a) 1%

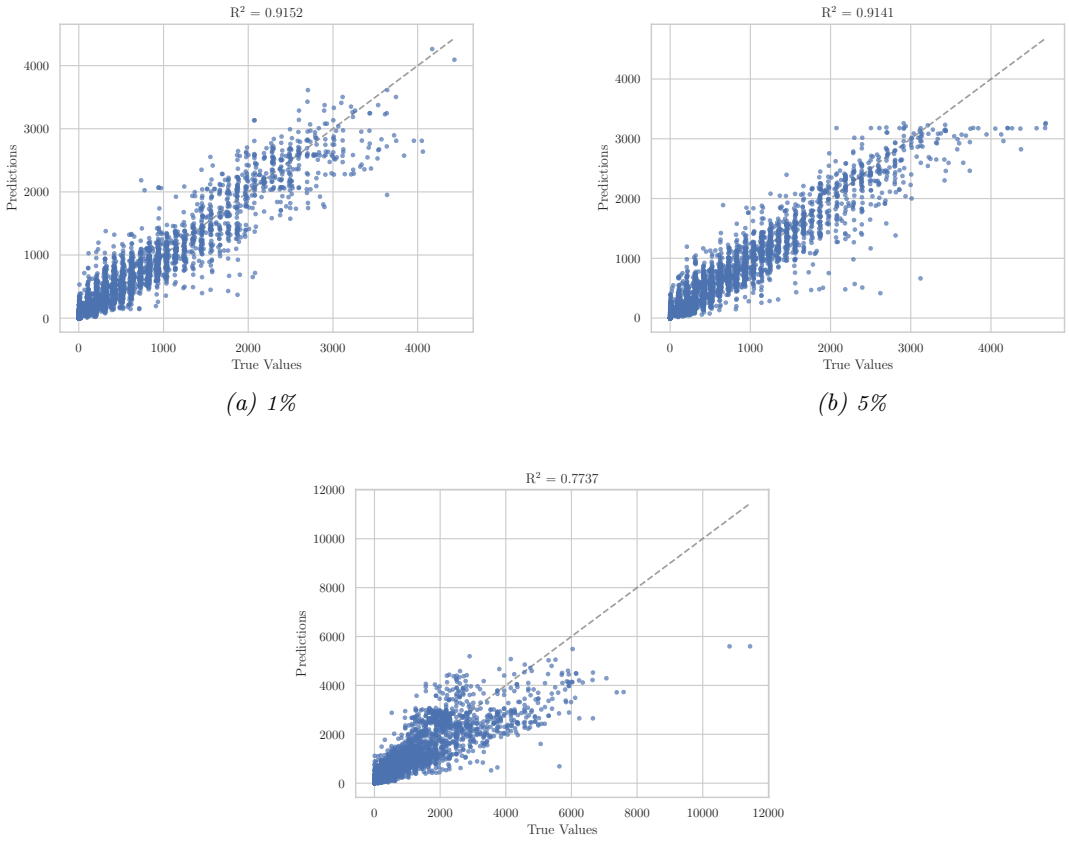


(b) 5%



(c) 10%

Figure 4.5: True vs. predicted duration (latency) for the linear regression and the limited CPU EMQ datasets.



(a) 1%

(b) 5%

(c) 10%

Figure 4.6: True vs. predicted duration (latency) for the 11-256-128-64-1 neural network and the limited CPU EMQ datasets.

4.2.1 Discussion

For the limited CPU EMQ datasets we achieved very good results with a multiple linear regression already, compared to the default dataset. The neural network performed even better and we could achieve a maximum R^2 value of 0.9152 for the 1% limited CPU dataset, with the predicted duration being aligned along the 45° line. The network architecture is also not too large and therefore does not contain too many weights to slow down the computation of the prediction. Taking around 1 – 2 ms for the prediction is significantly faster than the mean latency of the experiments on the SUT, which took around 500 – 600 ms in average.

As with the default dataset, we tried various different architectures. A small excerpt from the architectures we tried and the corresponding results are given in Table 4.6. The more parameters we used, the better the network performed in general. In order to speed up the computation of the prediction and make the networks as simple as possible, we decided to use the 11-256-128-64-1 architecture, because it provided decent results with a reasonable amount of parameters. All in all, the difference in the R^2 values is not too large between the architectures. A very simple architecture like 11-12-1 already performed better than the linear regression and it requires less effort for preprocessing. A scatter plot of the architecture is shown in Figure 4.7

Architecture	$\mathbf{R^2}$	**MSE**
11-12-1	0.8865	53133.7090
11-12-6-1	0.8894	51799.8317
11-12-12-6-6-1	0.9079	43093.4257
11-64-32-16-1	0.9110	41675.5917
11-64-32-16-8-1	0.9112	41579.1678
11-128-64-32-16-8-1	0.9133	40596.8134
11-256-128-64-1	0.9141	40214.0486
11-256-128-64-32-16-8-1	0.9168	38958.4936

Table 4.6: Comparison of a sample of different architectures for the 5% limited CPU EMQ dataset.

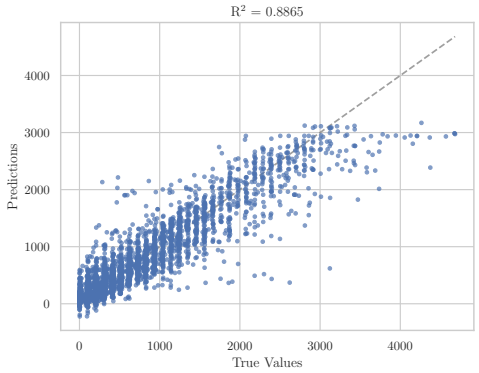


Figure 4.7: True vs. predicted duration (latency) for the 12-1 neural network and the 5% limited CPU dataset.

4.3 Summary

Compared to the multiple linear regression baseline model, the deep learning methods we used achieved significantly better results. It was possible to get an R^2 value of over 0.9 and therefore a better fit of the regression than with the multiple linear regression. The neural network performed best for the dataset with limited CPU resources. The limitation is also close to a real-world IoT scenario that utilizes small and energy efficient devices. Because of the CPU resource limitations, the operations performed by the broker take longer and therefore other influences of the latency like scheduling, errors etc. are negligible. The baseline model, the multiple linear regression, performed well on the limited CPU dataset already, but a lot of preprocessing had to be performed in order to achieve those results. For the neural network, only the most necessary preprocessing tasks, like one-hot encoding, feature and target scaling were performed. Those tasks do not require human interaction and can therefore be automated and integrated into the test system. It was not possible to find a well-performing neural network for the default dataset because of the time dependencies among the data samples. However, processing a sequence of three consecutive samples with a neural network or a GRU solved that issue and led to a higher R^2 value and the predictions aligned along the 45° line in the scatter plots. Compared to the unprocessed data in the linear regression, also the neural network already led to good results for the default datasets. Changing the architecture of the neural network increased its performance when using more layers and neurons. But compared to the amount of additional parameters, the performance was not much better and the network probably learned more noise of our datasets.

Of course, the time to train a neural network is higher than to fit the linear regression. Also the computation of the prediction takes longer as with the baseline linear regression model (approximately twice as long). Compared to the time it takes to execute the experiments on the SUT, the neural networks performed still at least twice as fast. For the limited CPU datasets, the prediction time of approximately $1 - 2$ ms and therefore a speedup by a factor of up to 500, compared to the mean latencies on the SUT, was a huge improvement in performance.

5 Conclusion

In this work we used deep learning methods to create a cost model for an MQTT performance test system. The aim of this thesis was to find a cost model which does not require a lot of effort in preprocessing and at the same time which improves the results of the prediction compared to the old cost model, a multiple linear regression.

The cost model was trained to predict the latency of MQTT broker implementations and integrated into a test system which then compares the performances of them. The datasets used for training were log-data of different testing scenarios and broker implementations. The data was automatically created by the test system, using model-based testing within a property-based testing tool.

To find the most suitable deep learning methods for creating the cost model, we performed an extensive amount of statistical analysis. We found out that, because of scheduling on the test server which ran the broker as well as all the clients, a strong dependency on clients and between consecutive samples exits. These dependencies made it necessary to pick a recurrent neural network, which is capable of processing multiple time steps from the past. The results achieved with GRUs were a huge improvement to the linear regression baseline, but, because of the complex architecture and vast amounts of weights, prediction took around 4 ms, which was not necessarily faster than the execution of one operation on the system-under-test.

The consequence of this performance issue was to slow down the broker and therefore also to create a more realistic testing environment. IoT applications usually run on small and energy efficient hardware. To create an environment comparable to this, we slowed down the the virtual machine, running the broker, to 1 – 10% of the CPU resources of the test server. This modification did not only give us a more realistic testing environment, it also helped to get rid of the time dependency for EMQ and VerneMQ. The errors, which occurred because of scheduling etc., became really small compared to the latencies of the broker and could therefore be neglected.

The CPU limitations also made the neural network perform really good and to reach an R^2 value of over 0.9 for the EMQ dataset. The architecture of the neural network was kept as simple as possible and the preprocessing steps do not require any human interaction and therefore can be performed automated. The integration of our deep learning cost model made it possible to make performance testing even easier and the prediction of the performance using a model simulation is faster than to run the performance tests on the SUT.

5.1 Outlook

In the future, we plan to completely automate the process of training the cost model using the created log-data and to integrate the learned model into the test system. The

evaluation of other test scenarios and datasets, like the influence of the network etc., is also a very interesting area we want to step into.

To speed up the prediction time even more, the neural network can be implemented in C and optimized to be more efficient. Those optimizations however, take a lot of effort to be implemented and the result is not expected to be dramatically faster than the current method of serving, using the standard input-output of the prediction process we wrote.

The ability for the neural network to be more precise in extrapolation could also be investigated. To achieve this, stronger regularization methods within the network can be used. It would also be interesting to use a Bayesian neural network to get uncertainty estimates of the prediction, which would also be a good indicator of how well the neural network performs.

Bibliography

[1] B. K. Aichernig and R. Schumi, "Statistical model checking meets property-based testing," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pp. 390–400, IEEE, 2017.

[2] R. Schumi, P. Lang, B. K. Aichernig, W. Krenn, and R. Schlick, "Checking response-time properties of web-service applications under stochastic user profiles," in *IFIP International Conference on Testing Software and Systems*, pp. 293–310, Springer, 2017.

[3] B. K. Aichernig and R. Schumi, "How fast is MQTT? statistical model checking and testing of iot protocols," in *Quantitative Evaluation of Systems – 15th International Conference, QEST 2018*, Lecture Notes in Computer Science, Springer, 2018.

[4] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *International conference on runtime verification*, pp. 122–135, Springer, 2010.

[5] A. Wald, "Sequential tests of statistical hypotheses," *The annals of mathematical statistics*, vol. 16, no. 2, pp. 117–186, 1945.

[6] B. K. Aichernig and R. Schumi, "Property-based testing with FsCheck by deriving properties from business rule models," in *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference*, pp. 219–228, IEEE, 2016.

[7] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[8] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating feasible transition paths for testing from an extended finite state machine (efsm)," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pp. 230–239, IEEE, 2009.

[9] P. Ballarini, N. Bertrand, A. Horváth, M. Paolieri, and E. Vicario, "Transient analysis of networks of stochastic timed automata using stochastic state classes," in *International Conference on Quantitative Evaluation of Systems*, pp. 355–371, Springer, 2013.

[10] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[11] B. K. Aichernig and R. Schumi, "Towards integrating statistical model checking into property-based testing," in *Formal Methods and Models for System Design (MEMOCODE), 2016 ACM/IEEE International Conference on*, pp. 71–76, IEEE, 2016.

[12] A. Banks and R. Gupta, "MQTT version 3.1. 1," *OASIS standard*, vol. 29, 2014.

[13] K. Backhaus, B. Erichson, W. Plinke, C. Schuchard-Ficher, and R. Weiber, *Multivariate analysemethoden: eine anwendungsorientierte einführung*. Springer-Verlag, 2013.

[14] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, Springer New York, 2 ed., 2009.

[15] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in human neuroscience*, vol. 3, p. 31, 2009.

[16] J. J. Hopfield and D. W. Tank, ""neural" computation of decisions in optimization problems," *Biological cybernetics*, vol. 52, no. 3, pp. 141–152, 1985.

[17] C. M. Bishop, *Pattern Recognition and Machine Learning*. Information Science and Statistics, Springer, 2006.

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[20] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[21] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[22] L. Deng, D. Yu, *et al.*, "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[24] "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program, redrawn by User:Dhp1080 / CC-BY-SA-3.0 / GFDL, "File:neuron.svg." `https://commons.wikimedia.org/wiki/File:Neuron.svg`. Accessed: 2018-07-30.

[25] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

[26] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions." `https://openreview.net/forum?id=SkBYYyZRZ`, 2018.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[29] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 IEEE international conference on*, pp. 6645–6649, IEEE, 2013.

[30] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[31] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[32] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[33] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, 1991.

[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[36] C. Olah, "Understanding LSTM networks." `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2018-07-21.

[37] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[38] B. K. Aichernig, S. Kann, and R. Schumi, "Statistical model checking of response times for different system deployments," in *Dependable Software Engineering. Theories, Tools, and Applications – 4th International Symposium, SETTA 2018, Beijing, China, Sep. 4-6, 2018*, Lecture Notes in Computer Science, Springer, 2018.

[39] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *Journal of Open Source Software*, vol. 2, no. 13, 2017.

[40] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[42] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[43] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[44] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.