CHALMERS
UNIVERSITY OF TECHNOLOGY

Reconstructing Missing Data in Event Logs using Deep Learning Methods

Master's thesis in Computer Science - Algorithms, Language and Logic

LINDA HAMP
OSKAR JÖNEFORS

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018
EX016/2018

Master's thesis EX016/2018

Reconstructing Missing Data in Event Logs using Deep Learning Methods

LINDA HAMP
OSKAR JÖNEFORS

Company supervisor:	Ian Hellström
University supervisor:	Irene Yu-Hua Gu, Department of Electrical Engineering, Chalmers University of Technology
Examiner:	Irene Yu-Hua Gu, Department of Electrical Engineering, Chalmers University of Technology

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Reconstructing Missing Data in Event Logs using Deep Learning Methods
LINDA HAMP
OSKAR JÖNEFORS

Company supervisor:	Ian Hellström
University supervisor:	Irene Yu-Hua Gu, Department of Electrical Engineering, Chalmers University of Technology
Examiner:	Irene Yu-Hua Gu, Department of Electrical Engineering, Chalmers University of Technology

Master's Thesis EX016/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Reconstructing Missing Data in Event Logs using Deep Learning Methods
LINDA HAMP
OSKAR JÖNEFORS
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Advances in computational power and deep learning methods enable many industries to use their data to gain more meaningful insights than previously possible. The data is commonly found in event logs from software systems where events are delivered via network connections. Consequently, the deliveries could be adversely affected by network congestion or weak signals. Unless care is taken to resend data that is not successfully delivered, events can be lost.

In this project, the aim is to reconstruct missing events using recurrent neural networks. The models evaluated are a bidirectional recurrent neural network (BRNN) and a sequence-to-sequence model. Deep learning methods have previously been used to reconstruct missing data in sequences, but the locations of the gaps have then been declared in the input and are thus known in advance by the models. In this project, deep learning models have been trained to reconstruct incomplete sequences where the locations of missing events are not given in advance. This makes them more useful for real applications.

In our proposed method, the models are trained using supervised learning with complete sequences as target output. The input consists of both incomplete and complete sequences, where the incomplete sequences are constructed by removing events from complete ones. In this project, the dataset comprises event data provided by the company where this thesis was conducted, and contains events related to the navigation of their application.

Our findings are that the sequence-to-sequence model succeeds in reconstructing 75.42% of all missing events with a 95% confidence interval of $[74.81\%, 76.02\%]$, whereas the BRNN is only able to reconstruct 64.99% of all missing events, with a 95% confidence interval of $[64.16\%, 65.83\%]$. From this we conclude that both models can be successfully used for the task of event reconstruction, with the sequence-to-sequence model being the best candidate.

Keywords: deep learning, bidirectional recurrent neural networks, sequence-to-sequence model, event logs, reconstruction.

Acknowledgements

First of all we would like to thank our company supervisor, Ian Hellström, for all the valuable support he has provided throughout the project. Thank you for your tireless commitment, your contributions and for guiding us through all datasets and systems. Without you, this project would not have been possible.

We would also like to thank our manager Sofia Lindberg, for taking time out of her busy schedule to support us. Additional thanks to all other co-workers at the company's office in Gothenburg for making us feel welcome and included.

Finally, many thanks to our supervisor and examiner Irene Yu-Hua Gu at the Electrical Engineering department at Chalmers University of Technology for her helpful guidance and feedback throughout the project.

Linda Hamp and Oskar Jönefors, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1

Introduction

In many software systems, event logs are used to store a record of activities occurring during operations [1]. Collecting logs enables system owners to, for instance, debug systems and analyze system failure. To ensure reliability of the analyses, the event logs should be as complete as possible. In fact, Cinque et al. [2] state that when conducting failure analysis, the reliability of the analyses depends on the accuracy of the event logs.

In network transmission, data can be lost due to network congestion [3] or transmission errors suffered by network links [4]. In cases where events are sent over networks without a handshaking procedure, such events may be silently dropped. This results in an incomplete or inconsistent record of events, which can lower the credibility of statistical analyses of a dataset. If missing events could be recovered they could improve statistical analyses and thereby help the system owners to maintain and develop quality products.

1.1 Background

The problem of reconstructing incomplete or corrupt sequences can be found in many areas, and several approaches have been used to solve it.

The *noisy channel model* introduced by Shannon [5] has among other problems been used in spelling correction [6]. Given a word or n-gram of words (a sequence of n words) which may have been modified in some way (via deletions, substitutions or insertions of characters) when passed through a *noisy channel*, it models the probabilities of the most likely intended word or n-gram.

Rule-based systems are built to incorporate human knowledge into a set of if-then rules. Their ability to evaluate contextually relevant rules makes them able to solve

a variety of different tasks [7]. A requirement for rule-based systems to perform well is that the necessary human expertise within the problem area can be translated into if-then rules [8].

Hidden Markov Models (HMMs) are probabilistic graphical models with hidden states [9]. They are commonly used to model sequential data, and can be applied in areas such as pattern recognition and signal processing. An advantage of HMMs is that it is possible to use a forward-backward algorithm to predict a state at any given time [10].

Recurrent neural networks (RNNs) have been used to model time series and to generate sequential data. For instance, Berglund et al. [11] used a bidirectional RNN (BRNN) to fill in missing data in time series with complex dynamics. However, in their research, the locations of missing data in sequences were known beforehand. The network was trained on sequences that in place of the removed data points contained special missing tokens, meaning that the network had to reconstruct missing data but not predict its location in the sequences. In this project, the locations of gaps in sequences are not stated in the input data.

The approach chosen for event reconstruction in this thesis is to use RNN-based models. The noisy channel model cannot be adapted to event reconstruction since it uses a dictionary of valid words or n-grams of words as a basis, and the number of possible valid sequences for the event data used in this thesis is too large to allow their efficient computation. Rule-based systems are, as previously mentioned, built on if-then rules. Precisely capturing the dynamics of event data from a software application in a rule-based system would essentially require a reimplementation of the logic of the application, which is too cumbersome to be considered for this project. HMMs could be applied to the problem, but have been used extensively for similar problems, whereas RNNs are less studied in the context of event reconstruction and are therefore considered a more interesting option.

1.2 Goal

The goal of this project is to reconstruct missing data in event logs by using RNNs. The models are fed with both complete and incomplete sequences and should reconstruct missing events where they determine that input sequences are incomplete. The success rate of this project is evaluated on how well the network succeeds in reconstructing missing events.

1.3 Approach

The ability to reconstruct events could vary between neural network architectures. Therefore, the performance of two different models are evaluated in this project, namely a BRNN and a sequence-to-sequence model. A BRNN is the network construction used by Berglund et al. [11], mentioned in Section 1.1, and this project will extend their work by investigating the success rate of a BRNN that both has to predict where data is missing and to fill in the gaps. Sequence-to-sequence models have previously been used when translating one language into another, but to the best of our knowledge, no one has so far used sequence-to-sequence modelling on incomplete event logs.

In order to fill in missing data in event logs, the models must also be able to produce complete sequences. They are therefore trained using supervised learning on both complete and incomplete sequences. However, to be able to perform supervised learning, complete target sequences must be available for all input sequences. Therefore, incomplete sequences are constructed by removing events from complete ones. The models are then trained on their ability to reconstruct these events.

The networks are implemented using Python, and the underlying library is the popular TensorFlow.

1.4 Scope

This project has been conducted at a company providing an application for desktop and mobile platforms. The dataset used in this thesis is not public, but consists of event logs provided by the company. These are logs of user interactions, which can be used to analyze user behavior and evaluate the effectiveness of, for instance, the navigational structures of the company's application. Events are sent over the internet to the company's servers from the application, and certain types of events are not always guaranteed to arrive.

2

Theory and background

This chapter aims to describe *deep learning* and the deep artificial neural network techniques required to fully understand how this project was conducted. We briefly review the general concepts in deep learning, followed by a more detailed description of the two models used in this project, a BRNN and a sequence-to-sequence model. In addition, concepts such as *feed-forward neural networks*, *LSTM* and *GRU* are explained, some common problems of artificial neural networks are pointed out, and possible solutions to these problems are presented.

2.1 Artificial Neural Networks and Deep Learning

Inspired by the complex networks of neurons found in human and animal brains, artificial neural networks (ANNs) are computational models consisting of interconnected units, so-called neurons, structured in layers. The typical structure consists of one input layer and one output layer, with a varying number of hidden layers in between. Data is fed to the input layer and then transmitted through the network to the output layer. The connections between neurons are assigned weights which amplify or reduce the transmitted signals. By adjusting these weights, the output of the network is altered.

To be more precise, a neuron n_i receives a set of inputs $\mathbf{x} = \{x_1, ..., x_n\}$ that are passed together with a *bias* term of the neuron through an activation function f that produces the output of the neuron. This output is then multiplied by the outgoing weights of the neuron when passed on to the next layer of neurons. Some common activation functions are the non-linear *sigmoid*, *logistic* and *tanh* functions, or the *Rectified Linear Unit* (ReLU) which is defined as $\max(0, x)$. A frequently used activation function for the final layer of neural classifiers is *softmax*, which squeezes the values in its input vector between 0 and 1 and produces an output vector where

all values add up to one. In other words, it creates a valid probability distribution from its input.

One of the more typical structures of neural networks is the Feed-forward Neural Network (FFNN). FFNNs are networks where all outputs of neurons lead to neurons in subsequent layers. Consequently, they contain no cycles.

ANNs with few layers have been around for decades (McCulloch and Pitts [12] introduced an early version of an artificial neuron in 1943). However, a functional multi-layered structure – so called *deep neural network* – has been harder to achieve. Artificial neural networks were well studied during the 1990s, but the concept of very deep networks was not relalised until the 21st century [13]. Today, deep learning architectures often outperform conventional neural networks and have been proven to be particularly successful in pattern recognition.

2.2 Training of neural networks

To improve the performance of ANNs, they undergo a training procedure. When using ANNs for problems where the desired output is known, so-called *supervised learning problems*, the weights in the networks are adjusted to minimize the value of a *loss function*. The loss function measures the degree of error between the desired output and the actual output of the network. A common way to measure the degree of error between two sequences is a metric called the *Levenshtein distance*, also known as the *edit distance*. This metric is often used to compare strings and measures the difference between two sequences by calculating the number of edits (insertions, deletions or substitutions) needed to transform one sequence into another. A loss function based on such a metric could prove highly useful in solving the problem posed in this paper. However, the Levenshtein distance is non-differentiable and TensorFlow is not able to perform optimization on it directly.

The data used in this project mainly comprises of categorical features, which makes the *cross-entropy* (CE) a suitable choice of loss function. The concept of the CE method was first proposed by Rubinstein [14] in rare-event simulation. It measures the divergence between a measured probability distribution and an expected one. Soon after its introduction, CE was adjusted to become a randomized objective function [15] and can nowadays be used to for instance solve classification problems.

The CE method is shown in (2.1), where y is the target output and $\hat{y}$ is the predicted output from the network.

$$H(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i) \tag{2.1}$$

To minimize the gradient of the loss function with respect to the weights, *gradient descent* is commonly used. The gradient is usually computed using the *backpropagation* algorithm.

Gradient descent is an optimization algorithm used to find the minimum of a function. This is done by taking steps in the negative gradient direction, as seen in (2.2), where $\Delta\mathbf{a}$ denotes the change in input from one step of the algorithm to the other, η is the step size and ∇F the gradient of the function F. Similarly, the modification of a weight w_{ij} in a neural network is shown in (2.3), where $\frac{\partial E}{\partial w_{ij}}$ is the partial derivative of the error E with respect to the weight w_{ij}.

$$\Delta\mathbf{a} = -\eta\nabla F(\mathbf{a}) \tag{2.2}$$

$$\Delta w_{ij} = \eta\frac{\partial E}{\partial w_{ij}} \tag{2.3}$$

Gradient descent in the context of neural networks can be used in a number of ways. Batch gradient descent computes the gradient of the loss function for an entire dataset, and can therefore be slow when dealing with large amounts of data. Stochastic gradient descent computes the gradient of a single training example in each update. Consequently, updates can be computed very quickly and the algorithm avoids getting stuck in local minima. Mini-batch gradient descent combines the strengths of the previous two approaches and computes the gradient for small batches of training examples.

A very important detail in gradient descent is the learning rate. Choosing a learning rate that is too small will result in a very slow convergence, and a learning rate that is too large may result in the algorithm overshooting the target, fluctuating around a minimum but not quite reaching it. To combat these challenges, many popular optimizing algorithms modify the learning rate throughout the optimization process by means such as learning rate decay or momentum [16, 17].

2.2.1 Overfitting

A common problem associated with neural networks is their tendency to *overfit*. Overfitting occurs when a model does not succeed in generalizing from a specific dataset. As a result, the model performs extremely well on training data but fails miserably when applied to a validation or test set. One of the regularization techniques commonly used to prevent overfitting is *dropout* [18]. When applying dropout, units and their connections are randomly and temporarily excluded from the network during the training phase [19], see Figure 2.1. Removing units makes the network more robust as it decreases the risk of neurons developing co-dependencies, which is one common cause of overfitting.

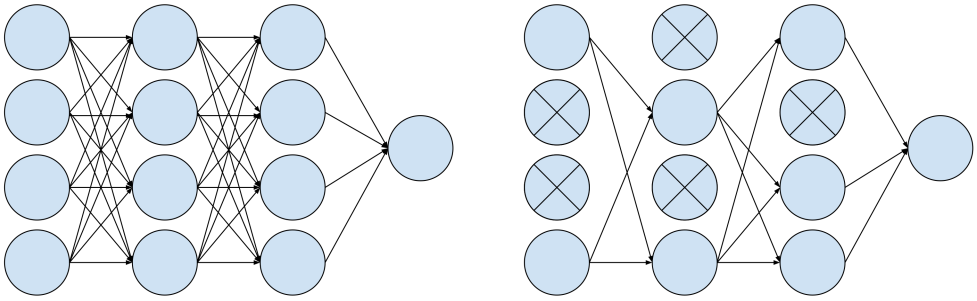


Figure 2.1: Two ANNs: the left one without dropout and the right one with dropout applied.

Another regularization technique used to prevent overfitting in neural networks is called *early stopping*. Simply put, early stopping means that the training phase is interrupted when the performance of the network has not improved for a certain number of training steps.

2.3 Recurrent Neural Networks

A common deep learning method is the recurrent neural network (RNN). In contrast to FFNNs, RNNs are neural networks that contain cycles and accordingly map all previous inputs to every subsequent output instead of just the current one [20]. This construction allows RNNs to possess an internal memory structure not found in FFNNs. RNNs do not restrict the input and output lengths as FFNNs do, but allow them to vary. As a result, RNNs are superior to FFNNs when it comes to sequential input data, such as time-varying patterns. A simple illustration of an RNN structure is displayed in Figure 2.2.

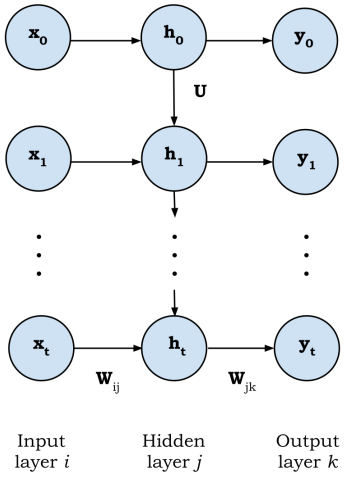


Figure 2.2: RNN structure with one hidden layer unrolled over t time steps.

Equation (2.4) describes an update of a hidden state in an RNN, where $\mathbf{h}_t$ is the hidden state for time t, $\mathbf{x}_t$ the network input vector at time t, $\mathbf{h}_{t-1}$ the hidden state for the previous input, $\mathbf{W}_{ij}$ the weight matrix for the input, $\mathbf{U}$ the weight matrix for the hidden state, $\mathbf{b}$ the bias and f the activation function. $\mathbf{h}_0$ is often defined as a constant.

$$\mathbf{h}_t = f(\mathbf{W}_{ij}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}) \tag{2.4}$$

The network output is given in (2.5), where $\mathbf{y}_t$ is the output at time t, $\mathbf{W}_{jk}$ the weight matrix between the hidden layer in the network and the output layer, $\mathbf{h}_t$ the hidden state at time t, $\mathbf{b}$ the bias and f the activation function.

$$\mathbf{y}_t = f(\mathbf{W}_{jk}\mathbf{h}_t + \mathbf{b}) \tag{2.5}$$

As illustrated by (2.4), the new hidden state at each time step depends on the hidden state of the previous time step. In other words, when training a network, subsequent multiplications are made to calculate the states of the network. Two problems associated with these calculations are *vanishing gradients* and *exploding gradients*, meaning that the calculated derivatives become very small or very large. As a result, the weights of the network are either not updated at all, or the network becomes unstable due to extremely large weight updates. *Gradient clipping* can be used to avoid exploding gradients [21]. The gradients are then simply cut off at

some threshold value, which prevents them from becoming too large. Additional ways of reducing the gradient problems are RNN mechanisms such as *long short-term memory* or the *gated recurrent unit*.

2.3.1 Long Short-Term Memory

Long short-term memory (LSTM) is a special type of RNN cell. Proposed in 1997 by Hochreiter and Schmidhuber [22] as a way of mitigating the problem of vanishing or exploding gradients, LSTM maintains an internal memory cell state $\mathbf{c}$ which is used in combination with the input when updating the hidden state of the unit. Avoiding vanishing or exploding gradients enables the LSTM unit to capture long-term dependencies that standard RNNs are ill-equipped to handle. One limit of the standard LSTM design is that cell states may grow in an unbounded fashion when presented with a continuous input stream [23]. To counter this problem, the LSTM model was improved by Gers et al. [23] with the addition of a forget gate that enables the cell to reset its state at appropriate times. In this thesis, only LSTMs with forget gates are considered.

The main components of an LSTM unit are the forget gate, the input gate, the output gate, and most importantly the memory cell. They are used in conjunction to produce the final output of the unit, and the three gate activations determine what to forget from the previous state, which values to update in the cell, and which values to emit as the new unit output. The output $\mathbf{h}_t$ of the LSTM unit at time t is calculated as seen in (2.6a), where $\mathbf{o}_t$ is the output gate activation, $\mathbf{c}_t$ is the current cell state and $\odot$ denotes element-wise multiplication.

The output gate activation $\mathbf{o}_t$ in (2.6b) is a vector of values between 0 and 1, and decides to which extent the various parts of the cell state contributes to the final output. It is produced by adding a bias term $\mathbf{b}_o$ to the current unit input $\mathbf{x}_t$ and the previous unit output $\mathbf{h}_{t-1}$ scaled by their respective weight matrices $\mathbf{W}_o$ and $\mathbf{U}_o$. This is passed through the sigmoid function to scale the output vector components between 0 and 1.

The current cell state vector $\mathbf{c}_t$ in (2.6c) comprises two terms. The first determines what will be remembered from the previous cell state, and is produced by filtering the previous cell state with the forget gate activation $\mathbf{f}_t$ in (2.6d). The second term

combines the current input with the previous output and filters it with the input gate activation $\mathbf{i_t}$ shown in (2.6e).

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{2.6a}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.6b}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.6c}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.6d}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.6e}$$

2.3.2 Gated Recurrent Unit

The gated recurrent unit (GRU) was first introduced by Cho et al. [24] and is a simpler version of the LSTM. A GRU cell provides the same advantages as an LSTM cell, with an effective long-term memory and ability to solve the problems with vanishing or exploding gradients, but its simpler structure makes it easier than the LSTM to both implement and compute. While the LSTM has three gates: the forget gate, the input gate and the output gate, the GRU has only two: the update gate and the reset gate. The GRU also lacks the memory cell that is an essential component of the LSTM.

The main formula for a GRU is described in (2.7a), where $\mathbf{h}_t$ is the output from the GRU, $\tilde{\mathbf{h}}_t$ the candidate update in (2.7b) and $\mathbf{u}_t$ is the update gate. The update gate is calculated according to (2.7c), with σ being the sigmoid function taking on values between 0 and 1. The reset gate $\mathbf{r}_t$ illustrated in (2.7d) is similar to the update gate. The difference between the two gates is that the update gate decides what to read of the candidate update, and the reset gate is selecting which part of the hidden state to overwrite.

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t \tag{2.7a}$$

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}\mathbf{x}_t + \mathbf{U}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}\right) \tag{2.7b}$$

$$\mathbf{u}_t = \sigma(\mathbf{W}_u \mathbf{x}_t + \mathbf{U}_u \mathbf{h}_{t-1} + \mathbf{b}_u) \tag{2.7c}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r) \tag{2.7d}$$

The new hidden state for each time step is calculated based on the previous hidden state and the new input. However, sometimes it can be more useful to carry information from even further back to calculate a new hidden state. While multiplication along every time step in a regular RNN causes the network to only remember a few time steps, carrying information from further back in a GRU creates a longer memory structure. If the update gate has values close to 1, a new hidden state will be calculated, while values close to 0 means the previous state is kept. If the reset gate has values close to 0, the previous hidden state is thrown away and a new hidden state is calculated based on the new input.

2.4 Bidirectional Recurrent Neural Networks

RNNs process the input one time step at a time, continuously producing output based on previous time steps. However, for problems where input data past the current state is readily available (unlike time-series prediction where future events are unknown), the network performance could be improved if the network had access to both previous and subsequent inputs at each time step. Bidirectional recurrent neural networks (BRNNs), first proposed by Schuster and Paliwal [25], consist of two RNNs that read input from two different directions and then merge the results (see Figure 2.3). By doing so, available information both before and after the current time step can be taken into account when producing the output.

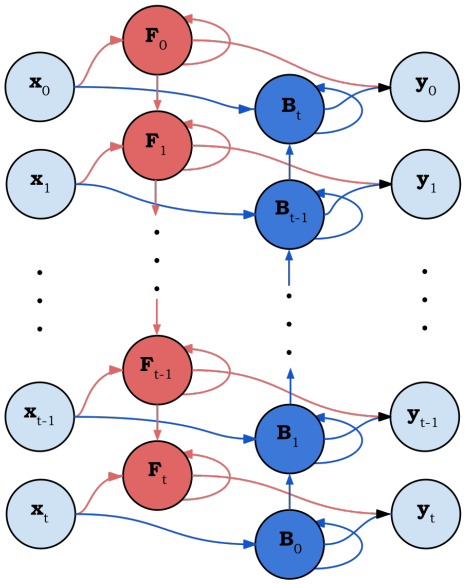


Figure 2.3: Simple BRNN structure with one bidirectional layer, unrolled over t time steps. The forward layer is in this figure illustrated in red, and the backward layer in dark blue.

2.5 Sequence-to-sequence models

One network construction suitable for mapping one sequence to another is the sequence-to-sequence model. This model was first introduced for machine translation and consists of two neural networks combined: an *encoder* and a *decoder* [24, 26]. The encoder takes an input sequence and outputs a feature vector. The decoder takes the feature vector and generates an output sequence token by token by taking its own output from the previous time step as input. Since the training and validation processes for a sequence-to-sequence model are slightly different, they are both illustrated in Figures 2.4 and 2.5.

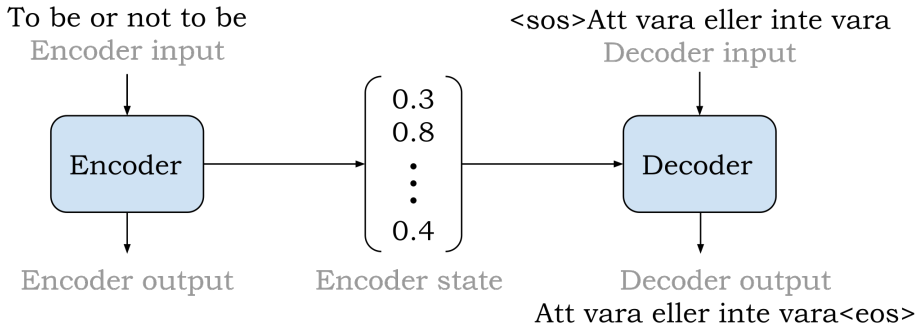


Figure 2.4: The training process of a sequence-to-sequence model translating an English sentence into a Swedish one. The English sentence is fed to the encoder one word at a time. After the whole sentence has been passed through the encoder, the resulting internal state of the encoder is used as a feature vector to represent the sentence. The decoder takes two inputs: the feature vector from the encoder and the expected output sentence. That is, the final internal state of the encoder becomes the initial state of the decoder. The target sentence is also fed to the decoder word by word, starting with a start-of-sequence token. The decoder then produces an output sentence word by word ending with an end-of-sequence token.

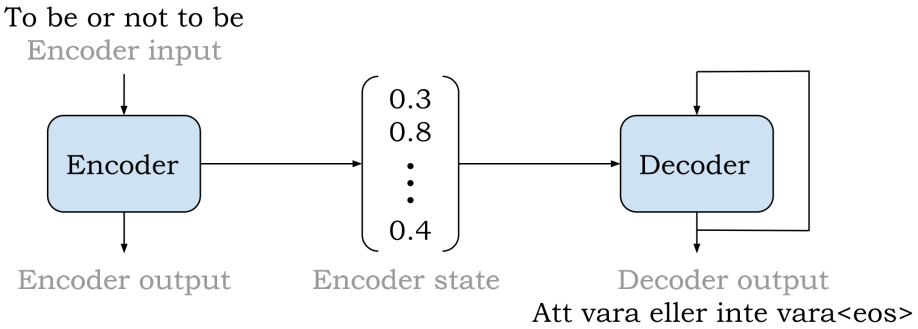


Figure 2.5: The validation process of a sequence-to-sequence model translating an English sentence into a Swedish one. The difference between the validation and training process is that the decoder is not given the target sentence as input. Instead, the decoder takes the last word it generated and uses as input for the next time step.

2.5.1 Attention

Although the sequence-to-sequence model performs well on short sequences, Cho et al. [27] show that the performance gradually decreases when the number of unknown input tokens and sequence lengths increase. The main reason for the worsened performance is that a sequence-to-sequence model relies on the encoder to compress all important aspects of the input sequence into a feature vector of fixed length. As the lengths of input sequences grow, their content becomes increasingly hard to capture in the fixed-size feature vector.

One mechanism devised to remedy the situation is called *attention*. At each time step, attention gives the decoder access to parts of the input sequence through *context vectors* [28]. A context vector $\mathbf{c}_i$ is calculated as described in (2.8a), where a_{ij} is a weight, $\mathbf{h}_j$ is the hidden state of the encoder at time step j, and n is the number of hidden states in the encoder. Weights are calculated according to (2.8b), where e_{ij} is an alignment measure, calculated from an alignment model a in (2.8c). The alignment model measures to which extent the input around index j corresponds to the output at position i. That is, the weight a_{ij} is a probability for how well the input x_j corresponds to the output y_i, and $\mathbf{s}_{i-1}$ is the state of the decoder at time step *i-1*.

$$\mathbf{c}_i = \sum_{j=1}^{n} a_{ij}\mathbf{h}_j \tag{2.8a}$$

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n}\exp(e_{ik})} \tag{2.8b}$$

$$e_{ij} = a(\mathbf{s}_{i-1}, \mathbf{h}_j) \tag{2.8c}$$

The decoder can then use the context vector $\boldsymbol{c}_i$ along with its so far generated output to produce its ith output. Correspondingly, the decoder does not have to rely solely on the last hidden state from the encoder when producing an output sequence, but can also incorporate information from the context vectors at each time step.

3

Methods

This chapter begins with a description of the datasets used in this project, both real and synthetic. The architectures of the BRNN and the sequence-to-sequence model are then shown, followed by the loss function used during training. The fixed and adjusted hyperparameters of the models are stated followed by a brief outline of the training process. Finally, the metrics constructed to evaluate the results are explained.

3.1 Datasets

Two main datasets are used in this project. One contains the real event data provided by the company where this thesis was conducted, and the other consists of synthetic data generated in order to test the performance of the models on simpler data. The size of both datasets, the number of complete and incomplete sequences they contain, and the maximum sequence length of each dataset are illustrated in Table 3.1. The percentage split between the training, validation and test set were 70/20/10 for both datasets.

Table 3.1: Illustration of the number of sequences, the split between complete and incomplete sequences, and maximum sequence length for the two datasets used in this project.

Dataset	**Number of sequences**			**Maximum sequence length**
	Total	**Complete**	**Incomplete**	
Synthetic data	10,000	5,000	5,000	100
Real event data	100,000	50,000	50,000	136

3.1.1 Real event data

The logs of user interactions collected by the company where this thesis was conducted contain information about the navigational patterns of users in the application. Such information can be used to for instance improve the navigational structure of the application based on user behavior. The number of different event types are many, but only a few are investigated and used in this project. The events have some attributes in common, but also contain information specific to each event type. An example of a common attribute present in a majority of events is a timestamp. Based on the timestamp, events can be ordered chronologically. The partition of sequences in this project is based on so-called sessions in the event logs. A session is a record of events occurring during a specific period of time. Sessions can be of different lengths and contain different types of events.

The event types of interest for this project have in this thesis been called *PageEnterEvent*, *PageExitEvent* and *InteractionEvent*. The PageEnterEvent represents the action of entering a page in the application, and the PageExitEvent represents exiting it. The InteractionEvent represents an interaction between a user and a page, which could result in leaving one page to enter another one – when tapping a link for instance – or staying on the same page. There are two distinct patterns in these sessions. First, two PageEnterEvents should not occur without a PageExitEvent in between them. That is, when entering a new page, one first has to exit the previous page. Second, an InteractionEvent has to occur before a PageExitEvent, since it is not possible to exit a page without interacting with the application first.

As previously mentioned each event type contains multiple attributes. All of these are not used in this project and will therefore not be mentioned in this thesis. However, in order to understand the metrics used to evaluate the results, the attributes used in this project need to be named. The core attributes are *type*, *page* and *time*. Type describes what kind of event it is, page describes what page each event is connected to, and time when the event occurred (the previously mentioned timestamp). The InteractionEvent also contains information of how the user taps or clicks in the application and what the intention of the interaction is. These two attributes are in this thesis called *interaction type* and *intent*. For clarification, all three event types and their attributes are illustrated in Table 3.2.

Table 3.2: Definitions of the event types in the real event dataset.

Event type	Attributes		User action
	Categorical	**Scalar**	
PageEnterEvent	Type, Page	Time	Entering a page
InteractionEvent	Type, Page, Interaction type, Intent	Time	Interacting
PageExitEvent	Type, Page	Time	Exiting a page

Events of the same type always contain the same attributes, but the attributes can vary between events of different types. In the session illustrated in (3.1), all events contain the attributes type, page and time. The InteractionEvent also contains interaction type and intent, while the PageEnterEvents and PageExitEvents do not. The size of each event, meaning the number of attributes it contains, can also vary. As shown in (3.1), the PageEnterEvent and PageExitEvent are of the same size while the InteractionEvent has one attribute less than the other two.

$$\begin{bmatrix} PageEnterEvent & Page1 & Time1 & < Attribute > & < Attribute > & < Attribute > \\ InteractionEvent & Page1 & Time2 & InteractionType1 & Intent1 & \\ InteractionEvent & Page1 & Time3 & InteractionType1 & Intent2 & \\ PageExitEvent & Page1 & Time4 & < Attribute > & < Attribute > & < Attribute > \\ PageEnterEvent & Page2 & Time5 & < Attribute > & < Attribute > & < Attribute > \end{bmatrix} \tag{3.1}$$

The lengths of sessions used in this project range between 6-136 events. Outlier sessions with atypically few or numerous events have been excluded from the data, as sessions of negligible length leave little room for predictions and sessions of extreme length would waste a lot of computational power when training the models. The number of distinct event types is five, namely a start-of-sequence token, the PageEnterEvent, the InteractionEvent, the PageExitEvent and an end-of-sequence token. The number of distinct pages is much larger than the number of event types. To create the input sequences, half of the complete sequences were kept intact, whereas events were dropped with a 10% probability in the other sequences. To be able to train both the BRNN and the sequence-to-sequence model within the restricted time frame of this project, the number of sequences was limited to 100,000.

3.1.2 Synthetic data

Since the real event data is rather complex, much simpler synthetic data was generated and used as the input in the early stages of the network building phase. The first version of synthetic input data consisted of 10,000 single-feature event sequences, which is ten times fewer sequences than the real event dataset contains. We reason that to account for the complexity of the real event data and enable the models to learn its dynamics, the size of that dataset had to be substantially larger than the simpler synthetic dataset. The lengths of sequences in this first version of the synthetic dataset ranged between 6-30 events, where each event contained no other information than its type. An example of a sequence consisting of such events is illustrated below:

$$E, I, I, X, E, I, I, X, E, I, X, E, I, I, X, E, I, X \tag{3.2}$$

As displayed, the sequences contained three different event types. The E event is a very simple version of the PageEnterEvent described in Section 3.1.1 since it represents the event type but has no additional attributes. Similarly, the I event is a simplification of the InteractionEvent and X a simplification of the PageExitEvent. The simplified events are shown in Table 3.3.

Table 3.3: Definitions of the event types in the synthetic dataset.

	Attributes		
Event type	**Categorical**	**Scalar**	**User action**
E	Type, Page	Time	Entering a page
I	Type, Page	Time	Interacting
X	Type, Page	Time	Exiting a page

In order to adapt the categorical input to the neural networks, it was transformed into one-hot vectors. A one-hot vector is a vector with a size equal to the number of categorical classes, each index representing a class. Each index can take on the values of either 0 or 1 and only one index can be set to 1 at the time. An illustration of a matrix of one-hot vectors for the three event types E, I and X is given below:

$$E, I, X \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

The complexity of the synthetic input data was increased by adding additional features, first a time variable and later the ID of the page that the event occurred on.

In the real application, pages have unique elements and properties, meaning not all interaction types or intents are possible on all pages. Furthermore, the time users spend on any given page depends on the properties of the page.

For the synthetic data, a navigational structure of pages with distinguishing features was deemed unnecessarily complex and thus no such navigational structure was incorporated into the data generating process, and the number of pages was limited to five.

Since interaction type and intent depend on a navigational structure, they were not added to the synthetic events. Accordingly, the synthetic data only contains event types, page IDs and timestamps. The generated timestamps are evenly spaced between 0 and 1 for each sequence. This allows models to at least learn the regular spacing of events. A sequence of synthetic events is illustrated in 3.4, where the event structure is $(Type, Page, Time)$.

$$(E, 0, 0.00), (I, 0, 0.17), (X, 0, 0.33), (E, 4, 0.50), (I, 4, 0.67), (I, 4, 0.83), (X, 4, 1.00) \tag{3.4}$$

Each event was transformed into a one-hot vector with 11 indices. The first five indices represent a start-of-sequence token, the E event, the I event, the X event and an end-of-sequence token. The next five indices hold the different page classes, and the final index contains the continuous time variable. Note that the number of pages in the synthetic data is very low compared to the number of pages in the real event data. In the sequence illustrated by the matrix below, the user enters page 1, taps or clicks something to switch page, and then the page changes from page 1 to page 2.

$$(E, 1, 0.00), (I, 1, 0.33), (X, 1, 0.67), (E, 2, 1.00) \Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0.00 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0.33 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0.67 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1.00 \end{bmatrix} \tag{3.5}$$

As mentioned in Section 3.1.1, the lengths of sequences in the real event data range between 6-136 events. The sequences in the first version of the synthetic dataset were kept quite short (6-30 events per sequence) and were only intended as a way to test the basic network structures. When increasing data complexity by adding multiple features, two different datasets were created – one with short sequences and one with longer sequences – to be able to evaluate the performance of the models on data of varying complexity. Each synthetic dataset contained 10,000 sequences, but the sequence lengths in one dataset ranged between 6-30 events, and the other between 20-100 events, which is similar in length to the real user sessions. The evaluation of model performance on simpler and more complex data was conducted on the synthetic dataset consisting of sequences of lengths 20-100 events, and the real event dataset described in Section 3.1.1.

When creating the input sequences, 50% of the complete sequences were kept intact. From the other sequences, E and X events were removed with a 10% probability. Since the synthetic data lacks a navigational structure, where some pages could be inaccessible from others, no I events were removed. If a sequence of E, I and X events related to the same page were removed, the model could therefore never deduce that a whole page was missing in the resulting sequence.

3.2 Description of models

Two models are chosen for this project, a BRNN model and a sequence-to-sequence model. In this section, an overview of the architectures of these models is given, along with descriptive figures. For simplicity, the figures depict networks with only one hidden layer, whereas the number of layers for both models was adjusted iteratively to find the best performing model for event reconstruction. Likewise, only LSTM cells are shown. However, GRUs were also evaluated to ascertain the best cell structure for the task.

3.2.1 Bidirectional Recurrent Neural Network

Since the project entailed generation of sequences of variable length, a recurrent network structure was a natural choice. A BRNN was considered a more suitable model to reconstruct events than a regular RNN since BRNNs process input sequences from two different directions. For instance, if a certain event must always be produced during a user's visit on a particular page, a BRNN could refrain from

producing it prematurely if knowing that the event in question is present in a subsequent time step of the input data. It is likely that both detecting that an event is missing and reconstructing that event is easier done if the network has access to events both before and after the gap(s). The structure of the BRNN is illustrated in Figure 3.1.

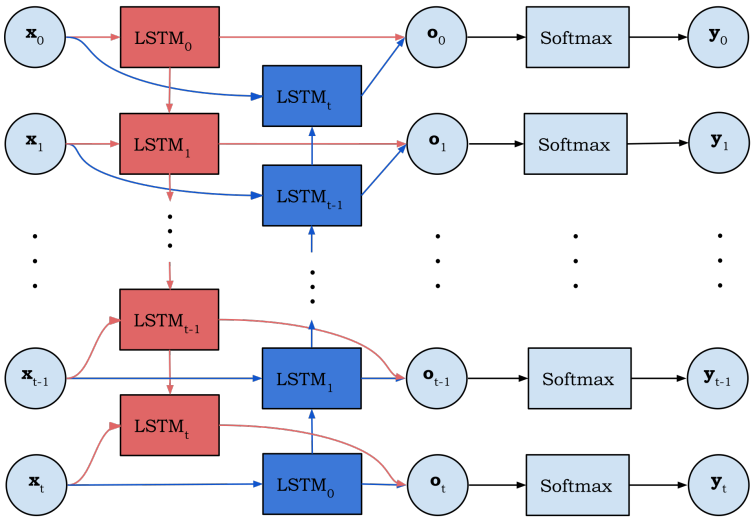


Figure 3.1: Illustration of the BRNN architecture, where the forward layer is illustrated in red, and the backward layer in dark blue. $\mathbf{x}$ is the input vector, $\mathbf{o}$ the output vector before it is passed through the dense layer, and $\mathbf{y}$ the output vector after it has been passed through the dense layer.

3.2.2 Sequence-to-sequence model

Sequence-to-sequence models have been used extensively for tasks like neural machine translation and image captioning. However, to the best of our knowledge, they have not been used for data reconstruction, which is why they were deemed interesting to explore for that purpose. The sequence-to-sequence model allows the lengths of sequences to vary, which is beneficial in this project due to the different lengths of sessions in the real event data. Figure 3.2 mirrors the high level architecture for the sequence-to-sequence model proposed in this project (compared to the general case presented in Figure 2.4).

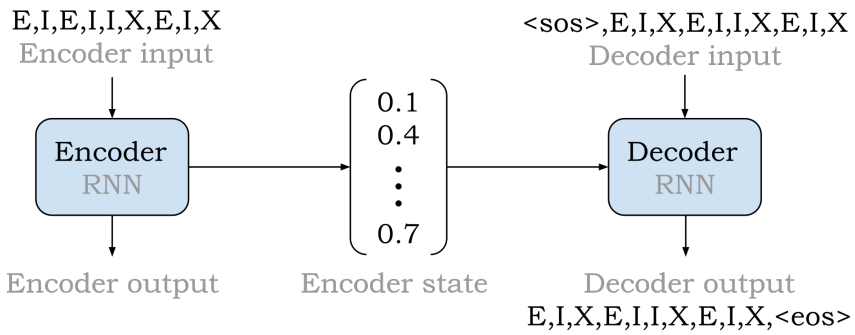


Figure 3.2: The training process of the sequence-to-sequence model used in this project. The state of the encoder is passed on to the decoder, which during training takes the target sequence as input and then produces an output. In this illustration the model is fed an input sequence where an X event is missing, succeeds in reconstructing the missing event, and produces the correct output sequence.

As mentioned in Section 2.5.1, the sequence-to-sequence model might encounter problems with increasingly long input sequences. Therefore, when training on datasets containing sequences with a maximum length equal to or greater than 100, separate models were trained with and without attention. Figure 3.3 shows the sequence-to-sequence architecture for this project with attention. The final state of the encoder is still used to set the initial state of the decoder, but the decoder can also access information from the input sequence through context vectors as described in Section 2.5.1.

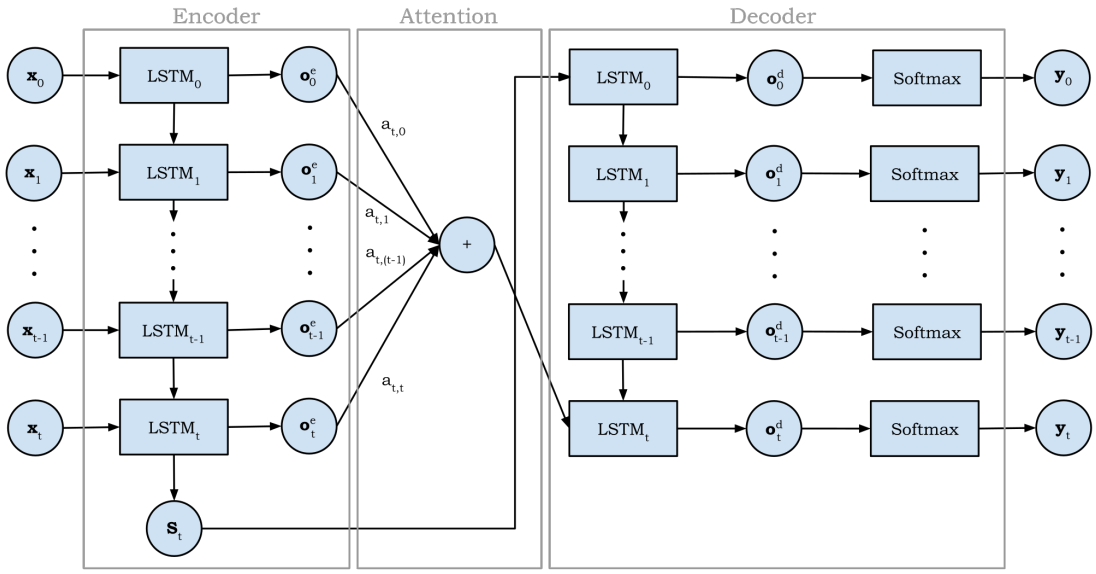


Figure 3.3: Illustration of the sequence-to-sequence architecture with attention. To keep the figure easy to overview, attention has only been illustrated in one case, namely how it works when the model is generating output in time step t. Attention can be removed, in which case this figure illustrates a standard sequence-to-sequence model. For each time step t in the input sequence, $\mathbf{x}_t$ is an input vector and $\mathbf{o}_t^e$ and $\mathbf{o}_t^d$ are output vectors from the encoder and decoder. The encoder output is not used in a standard sequence-to-sequence model, as the input sequence is to be represented as a fixed-length vector. However, the decoder output is passed through the dense layer and becomes the final output $\mathbf{y}$. $\mathbf{s}_t$ is the final state of the encoder, which is the aforementioned fixed-length vector used as the initial state of the decoder.

3.3 Loss function

The same CE-based loss function was used to train both the BRNN and the sequence-to-sequence model. Specifically, the CE loss was computed for each categorical feature, and the *mean squared error* (MSE) was computed for the scalar time feature. All of these values were then added together and scaled by the number of features to produce the final loss. To penalize predictions of the wrong length, the constant 1.0 was added to the loss for any event position where the length of the target and output sequences differed.

3.4 Training process

The models were continuously optimized on shuffled training data in small batches, and the training and validation losses were calculated at the end of each epoch. Early stopping was used on the validation loss, meaning that training was stopped once the validation loss had not improved for a number of epochs. During the training of both models, certain hyperparameters were fixed whereas others were adjusted in order to find the best set of parameters for the task at hand. Once we had settled on the best hyperparameters for each dataset, the models were evaluated on *test sets*. That is, sets containing data which the models had previously not seen during training or validation.

3.4.1 Fixed hyperparameters

Due to time constraints, certain parameters were not tweaked during training, but instead set to values commonly known as good choices. These are detailed below:

- The **optimizer** used was *Adam* [17] (derived from *adaptive moment estimation*), which is a popular choice and known to achieve good results fast.
- The **learning rate** was set to 0.001. The Adam optimization algorithm then computes and adapts individual learning rates for each weight in the network.
- The **activation function** used internally in both the LSTM and GRU cells was the tanh function. For the final dense layer of both models, the softmax function was used.
- **Dropout** was used on the input for both models (for both the encoder and decoder of the sequence-to-sequence model), with the rate set to 50%, which in a large number of cases of neural network constructions is near-optimal [19].

3.4.2 Adjusted hyperparameters

To achieve optimal performance for the two models, the following hyperparameters were adjusted between training runs:

- The **number of units** in each layer was initially low and subsequently increased until no performance improvement was achieved.

- The **number of layers** for each model was similarly set as low as possible at the start, with a single-layer BRNN and a sequence-to-sequence model with both a single-layer encoder and decoder. The models were then made deeper until no gains in performance were made.
- Various **batch sizes** for the different models were tried until the best choice for each model and dataset had been established.
- The **cell structures** LSTM and GRU were both evaluated.
- The sequence-to-sequence model was trained both with and without the **attention mechanism** to evaluate its impact on the model performance.

3.5 Model evaluation

Model performance has been evaluated in two ways. Firstly, event-level metrics based on the Levenshtein distance were created in order to evaluate how well the models succeeded in retaining input events and reconstructing missing events accurately. Secondly, sequence-level metrics that classify entire sequences were created to show model accuracy on a sequence-level. In this thesis, the event-level metrics are considered the better choice to evaluate model performance since they are independent of sequence length. A sequence-level classification can be changed by a single mistake. The probability of the presence of such mistakes is higher for longer sequences.

3.5.1 Metrics

Although the Levenshtein distance, mentioned in Section 2.2, cannot be used by TensorFlow directly as a loss function during optimization it can be used to evaluate the output of a network. Based on the Levenshtein distance, metrics were constructed to assess the performance of the models. Since the real event data used in this project contains multiple attributes, events from the input and target sequences were matched with the output sequence using the categorical features time and page. The matching was done with a dynamic programming algorithm for Levenshtein distance as a basis. The original algorithm calculates the distance as the sum of all insertions, substitutions and deletions needed to transform one sequence into another. In this project, the actions needed to transform the output sequence into the target sequence are tracked. Substitutions are regarded as a combination

of an insertion and a deletion. Events from the two sequences which the algorithm regard as equal are paired together, and events present in the target sequence but missing from the output sequence and vice versa are kept track of.

The metrics are called *deletions*, *insertions* and *reconstructions*:

- **Deletions** (D) measure the percentage of events present in the input sequences but missing from the output sequences. Since all of the events in the input sequences are expected to be part of the output sequences, deletions should be as low as possible for good network performance.

$$D = \frac{|\text{Input events} \setminus \text{Output events}|}{|\text{Input events}|} \tag{3.6}$$

- **Insertions** (I) measure the percentage of events that are present in the output sequences but not in the target sequences. These are events that have been mistakenly added, and therefore insertions should also be as low as possible.

$$I = \frac{|\text{Output events} \setminus \text{Target events}|}{|\text{Target events}|} \tag{3.7}$$

- **Reconstructions** (R) measure the percentage of removed events that the model succeeds in reconstructing, meaning that a higher percentage is better.

$$R = \frac{|\text{Output events} \cap (\text{Target events} \setminus \text{Input events})|}{|\text{Target events} \setminus \text{Input events}|} \tag{3.8}$$

To illustrate these metrics with an example, the three matrices in (3.9) represent the input fed to the model, the output produced by the model, and the target sequence (the expected output). Each row in the matrices represents an event with structure $(Type, Page, Time)$.

$$\begin{array}{c} \textbf{Input} \\ \begin{bmatrix} E & 1 & 0.00 \\ I & 1 & 0.33 \\ E & 2 & 1.00 \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Output} \\ \begin{bmatrix} E & 1 & 0.02 \\ I & 1 & 0.28 \\ I & 2 & 0.73 \\ X & 1 & 0.97 \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Target} \\ \begin{bmatrix} E & 1 & 0.00 \\ I & 1 & 0.33 \\ X & 1 & 0.67 \\ E & 2 & 1.00 \end{bmatrix} \end{array} \tag{3.9}$$

In (3.9), the event (E, 2, 1.00) is present in the input sequence but not in the output sequence, resulting in deletions of 0.25. The insertions are 0.25 as well, since the

event (I, 2, 0.73) is present in the output sequence but not in the target sequence. Reconstructions are equal to 1 since only the target event (X, 1, 0.67) is missing in the input sequence and that event is successfully recreated in the output sequence as (X, 1, 0.97). This event does have a timestamp that deviates quite a bit from the target event but is considered correct by having the correct type and page.

Except for the deletions, insertions and reconstructions, an evaluation of how well the models perform in reconstructing features other than type and page is done as well. The other features considered are the timestamp, interaction type and intent. Since the interaction type and intent are not present in the PageEnterEvents or PageExitEvents, these will be set to *missing* in the input data. The metrics are called *time MSE*, *accuracy interaction type* and *accuracy intent*. The MSE is used to calculate the accuracy for the timestamp. The accuracy for the categorical features is calculated by taking the total number of correct interaction types/intents in the output sequences divided by the total number of events.

During the training of the models, early stopping is applied to the value of the CE-based loss function. The hypothesis was that although the optimization is conducted on the loss function, the metrics improve as the CE-based loss decreases. To test this hypothesis, the CE-based loss and a weighted sum of the metrics were investigated and compared. The *weighted metrics loss* is defined by first multiplying each metric with a weight and then adding them all together. These weights could be chosen according to the characteristics desired in a predicted sequence and could consequently vary from case to case. In this project, the weights were set based on the logic that a deletion is worse than both a failed reconstruction and an insertion, since a deletion should never occur. A failed reconstruction is worse than an extra insertion, since we would rather see the models reconstructing missing events and accidentally adding extra events than failing to reconstruct missing events. In turn, an insertion is worse than getting the wrong interaction type, intent or time. The weighted metrics loss function adapted in this project is given in (3.10). In this equation, the weighted metrics loss is notated as loss_{wm}, D is the deletions, R the reconstructions, I the insertions, a_{it} the accuracy for interaction type, a_i the accuracy for intent, and MSE_t the time MSE.

$$\text{loss}_{wm} = 5D + 2R + I + 0.5((1 - a_{it}) + (1 - a_i) + \text{MSE}_t) \tag{3.10}$$

3.5.2 Accuracy matrices

In addition to the metrics, a modified version of *confusion matrices* was created to present the accuracy of the models on a sequence level. In classification problems, confusion matrices can be used to illustrate the performance of a classifier for each individual class. The rows and columns in a confusion matrix show the *actual* and *predicted* classes respectively. That is, the number at row m and column n shows how many elements belonging to class m that were classified as belonging to class n. Applying similar logic to the task of reconstructing incomplete sequences, we devise the classes *complete* and *incomplete* since a sequence must belong to either class. However, a regular confusion matrix does not fully capture the complexity that this project proposes since the output of each model is a sequence and not just an indicator to which class the input belongs. In order to present our results in a format similar to a confusion matrix we therefore have to define when we considered the model to have made a correct assumption regarding the completeness of the sequence when producing an output sequence. The number of classes are still two – complete and incomplete – but we have added another dimension to the predicted output called *failed*. Our full definitions of the different classes are listed below:

- An output sequence is considered **complete** if all events present in the input sequence are passed through to the output sequence. No additional events are added to the output sequence compared to the input sequence.
- An output sequence is considered **incomplete** if all events present in the input sequence also exist in the output sequence, and at least one additional event has been added to the output sequence.
- An output sequence is considered to be **failed** if not all of the events in the input sequence are present in the output sequence.

To clarify, Table 3.4 holds a complementary definition of the classes.

Table 3.4: Definitions of the different classes.

Class	**Condition**
Complete	Input events = Output events
Incomplete	Input events $\subsetneq$ Output events
Failed	Input events $\nsubseteq$Output events

These complete, incomplete and failed classes are in this thesis used to build what we will refer to as *accuracy matrices*. An example of an accuracy matrix is illustrated in Table 3.5. In this example, the model has successfully classified a% of all complete

sequences and e% of all incomplete sequences. These are the correct classifications, which is indicated by the bold font. In b% of all complete sequences, the model has added events and these sequences are therefore classified as incomplete. In d% of all incomplete sequences, the model has instead generated an output identical to the input sequence and accordingly these sequences are classified as complete. In c% of all complete sequences and f% of all incomplete sequences the model has removed events in the output sequence compared to the input sequence, meaning that these sequences are classified as failed.

Table 3.5: Example of an accuracy matrix used in this project to evaluate the number of successful/unsuccessful sequence predictions.

Predicted / **Actual**	**Complete**	**Incomplete**	**Failed**
Complete	**a**	b	c
Incomplete	d	**e**	f

The accuracy is calculated as the number of correctly classified sequences divided by the total number of sequences. In the example illustrated in Table 3.5 the accuracy is $\frac{a+e}{a+b+c+d+e+f}$.

4

Results and Evaluation

This chapter summarizes the results and performance of the models when run on the test sets. The values of the hyperparameters for both the BRNN and the sequence-to-sequence model are described, and model performance is evaluated and reflected on. If nothing else is specified, the synthetic dataset referred to in this chapter contains sequences of 20-100 events per sequence.

4.1 Setups

Each experimental setup was run four times, and the results presented in this chapter show the average results for the best performing BRNNs and sequence-to-sequence models. The datasets used are presented in Sections 3.1.1 and 3.1.2, and the percentage split between the training, validation and test set was 70/20/10. The models were run on Google ML Engine instances with a single NVIDIA Tesla K80 GPU and TensorFlow version 1.7.

The fixed hyperparameter values were the same for both models and are detailed in Section 3.4.1. The values of the adjusted hyperparameters for the best performing BRNNs are illustrated in Table 4.1, and for the best sequence-to-sequence models in Table 4.2. As can be seen, both models used in this project contain fairly few units. The maximum number of units is 500 and is used in the BRNN that handles synthetic data, all other models only have 250 units each. In comparison, Berglund et al. [11] used a BRNN with 684 units and 2 layers to achieve their results. Although we tried structures with more units and fewer layers, we did not manage to improve the performance of the BRNN. Therefore, the networks contain fewer units but more layers than the BRNN used by Berglund et al. [11].

Table 4.1: Best values of the adjusted hyperparameters for the BRNNs.

Dataset	Units	Layers	Batch size	Cell structure
Synthetic	500	5	30	LSTM
Real	250	5	30	LSTM

Table 4.2: Best values of the adjusted hyperparameters for the sequence-to-sequence models.

		Layers			
Dataset	Units	Encoder	Decoder	Batch size	Cell structure
Synthetic	250	3	3	30	LSTM
Real	250	3	3	100	LSTM

The values of the hyperparameters were adjusted as follows:

- The **number of units** was initially set to 50 when applied to the synthetic single-feature data. When adding complexity to the data, the number of units was increased until the model performance plateaued. The highest number of units tried out in this project was 1000. It turned out that the BRNN performance on synthetic data peaked when the networks contained 500 units each, and 250 units each for the real event data. The sequence-to-sequence model contains 250 units for both datasets.

- The **number of layers** was adjusted from a single-layer BRNN and both a single-layer encoder and decoder for the sequence-to-sequence model. When increasing the number of layers it was noted that models with twice as many layers were better than models with twice as many units. However, increasing the depth beyond 5 layers for the BRNN and 3 layers for both the encoder and decoder in the sequence-to-sequence model did not improve the results.

- The **batch size** was varied from 1 to 500. For the BRNN it was noted that batch sizes less or greater than 30 worsened the performance. The sequence-to-sequence model had the same optimal batch size of 30 for the synthetic data, but a batch size of 100 for the real event data. Batch sizes over 100 worsened the performance of the model.

- The **cell structures** used in this project were LSTM and GRU cells. Each parameter setting was run multiple times with both LSTM and GRU cells. LSTM cells proved to generate better results than GRUs, so LSTM cells are exclusively used in the models.

4.2 Model performance

This section presents the model performance on both datasets in several ways. Results of the Levenshtein distance-based metrics are presented, followed by the training and validation losses. The correlation between the aforementioned metrics and the loss function is shown along with the final test losses. In addition, the effects of using attention on the sequence-to-sequence model is presented. Finally, accuracy matrices are presented as a way to measure the accuracies of both models on a sequence level.

4.2.1 Metrics

As illustrated in Table 4.3, the average number of reconstructions in the real event test set is 64.99% for the BRNN and 75.42% for the sequence-to-sequence model. These are the means of four runs with the same hyperparameters for each model. To give an idea of the distribution of reconstruction rates in these runs, 95% confidence intervals were calculated for both models. A 95% confidence interval shows that in 95% of all runs, the mean of the reconstruction rates is found between a lower and an upper limit. For the BRNN, the 95% confidence interval is $[64.16\%, 65.83\%]$, while the reconstruction rates for the sequence-to-sequence model have a 95% confidence interval of $[74.81\%, 76.02\%]$. These confidence intervals show that the reconstruction rates are not likely to overlap for the BRNN and the sequence-to-sequence model. Accordingly, the sequence-to-sequence model can be considered a better performing model for event reconstruction in this project.

Table 4.3: Average deletions, insertions and reconstructions of four independent runs with the best parameter settings on the test sets. The synthetic dataset contains 10,000 sequences of size 20-100 events, and sequences in the real event dataset are 100,000 with a length of 6-136 events.

Model	Dataset	Deletions	Insertions	Reconstructions
BRNN	Synthetic	2.86%	3.14%	82.82%
BRNN	Real	8.18%	9.23%	64.99%
Sequence-to-sequence	Synthetic	0.20%	0.14%	98.71%
Sequence-to-sequence	Real	1.50%	2.04%	75.42%

To be able to evaluate how well the models succeeded in getting the other attributes right, Table 4.4 illustrates the time MSE, the accuracy for interaction type and the accuracy for intent for the real event data.

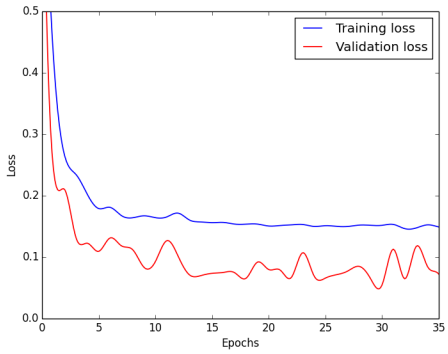
Table 4.4: Average time MSE, accuracy for interaction type and accuracy for intent of four independent runs with the best parameter settings on the real event test set.

Model	Time MSE	Accuracies	
		Interaction type	Intent
BRNN	0.0611	72.74%	65.83%
Sequence-to-sequence	0.0659	94.69%	88.78%

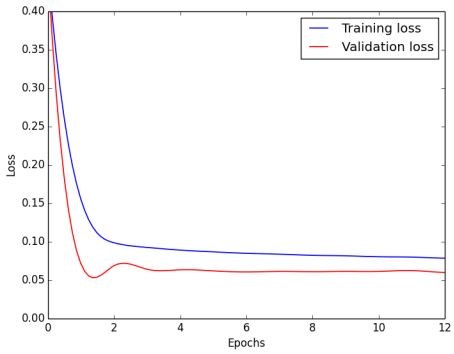
The fact that the models perform better on the synthetic data compared to real event data is expected since the synthetic data contains events evenly distributed over time, I events present for every page visited, and only a small number of pages to choose from. Although the models have a harder job with the increased complexity in the real event data, the real event data has an advantage that the synthetic dataset does not have. The page IDs in the synthetic data were chosen uniformly at random for each set of subsequent events relating to the same page. In the real event data however, the event sequences conform to the navigational structure of the application, meaning that some pages are impossible to access from others. Furthermore, the real event pages have unique characteristics which make the level of interaction or time spent on any given page vary greatly. These patterns are something that the models could learn and use to predict where events are missing from input sequences.

4.2.2 Training and validation losses

The training and validation losses for the best performing BRNNs are plotted in Figure 4.1, and for the best performing sequence-to-sequence models in Figure 4.2.

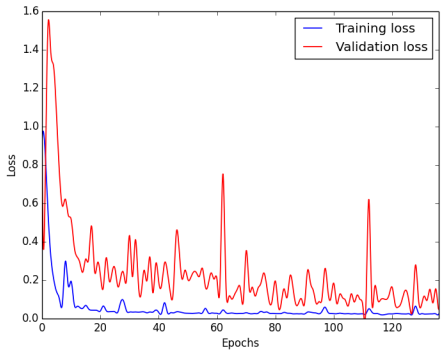


(a) Synthetic data.

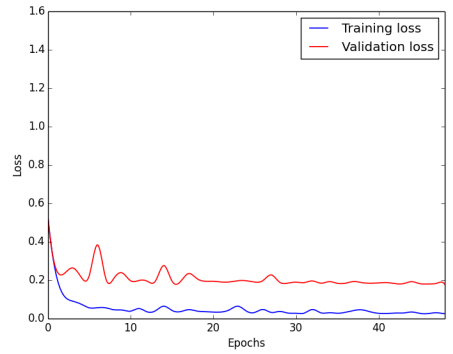


(b) Real event data.

Figure 4.1: Illustration of both the training and validation loss for the best performing BRNNs.



(a) Synthetic data.



(b) Real event data.

Figure 4.2: Illustration of both the training and validation loss for the best performing sequence-to-sequence models.

Figure 4.2 shows the decrease of the training and validation losses for the sequence-to-sequence models, which looks as expected. However, as seen in Figure 4.1 the validation losses for the BRNNs are lower than the training losses, which is unusual. We have evaluated the construction of our models and cannot find any errors, so we do not know why this occurs.

4.2.3 CE-based loss and weighted metrics loss correlation

The hypothesis, mentioned in Section 3.5.1, was that the weighted metrics loss decreases in line with the CE-based loss. This hypothesis was tested, and Table 4.5 contains the average final CE-based and weighted metrics losses for the BRNNs and the sequence-to-sequence models after four independent runs on the test set. In Figures 4.3 and 4.4, the weighted metrics loss is plotted together with the validation loss for the four different runs.

Table 4.5: Mean of the CE-based loss and the weighted metrics loss for four independent runs with the best parameter settings on the test sets. The intervals of epochs in this table illustrate the ranges in which the four runs of each model achieved the lowest CE-based loss.

Model	Dataset	Losses		Epochs
		CE-based	Weighted metrics	
BRNN	Synthetic	0.0545	0.5189	[17, 41]
BRNN	Real	0.0590	1.3854	[11, 14]
Sequence-to-sequence	Synthetic	0.0729	0.0377	[73, 134]
Sequence-to-sequence	Real	0.1756	0.6616	[26, 60]

As illustrated in Table 4.5, the CE-based loss for the real event data for the BRNN is much lower than the CE-based loss for the sequence-to-sequence model although the weighted metrics loss is higher for the BRNN than for the sequence-to-sequence model. The CE-based loss for the BRNN is not as well correlated with the weighted metrics loss as it is for the sequence-to-sequence model, which is illustrated in Figures 4.3 and 4.4.

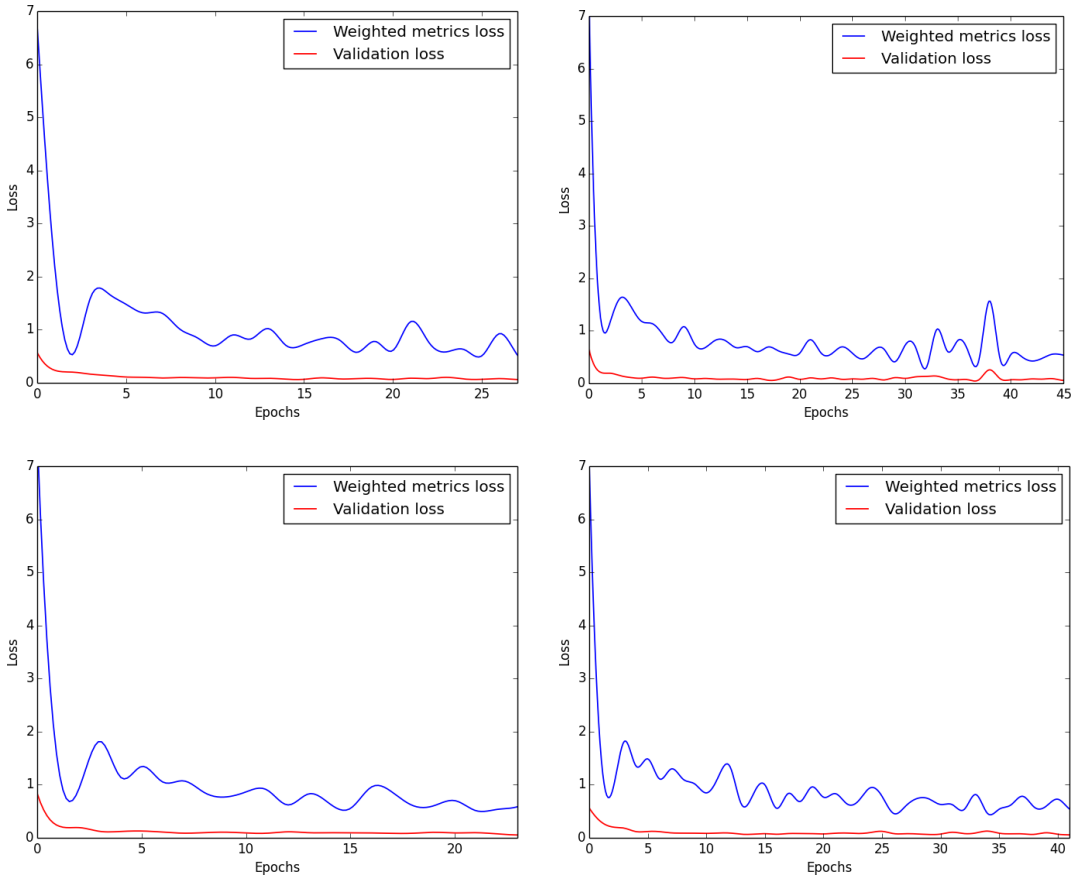


Figure 4.3: Illustration of the relationship between the validation and weighted metrics loss in four runs of the best BRNN on the synthetic validation set.

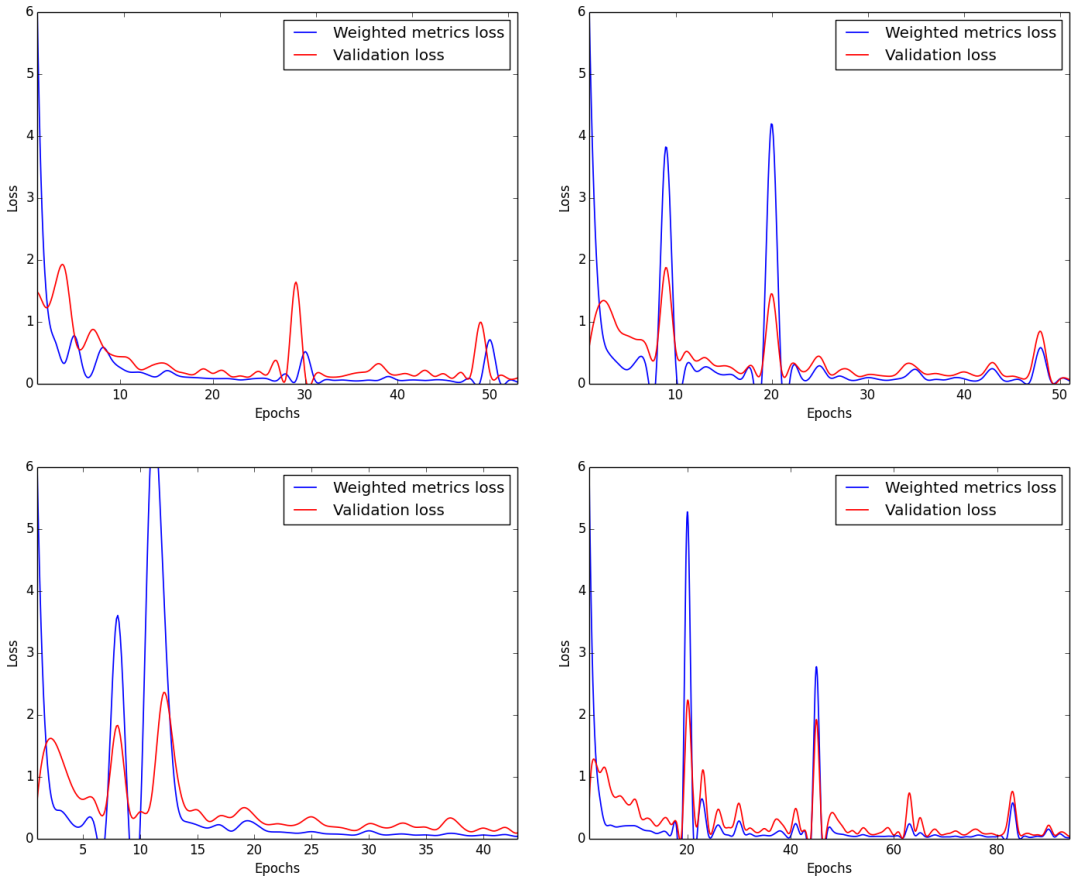


Figure 4.4: Illustration of the relationship between the validation and weighted metrics loss in four runs of the best sequence-to-sequence model on the synthetic validation set.

In order to statistically validate the relationships between the CE-based loss and the weighted metrics loss, Spearman's rank correlation coefficient was calculated on the four runs for both models. This coefficient is defined in (4.1), where $\bar{x}$ and $\bar{y}$ are sample means of variables x and y. It measures correlation between two variables and takes on values between ± 1, where a value of ± 1 is a perfect correlation. The alpha value was set to the commonly used 0.05, meaning that a p-value smaller than 0.05 indicates that Spearman's rank correlation coefficient is statistically significant and that the probability is less than 5% that the correlation between the variables is coincidental. In this case, Spearman's rank correlation coefficient for the four sequence-to-sequence model runs in Figure 4.4 ranges between $[0.82, 0.93]$, with p-values in range $[1.37 \times 10^{-56}, 9.97 \times 10^{-22}]$. Since all results are statistically significant, the probability is less than 5% that the correlation between the losses occur by chance. The coefficients are all very strong and accordingly,

optimizing the loss function can also be considered improving the weighted metrics loss.

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{4.1}$$

For the BRNN however, Spearman's rank correlation coefficient ranges between $[0.28, 0.49]$ with p-values in range $[8.13 \times 10^{-6}, 0.021]$. With an alpha value of 0.05, all results are significant, but compared to the very strong correlation for the sequence-to-sequence model, the correlation between the loss function and the weighted metrics loss is merely weak to moderate for the BRNN.

A hypothesis to why the correlation is weaker for the BRNN than for the sequence-to-sequence model, is that the model learns some patterns internally that minimizes the loss function, but that does not minimize the weighted metrics loss. The higher weighted metrics loss value could accordingly be explained by the composition of the weighted metrics loss. Since each metric is assigned a weight based on the logic that some types of errors are worse than others (explained in Section 3.5.1), the weighted metrics loss can differ between models although the CE-based loss is similar. In this project, a high rate of deletions is considered worse than any other errors. The BRNN could hence be doing more deletions than the sequence-to-sequence model, and would in that case be assigned a higher weighted metrics loss. This is confirmed by Table 4.3, which illustrates the deletions, insertions and reconstructions when both models are run on the test set.

4.2.4 Effects of using attention

Attention was one of the adjusted hyperparameters, and improved model performance dramatically. The sequence-to-sequence model was run both with and without attention, for synthetic sequences of 6-30 and 20-100 events. The results are illustrated in Table 4.6.

Table 4.6: Summary of the effects that attention has on the sequence-to-sequence model's ability to reconstruct events in sequences of different lengths.

Events per sequence	Attention	Deletions	Insertions	Reconstructions
6-30	No	4.36%	5.14%	94.96%
6-30	Yes	0.76%	0.28%	97.29%
20-100	No	81.29%	132.55%	17.36%
20-100	Yes	0.44%	1.99%	98.46%

For short sequences, attention improved the number of deletions, insertions and reconstructions by a few percentage points. However, as sequences grew longer and more rich in content than the encoder could capture in its internal state, attention had an enormous effect on the model's performance. Accordingly, attention was always activated when the models ran on real event data.

A hypothesis as to why attention is a good choice for the sequence-to-sequence model in this project, is that the access to context vectors can help the model recover if it generates the wrong event. Since the sequence-to-sequence model takes its previously generated output as input during the validation process, it seems possible that if the model generates the wrong event, it is more likely to make additional mistakes in the same sequence since the wrong event would be fed to the model as input. However, by having access to context vectors, the model can more easily recover from a mistake since it does not only have to rely on the fixed-size feature vector and its own generated output, but can incorporate information from the context vectors as well.

4.2.5 Accuracy matrices

The accuracy matrices for the BRNNs are presented in Tables 4.7 and 4.8. Table 4.7 holds the results from the synthetic data and Table 4.8 from the real event data. In the same way, accuracy matrices for the sequence-to-sequence models are presented in Tables 4.9 and 4.10.

Table 4.7: Accuracy matrix, illustrating the number of successful/unsuccessful sequence predictions on the *synthetic* test dataset for the *BRNN*. The accuracy is 68.8%.

Actual \ Predicted	**Complete**	**Incomplete**	**Failed**
Complete	**99.42%**	0.10%	0.48%
Incomplete	0.05%	**38.18%**	61.77%

Table 4.8: Accuracy matrix, illustrating the number of successful/unsuccessful sequence predictions on *real* event test data from event logs for the *BRNN*. The accuracy is 57.2%.

Actual \ Predicted	**Complete**	**Incomplete**	**Failed**
Complete	**86.47%**	3.65%	9.88%
Incomplete	4.82%	**27.94%**	67.24%

Table 4.9: Accuracy matrix, illustrating the number of successful/unsuccessful sequence predictions on *synthetic* test data for the *sequence-to-sequence model*. The accuracy is 91.6%.

Actual \ Predicted	**Complete**	**Incomplete**	**Failed**
Complete	**98.02%**	0.24%	1.74%
Incomplete	0.00%	**85.27%**	14.73%

Table 4.10: Accuracy matrix, illustrating the number of successful/unsuccessful sequence predictions on *real* event test data from event logs for the *sequence-to-sequence model*. The accuracy is 69.4%.

Actual \ Predicted	**Complete**	**Incomplete**	**Failed**
Complete	**87.57%**	5.69%	6.74%
Incomplete	6.06%	**51.30%**	42.64%

The sequence-to-sequence model performs very well on the synthetic data and correctly outputs 98.02% of all complete sequences, as illustrated in Table 4.9. For 85.27% of all incomplete sequences, the model succeeds in retaining all events from the input sequence and to add at least one additional event in its output. On the real event data, with additional pages and the features interaction type and intent,

the sequence-to-sequence model succeeds in correctly outputting 87.57% of all complete sequences, and to retain all input events and add new events into 51.30% of all incomplete sequences (see Table 4.10). We consider the results for both datasets to be good for a model that has – to the best of our knowledge – never been used for event reconstruction before.

As seen in Tables 4.7 and 4.9, the BRNN outperforms the sequence-to-sequence model when it comes to correctly classifying complete sequences from the synthetic dataset, with 99.42% compared to 98.02%. For the corresponding classification in the real event dataset, the sequence-to-sequence model actually performs slightly better with 87.57% compared to the BRNN's 86.47% (see Tables 4.10 and 4.8). The huge difference between the models is the number of correctly classified incomplete sequences. The BRNN is then remarkably worse than the sequence-to-sequence model with a correct prediction of only 38.18% compared to the sequence-to-sequence model's 85.27% (see Tables 4.7 and 4.9). Accordingly, the latter is more than twice as good at classifying incorrect sequences compared to the BRNN. For the real event dataset the difference in performance is similar. As illustrated in Table 4.8, the BRNN has a correct classification rate of 27.94% for all incomplete sequences, and the corresponding percentage for the sequence-to-sequence model is – as displayed in Table 4.10 – 51.30%.

4.3 Discussion

When comparing the accuracy matrices to the number of reconstructions, illustrated in Table 4.3, one thing to note is that although the sequence-to-sequence model succeeds in reconstructing 75.42% of all missing events in the real event dataset, it also incorrectly classifies 48.70% of all incomplete sequences as either complete or failed. However, what to keep in mind when viewing these accuracy matrices is that the problem we are aiming to solve in this project is not a classification problem. Therefore, the conditions for different classes are not obvious and have instead been defined in Section 3.5.2. A sequence is classified as failed if any event in the output sequence has been deleted compared to the input sequence. This condition is very strict considering that real sequences are up to 136 events in length, which makes one deleted or inserted event in a complete sequence quite plausible.

Another important thing to be aware of when studying the results presented in this chapter is that some events in the real event data are impossible to reconstruct. To be able to recreate missing events in sequences, an important prerequisite is that the information given in the available events together with the information learned from

the training data is sufficient to infer what has been left out. As such, the process generating the events must possess some degree of determinism which makes some sequences more likely than others and implies some interdependence between events. To pose an example, a partially incomplete sequence of events from a completely stochastic process such as a coin tossing process would be impossible to reconstruct with any accuracy, especially if the number of missing events is unknown. Regarding the events used in this project, there are other patterns which make reconstruction challenging when only given access to the three available event types. For instance, if a user stays on a page and repeatedly produces repetitions of identical InteractionEvents to alter some application state which is unrelated to the navigation, such InteractionEvents would be difficult to reconstruct without the model having access to additional information. For instance, if a user were to repeatedly increment the value of a numeric counter in the application before switching pages, thereby producing InteractionEvents only stating an increment but not the resulting value, lost interactions could never be reconstructed without knowing the final value of the counter. Accordingly, the inability to reconstruct certain events is a contributing factor to the lower rate of reconstructions for the real event data compared to the synthetic data.

What could be interesting is to compare the results presented in this thesis to a baseline model. For instance, for the synthetic data, a rule-based model much simpler than an RNN would probably be able to reconstruct all missing events. We state that this is the case since the timestamps are evenly distributed in the synthetic data, and the logic is very simple. It would therefore most likely be possible for a simpler model to match E and X events with each other.

As for the performance of a baseline model on the real event data, we state that it will not be as high as the performance of our models. In the real event data, the basic logic for PageEnterEvents and PageExitEvents is the same as for the synthetic data, namely that a PageExitEvent has to be followed by a PageEnterEvent, and a PageEnterEvent and a PageExitEvent cannot occur without an InteractionEvent in between them. However, it is likely that the navigational structure of the application producing the events is too complex for a simple model to capture. We reason that a baseline model would therefore not be able to reconstruct the InteractionEvents, but only PageEnterEvents and PageExitEvents.

Importantly, if both a PageEnterEvent, intermediate InteractionEvents and a PageExitEvent were removed for the same page, the resulting sequence would seem valid to a baseline model which has no knowledge of the navigational structure of the application. The baseline model could therefore never reconstruct such lost event sequences. However, we can calculate the baseline performance in the *best* case,

when every page visited has either a PageEnterEvent or PageExitEvent present in the input data for each page visited.

The number of missing events of each type in the real event dataset is illustrated in Table 4.11. Accordingly, a baseline model would in the best case be able to reconstruct 64.85% of all missing events (all PageEnterEvents and PageExitEvents). As shown in Table 4.3, the BRNN succeeded in reconstructing 64.99% of all missing events, which is slightly better than the baseline model's best case performance. The sequence-to-sequence model on the other hand succeeded in reconstructing 75.42%, outperforming the baseline model's best case performance by over 10 percentage points.

Table 4.11: Illustration of how many of the missing events in the real event data test set that belong to each event type. The rate is calculated by dividing the number of removed events of the specified event type by the total number of missing events in the dataset.

Event type	**Number of removed events**	**Rate of removed events**
PageEnterEvent	4934	32.20%
InteractionEvent	5387	35.15%
PageExitEvent	5004	32.65%

Lastly, a possible flaw of the Levenshtein distance-based algorithm described in Section 3.5.1 is worth mentioning. As previously mentioned, it matches events in the target sequence with the events deemed as equal in the output sequence to deduce the actions needed to transform an output sequence into a target sequence. The possible flaw is that it only judges equality by the two attributes type and page, which opens up for the possibility that the models insert another event with the same type and page *before* a more suitable event, and that the algorithm matches with that event instead of the latter one. Although this event will have the same type and page as the original event, the other attributes might be different. The number of insertions, deletions and reconstructions will remain the same with the mismatched event, but values of other metrics could change. The time MSE along with accuracies for interaction type and intent – illustrated in Table 4.4 and presented in Section 3.5.1 – might therefore worsen.

4.4 Challenges

During the course of this project, we faced a number of challenges. Transitioning from synthetic data to real event data greatly increased the complexity of the learn-

ing task. Just as in the synthetic dataset, the real event dataset contains five event types and one scalar feature, namely the time. However, the maximum sequence length is 136 instead of 100, and the total number of distinct pages is much larger than in the synthetic dataset. Additionally, the real event data is dynamic in both the temporal and behavioral sense. That is, the timing and content of the real events vary according to real user actions in a real software application, which may also change in functionality from version to version. In contrast, the synthetic data is more regular with evenly spaced timestamps and pages chosen uniformly at random.

When adjusting the models to be able to handle real event data we encountered some problems. During the initial implementation stages, we used Keras, a high-level neural networks API, with TensorFlow as the underlying library. Initially, synthetic sequences were stored in and parsed from text files. However, due to the large amounts of input data in the real event sequences we had to switch to a more efficient way of reading and storing data, the TensorFlow-native TFRecord format. Commonly, the reading of contents from TFRecords are included in the TensorFlow computation graph in the form of data tensors. Unfortunately, feeding data tensors as input to the Keras training loop is currently not possible, which is why the models had to be reimplemented in TensorFlow instead.

5

Conclusion

The goal of this project was to reconstruct missing data in event logs by using RNNs. Two models were used to solve the problem, namely a BRNN and a sequence-to-sequence model. The models were given either complete or incomplete sequences as input, and were trained to output complete sequences. One thing that complicated the task was that the locations of the missing events in the input sequences were not revealed to the models.

After conducting this project we can state that for our real event dataset it is possible to reconstruct up to 75.42% of all missing events using a sequence-to-sequence model. It is also possible to use a BRNN to solve this problem, but it only manages in reconstructing 64.99% of all missing events. However, both models perform better than the baseline model we have compared them to in this thesis, which means that both models can successfully be used for the task. To the best of our knowledge, sequence-to-sequence models have not previously been used for data reconstruction. Also, as far as we know, RNNs have never been used for data reconstruction without stating the locations of missing data in the input. Accordingly, we consider this project a success.

Future work could include training the models on larger datasets than 100,000 sequences to increase the variation in the dataset. Additionally, adding new event types could be interesting, to see how the models would handle increases in complexity and contextual information. Lastly, our algorithm based on the Levenshtein distance deem events as being equal only by checking the attributes type and page. It would be interesting to incorporate a more sophisticated distance measure which takes the other attributes into account when pairing events.

References

[1] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 114–117. IEEE, 2010.

[2] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, 2013.

[3] James F Kurose and Keith W Ross. Computer networking: a top-down approach. *Addison Wesley Computing*, 2013.

[4] Ramon Caceres and Liviu Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE journal on selected areas in communications*, 13(5):850–857, 1995.

[5] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[6] Denny Britz, Anna Goldie, Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.

[7] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28 (9):921–932, 1985.

[8] Crina Grosan and Ajith Abraham. Rule-based expert systems. In *Intelligent Systems*, pages 149–185. Springer, 2011.

[9] Ming Ji, Fei Wang, Jia-ning Wan, and Yuan Liu. Literature Review on Hidden Markov Model-based Sequential Data Clustering. *Applied Mechanics & Materials*, 2014.

[10] Lawrence R Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[11] Mathias Berglund, Tapani Raiko, Mikko Honkala, Leo Kärkkäinen, Akos Vetek, and Juha T Karhunen. Bidirectional recurrent neural networks as generative models. In *Advances in Neural Information Processing Systems*, pages 856–864, 2015.

[12] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[13] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[14] Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[15] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[16] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[20] Alex Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012.

[21] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[24] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[25] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[26] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[27] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[28] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.