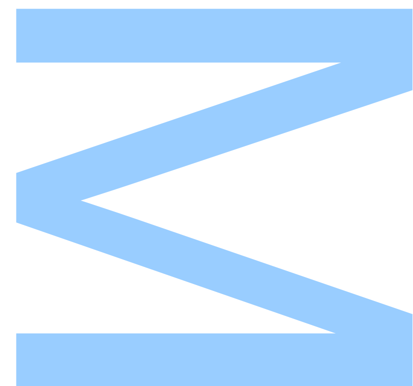
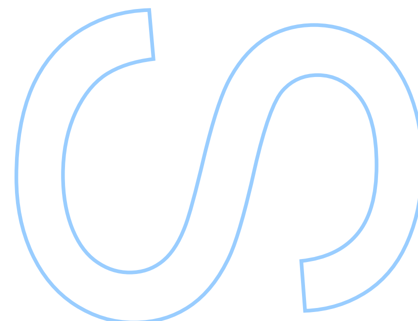
Finding patterns that predict hyper and hypoglycaemia

Ricardo Bruno Faria
MSc. Data Science
Department of Computer Science
2022

Advisor
Pedro Brandão, Faculty of Sciences of the University of Porto

Co-advisor
Vitor Santos Costa, Faculty of Sciences of the University of Porto

Supervisor
Diogo Machado, Faculty of Sciences of the University of Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/________

MSc

Declaração de Honra

Eu, Ricardo Bruno Faria, inscrito no Mestrado em Ciência de Dados da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega. Ao entregar esta dissertação, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição. Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor. Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Ricardo Faria

Braga, 29 / 09 / 2022

UNIVERSITY OF PORTO

MASTERS THESIS

Finding patterns that predict hyper and hypoglycaemia

Author:
Ricardo Bruno FARIA

Advisor:
Pedro BRANDÃO

Co-advisor:
Vitor Santos COSTA

Supervisor:
Diogo MACHADO

A thesis submitted in fulfilment of the requirements for the degree of MSc. Data Science

at the

Faculty of Sciences of the University of Porto
Department of Computer Science

December 12, 2022

UNIVERSITY OF PORTO

Abstract

Faculty of Sciences of the University of Porto

Department of Computer Science

MSc. Data Science

Finding patterns that predict hyper and hypoglycaemia

by Ricardo Bruno FARIA

Diabetes is a serious illness due to either a malfunction of the pancreas that stops producing insulin or the body's inability to use the insulin produced and if left untreated can cause serious problems. Personalized insulin injections are needed to control the glucose level of patients with type 1 diabetes and in order to try to predict the glucose level over time a machine learning analysis with the OhioT1DM Dataset will be made.

This dataset contains continuous glucose insulin, physiological sensor and self-reported life-event data collected over an eight-week period for 12 patients. The Deep Learning models used in the analysis are the $LSTM$, GRU, $BiLSTM$, $BiGRU$, $1D\ CNN$, TCN and mixed models combining $1D\ CNN + LSTM$ or GRU.

This dataset contains missing values and inconsistencies in the number of rows and columns for each variable. In order to merge every variable into a single dataframe for each patient, these variables are re-sampled to a time window of 30 minutes containing the mean, mode or sum of the values within that time window. To predict the value of the glucose level of the next 30 minutes information of all variables within the past 24 hours is used.

Different versions of the models are tested, a personalized version that only trains the models with one patient's data and a generalized version that trains with all data. The generalized version had better results and lower training time.

After this test, two more versions are tested, one that trains models with all variables and one uses only a selection of variables. The version with the selected variables had a smaller training time and slightly better results on the top performing models.

The best final models were the simpler versions of $LSTM$, GRU and $1D\ CNN + LSTM$. The combined model of $1D\ CNN$ and $LSTM$ had the lowest training time of 97 seconds

and the best performance with a $RMSE$ of 0.0617. This show that for this dataset, maybe due to the dataset size or number of variables, simple deep learning models outperform more complex models.

UNIVERSITY OF PORTO

Resumo

Faculty of Sciences of the University of Porto
Department of Computer Science

Mestrado Ciência de Dados

Encontrar padrões que prevêm hyper e hipoglicemia

por Ricardo Bruno FARIA

A diabetes é uma doença grave devido ou a um mau funcionamento do pâncreas que deixa de produzir insulina ou à incapacidade do corpo de utilizar a insulina produzida e, se não for tratada, pode causar problemas graves. São necessárias injecções personalizadas de insulina para controlar o nível de glucose dos pacientes com diabetes tipo 1 e para tentar prever o nível de glucose ao longo do tempo será feita uma análise de *Machine Learning* com o OhioT1DM Dataset.

Este dataset contém valores continuos da insulina, glicose, sensores fisiológicos e dados de eventos de vida auto-reportados, recolhidos durante um período de oito semanas para 12 pacientes. Os modelos de *Deep Learning* utilizados na análise são os $LSTM$, GRU, $BiLSTM$, $BiGRU$, $1D\ CNN$, TCN e modelos mistos que combinam $1D\ CNN + LSTM$ ou GRU.

Este conjunto de dados contém alguns *missing values* e inconsistências no número de linhas e colunas para cada variável. A fim de juntar cada variável num único dataframe para cada paciente, estas variáveis são *re-sampled* para uma janela de tempo de 30 minutos contendo a média, moda ou soma dos valores dentro dessa janela de tempo. Para prever o valor do nível de glicose dos próximos 30 minutos, é utilizada a informação de todas as variáveis nas últimas 24 horas.

São testadas diferentes versões dos modelos, uma versão personalizada que apenas treina os modelos com dados de um paciente e uma versão generalizada que treina com todos os dados. A versão generalizada teve melhores resultados e menor tempo de treino.

Após este teste, são testadas mais duas versões, uma que treina modelos com todas as variáveis e outra que utiliza apenas uma selecção de variáveis. A versão com as variáveis

seleccionadas teve um tempo de treino menor e resultados ligeiramente melhores nos modelos com melhor desempenho.

Os melhores modelos finais foram as versões mais simples de $LSTM$, GRU e $1D\ CNN$ + $LSTM$. O modelo combinado $1D\ CNN$ e $LSTM$ teve o menor tempo de treino de 97 segundos e o melhor desempenho com um $RMSE$ de 0,0617. Isto mostra que para este dataset, talvez devido ao tamanho do dataset ou ao número de variáveis, os modelos simples de *Deep Learning* superam os modelos mais complexos.

Contents

Abstract iii

Resumo v

Contents vii

List of Figures ix

1 Introduction 1
1.1 Diabetes 2
1.2 Dataset 2

2 State of the Art 9
2.1 Neural Networks 9
2.2 ANN - Artificial Neural Network 10
2.3 RNN - Recurrent Neural Network 14
2.4 CNN - Convolutional Neural Network 17

3 Preprocessing 23
3.1 Loading Dataset 23
3.2 Preprocessing 25
3.3 Re-sampler 27
3.3.1 preprocessing_ts 27
3.3.2 preprocessing_ts_sum 28
3.3.3 preprocessing_tbte 29
3.3.4 preprocessing_ex 31
3.3.5 preprocessing_bolus 32
3.3.6 preprocessing basal 34
3.4 Filler 36
3.4.1 Dataframe merge 37

4 Modeling 45
4.1 Data Transformation 45
4.2 Model Functions 47
4.3 1st Models 50

5 Results **57**
5.1 Personalized vs Generalized models 57
5.2 All vs Selected variables 59
5.3 Comparing Other Methods 64

6 Conclusions and Future Work **69**

Bibliography **70**

A Further results **75**
A.1 Additional Tables 75
A.2 Additional Figures 79

List of Figures

2.1 Machine Learning vs Deep Learning [9] 10
2.2 Artificial Neural Network Diagram [9] 11
2.3 Neuron Diagram [10] 11
2.4 Activation Function [11] 12
2.5 Cost and Gradient [12] 12
2.6 Hughe's Phenomenon [12] 13
2.7 RNN vs ANN [9] 14
2.8 RNN data on each iteration [9] 14
2.9 LSTM Architecture [17] 15
2.10 GRU Architecture [17] 16
2.11 CNN – Image Classification [9] 17
2.12 CNN – Image Classification 18
2.13 CNN – Convolving image 5x5 with a kernel = 2 and stride = 2 [9] 18
2.14 CNN – Convolving image 5x5 with a kernel = 2, stride = 1 and padding (2 on the left, 1 on the right) [20] 18
2.15 CNN – Convolving 3x3 image with a 2x2 filter Example [21] 19
2.16 CNN – Fully Connected Layer [21] 19
2.17 1D CNN – Standard vs Causal [22] 20
2.18 Causal Dilated CNN [22] 20
2.19 Temporal Convolutional Network [24] 21

3.1 CSV data for one patient in the train folder 24
3.2 Example of patients dictionary 24
3.3 Example of df_dict dictionary. 25
3.4 Types of bolus [26] 32
3.5 Dual Wave bolus [26] 33
3.6 Bolus split function for the 2nd (left) and 4th (right) row of Table 3.10 35
3.7 Example of df_all dictionary for the test folder. 40
3.8 Time series of glucose level and skin temperature for train dataset of patient 567. 42
3.9 Time series of glucose level and skin temperature for test dataset of patient 552. 43

4.1 Data transformation from training dataframe to arrays for patient 540. 46
4.2 df_final dictionary for patient 540. 47
4.3 Early Stopping. 49
4.4 Train and Validation Loss for two different models with 200 and 1000 epochs. 49
4.5 Model dictionary. 52

5.1 Plot of predictions for patient 552 model lstm1. 63
5.2 Plot of predictions for top 2 and bottom 2 performing patients using model convlstm1. 64

A.1 Plot of predictions for patient 570 and 591 using model lstm3. 79
A.2 Plot of predictions for patient 570 and 591 using model bigru1. 79
A.3 Plot of predictions for patient 570 and 591 using model gru3. 79
A.4 Plot of predictions for patient 570 and 591 using model bilstm2. 80
A.5 Plot of predictions for patient 570 and 591 using model convgru1. 80
A.6 Plot of predictions for patient 570 and 591 using model conv1. 80
A.7 Plot of predictions for patient 570 and 591 using model tcn2. 81
A.8 Plot of predictions for top 2 and bottom 2 performing patients using model lstm1. 81
A.9 Plot of predictions for top 2 and bottom 2 performing patients using model gru1. 82

Chapter 1

Introduction

Diabetes mellitus, commonly known as diabetes, is a metabolic illness in which the patient's blood sugar levels remain elevated for an extended period of time. If left untreated it causes major health problems such as cardiovascular disease, blindness, kidney failure, lower limb amputation and even death. The source of this condition is a dysfunction of the pancreas which causes it to produce insufficient or no insulin.

Nearly 537 million adults (20-79 years) had diabetes in 2021 and this number is predicted to rise over time. By 2030, the total number of people living with diabetes are predicted to be 643 million and by 2045, 783 million. In 2021, diabetes claimed the lives of 6.7 million people. Almost half of all adults with diabetes (240 million) are undiagnosed [1].

Since diabetes is such an important problem that affects an increasing number of people, it is critical to delay or avoid the disease by eating a balanced diet and exercising regularly. However, diabetes cannot always be prevented, and in some cases it must be controlled with insulin injections to maintain normal glucose levels. Insulin doses, however, are dependent on various parameters such as physical activity, nutrition, and the glycaemic level at that particular time. Because all of these vary depending on a person's habits, it is impossible to treat everyone with diabetes in the same manner. It is necessary to personalize the treatment in order for it to be effective, however, even if the patient follows a strict diet, glucose levels can quickly fluctuate.

To have a more personalized treatment, we need to understand what causes the glucose level to oscillate so that it can be forecasted when blood glucose levels are nearing dangerous levels, to avoid hypo and hyperglycemia. In this thesis we aim to predict these glucose oscillations with a Machine Learning analysis, using a dataset with various variables and glucose levels. The dataset used is the OhioT1DM Dataset [2] the contains

continuous glucose insulin taken, physiological sensor and self-reported life-event data collected over an eight-week period for 12 patients with type 1 diabetes.

1.1 Diabetes

Diabetes is a long-term chronic condition that affects how the body transforms food into energy, either to the pancreas' inability to create enough insulin or the body's inability to use the insulin produced. Insulin is a hormone that aids glucose absorption into cells for energy production. Hyperglycemia occurs when the body's ability to use or manufacture insulin is impaired, resulting in a rise in blood glucose levels, and it has been linked to several health concerns as well as organ and tissue failure over time.

Type 1, Type 2, and Gestational Diabetes are the three most common kinds of diabetes [3, 4].

Type 1 diabetes is also known as insulin-dependent diabetes or juvenile diabetes since it is most often initially diagnosed in children and young people. However, it can strike anyone at any age. Because the immune system targets and destroys the cells in the pancreas that create insulin, people with this type of diabetes are unable to produce any insulin. To keep blood glucose levels under control, these individuals must receive insulin injections on a daily basis. Type 1 diabetes affects 5-10% of people with diabetes and there is currently no way to prevent it[3].

Type 2 diabetes is the most prevalent type of diabetes, which arises when the body does not use insulin as it should. It most commonly affects middle-aged and older people, but it can strike anyone at any age[5]. Because the symptoms are not always obvious, testing is critical. Healthy habits, such as eating nutritious foods and exercising regularly, can help avoid or at least delay the onset of this illness.

Gestational diabetes can develop in pregnant women, which, most of the time, disappears after the baby is born. However, women affected and their children have an increased risk of developing type 2 diabetes later in life.

1.2 Dataset

The OhioT1DM Dataset, used in this thesis, collected data on several variables for 6 patients in 2018 and 6 patients in 2020 for a total of 12 patients over the course of eight weeks. This dataset was first made accessible for research purposes in 2018 for the first

Blood Glucose Level Prediction (BGLP) Challenge [6] and it contained information on the first six patients.

All patients are anonymous and therefore referred by random ID numbers. All patients were on insulin pump therapy with continuous glucose monitoring (CGM) that tracks the glucose level through a tiny sensor inserted under the skin, usually on the belly or arm [7]. The insulin pumps used by the patients were either the Medtronic 530G or the Medtronic 630G. Additionally, the patients used the Medtronic Enlite CGM sensors to monitor glycaemic levels. Patients reported life-event data via a custom smartphone app and physiological data was collected with a fitness band. The patients of 2018 used Basis Peak fitness bands while patients of 2020 wore Empatica Embrace.

The static data of each patient is shown in Table 1.1, this information includes patient ID, gender, age range, insulin pump model, sensor band type and cohort.

TABLE 1.1: Patient data from OhioT1DM 2018 and 2020 [2]

ID	Gender	Age	Pump Model	Sensor Band	Cohort
540	male	20–40	630G	Empatica	2020
544	male	40–60	530G	Empatica	2020
552	male	20–40	630G	Empatica	2020
567	female	20–40	630G	Empatica	2020
584	male	40–60	530G	Empatica	2020
596	male	60–80	530G	Empatica	2020
559	female	40–60	530G	Basis	2018
563	male	40–60	530G	Basis	2018
570	male	40–60	530G	Basis	2018
575	female	40–60	530G	Basis	2018
588	female	40–60	530G	Basis	2018
591	female	40–60	530G	Basis	2018

For the sequential data, the dataset includes [2]:

- CGM blood glucose level every 5 minutes
- Periodic self-monitoring Blood glucose level (finger sticks)
- Insulin doses, both bolus and basal
- Self-reported (SR) meal times with carbohydrate estimates

  SR times of exercise

  SR times of sleep

  SR times of work

SR times of stress

SR times of illness

- Data from the Basis Peak Band (5-minute aggregations):

  Heart Rate

  Galvanic Skin Response (GSR)

  Skin Temperature

  Air Temperature

  Step Count

  Sleep Time

- Data from the Empatica Embrace Band (1-minute aggregations):

  Galvanic Skin Response (GSR)

  Skin Temperature

  Magnitude of Acceleration

  Sleep Time

The data is divided in two sets for each patient, one for Training and one for Test examples.

TABLE 1.2: Train-Test split of glucose entries for each patient [2]

ID	BGLP Challenge	Training Examples	Test Examples
540	2020	11947	2884
544	2020	10623	2704
552	2020	9080	2352
567	2020	10858	2377
584	2020	12150	2653
596	2020	10877	2731
559	2018	10796	2514
563	2018	12124	2570
570	2018	10982	2745
575	2018	11866	2590
588	2018	12640	2791
591	2018	10847	2760

Each set contains the following data fields:

- **3 data fields** - Patient ID, insulin type and weight.

- **glucose level** - Continuous glucose monitoring (CGM) data every 5 minutes. Has two columns: *"ts"* - contains the information of the day, month, year, hour, minute, second; *"value"* - blood glucose in mg/dL.

- **finger stick** - Self-monitoring values of blood glucose. It has the same columns as *"glucose_level"* but has irregular time intervals as this test is taken by initiative of the patient.

- **basal** - Basal insulin rate that is injected continuously until a new basal rate is selected. Also has as columns *"ts"* and *"value"*. It Has irregular time intervals that depend on how long a basal rate was set.

- **temp basal** - A temporary basal insulin rate that supersedes the patient's normal basal rate. When the value is 0, this indicates that the basal insulin flow has been suspended. At the end of a temp basal, the basal rate goes back to the normal basal rate, basal. It has 3 columns: *"ts_begin"* - Initial time; *"ts_end"* - ending time; *"value"*. The number of rows depends on how many temp basal rates were set by the patient.

- **bolus** - Insulin delivered to the patient. It has 5 columns for 2018 patients and 4 columns for 2020 patients: *"ts_begin"*, *"ts_end"*, *"type"*, *"dose"* for both years and *"bwz_carb_input"* for 2018 patients. There are 4 types of insulin: Normal - delivers all insulin at once; Square - delivers all insulin over a period of time; Normal dual and Square dual that is a combination of the previous types. The fifth column has information of the carbs ingested similar to the *"meal"* field.

- **meal** - Self-reported meals, it has 3 columns: *"ts"*; *"type"* - type of meal; *"carbs"* - patient's estimate for the carbohydrates.

- **sleep** - Self-reported sleep with 3 columns: *"ts_begin"*; *"ts_end"*; *"quality"* - Patient's rating of the sleep quality: 1 - Poor; 2 - Fair; 3 - Good.

- **work** - Self-reported time of going to and from work with the subjective physical intensity of the activity. Columns: *"ts_begin"*, *"ts_end"* and *"intensity"*.

- **stressors** - Self-reported times of stress. It has 3 columns *"ts"*, *"type"* and *"description"* but only the time column has values. Some patient's have no entries on this dataset.

- **hypo event** - Self-reported hypoglycemic episodes, has only one column: *"ts"*. Some patient's have no entries on this dataset.

- **illness** - Self-reported time of when the patient felt ill. It has 4 columns: *"ts_begin"*, *"ts_end"* , *"type"* and *"description"* but only the first column has values. Some patient's have no entries on this dataset.

- **exercise** - Self-reported start time and duration of exercise with subjective Intensity. It has 5 columns: *"ts_begin"*, *"intensity"*, *"duration"*, *"type"* and *"competitive"* but the last two columns have no values.

- **basis heart rate** - Heart rate in bpm every 5 minutes. This data is only available for 2018 patients. It has 2 columns: *"ts"* and *"value"*.

- **basis gsr** - Galvanic skin response, also known as skin conductance or electrodermal activity every 5 minutes for 2018 patients and every 1 minute for 2020 patients. It has 2 columns: *"ts"* and *"value"*.

- **basis skin temperature** - Skin temperature, in Fahrenheit, every 5 minutes for 2018 patients and every 1 minute for 2020 patients. It has 2 columns: *"ts"* and *"value"*.

- **basis air temperature** - Air temperature, in degrees Fahrenheit every 5 minutes. This data is only available for 2018 patients. It has 2 columns: *"ts"* and *"value"*.

- **basis steps** - Step count, aggregated every 5 minutes. This data is only available for 2018 patients. It has 2 columns: *"ts"* and *"value"*.

- **basis sleep** - Sensor's report of the times when the patient was asleep. It has 4 columns: *"tbegin"*, *"tend"*, *"quality"* and *"type"*. The last column is always empty and the *"quality"* has a numeric estimate of sleep quality for 2018 patients while for 2020 has only 0s.

- **acceleration** - Magnitude of acceleration every 1 minute. This data is only available for 2020 patients, therefore, it will not be used.

All these variables can be used to make predictions on the glucose level. .

However, in traditional Machine Learning models feature engineering is done manually, while on Deep Learning these features are learned directly from the data, saving time and learning more complex data patterns [8].

There are several types of Deep Learning models:

- Artificial Neural Networks – ANN – Used for tabular data
- Recurrent Neural Networks – RNN – Used for sequential data
- Convolutional Neural Networks – CNN – Used for images

More complex neural networks are made in order to solve problems with these vanilla networks so it is needed to test more than one NN to know which one is better in a particular case. The next chapter will cover the differences in these NN.

A deep learning analysis of this dataset has already been made by several AI researchers, however, most of these analyses did not use all variables or had a poor preprocessing of the used variables. Additionally, most did not use any type of scaling of the data. The amount of information used to make predictions is also different, some researchers use only the previous 30 minutes of data to predict the following 30 minutes while others use the previous 1 or 2 hours.

The objective of this thesis is to find a NN that can accurately predict the glucose level after a certain period of time and compare the performance of a personalized (trained with only one patient) and generalized (trained with all patients) model.

Before using the models, a preprocessing of the data is needed as each variable has a different number of columns, rows and time stamps. Depending on the structure of each variable's dataframe, different types of preprocessing are applied and then all dataframes are merged into a single dataframe with only one column for the time stamps. To make predictions on the glucose level of the following 30 minutes, the information of all variables within the previous 24 hours will be used. All types of models with different numbers of layers will be compared and the model with the lowest error and training time will be selected.

Chapter 2

State of the Art

In this chapter, the different types of Deep Learning models used for time series forecasting are explained. Each model has some strengths and weaknesses, so these models are always being improved and new models are created in order to counter some of the weaknesses.

2.1 Neural Networks

Time series forecasting is a vital topic of study in a variety of fields, including weather forecasting, stock price forecasting, the number of customers at a store or hospital, among others.

There are numerous options for dealing with Time Series. ARIMA (*Autoregressive integrated moving average*) and SARIMA (*Seasonal ARIMA*) models are two of these alternatives.

These methods often require hand-engineered features, prepared by domain exerts. Most real-world problems have multiple variables as input so they are not suited to ARIMA that focus on univariate data. They are also focused on linear relationships and complete data so they do not perform well in predicting nonlinear patterns or predicting with missing values [8].

Deep Learning models, on the other hand, learn these features directly from the data, which saves time spent fine-tuning parameters and allows them to learn more complicated data patterns. These models are robust to noise and can learn even with missing values. They also support multivariate inputs and can learn nonlinear and complex relationships [8].

Having an image classification problem as an example, to extract features from an image, a strong understanding of the image's subject is required and it takes time to do so.

In Deep Learning, feature extraction is done automatically, as seen in Figure 2.1.

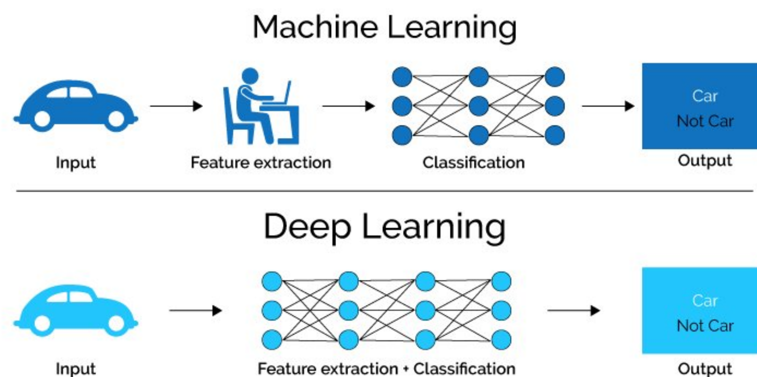


FIGURE 2.1: Machine Learning vs Deep Learning [9]

These Deep Learning models are called Neural Networks and there are several types of NN:

- ANN - Artificial Neural Network
- RNN - Recurrent Neural Network
- CNN - Convolutional Neural Network

2.2 ANN - Artificial Neural Network

An Artificial Neural Network is a collection of interconnected nodes known as perceptrons or neurons that are arranged in one or more layers. Each neuron receives a signal and processes it before passing it on to the next layer of neurons. The weight of each neuron and connection changes as the learning progresses. Because the inputs are exclusively processed in the forward direction, ANNs are also known as Feed-Forward Neural Networks (Figure 2.2).

The three layers of an ANN are also illustrated in the same image. The input layer receives the data, the hidden layer processes the inputs (which can be a single or more layers), and the output layer returns the output [10]. The structure of each neuron is represented in Figure 2.3.

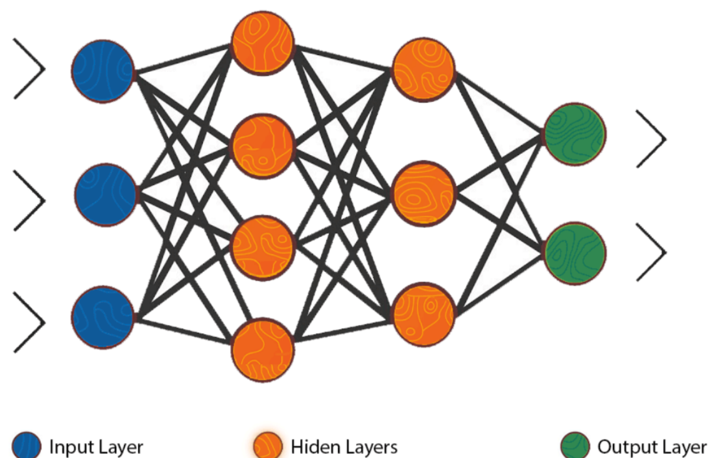


FIGURE 2.2: Artificial Neural Network Diagram [9]

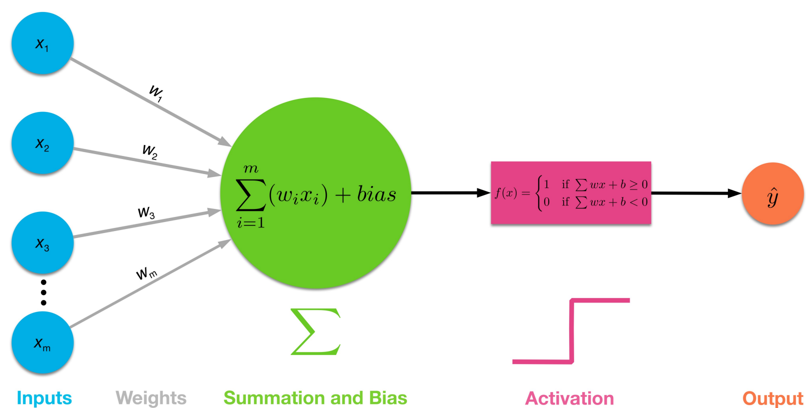


FIGURE 2.3: Neuron Diagram [10]

Each neuron receives all the input values (blue circles) multiplied by the weight assigned to each link (grey lines). The weight attributed to the neuron itself, known as bias, is added after the sum of all these values. Because the sum of all these values can quickly escalate as the number of layers grows, an Activation function is applied before transferring the value shown in the green circle on to the next neuron to keep the value between -1 and 1.

As Activation Functions, there are numerous alternatives, like those presented in Figure 2.4, these functions should be differentiable and continuous everywhere [11].

The first activation function, on the left side, is the ReLU or Rectified linear function. This function returns the max value between 0 and X, so, any negative number will be returned as 0 and the other numbers are kept the same.

The second function, in the middle, is the Sigmoid function, often known as the Logistic function, displayed in the middle. This function ensures the values on the green

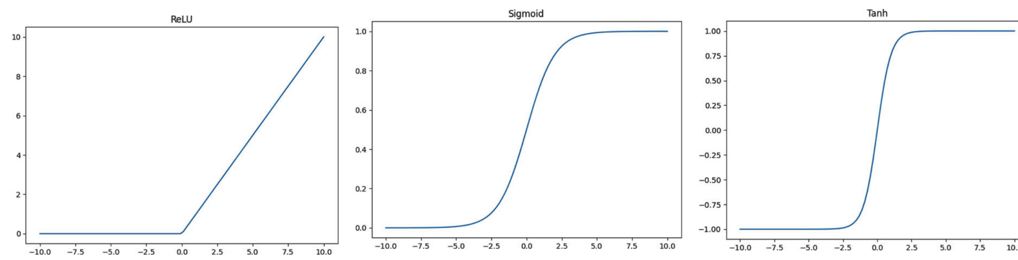


FIGURE 2.4: Activation Function [11]

circle stay between 0 and 1. It is linear for values near 0 but flattens off for larger values, resulting in numbers near +1 or 0.

The last function on the right is the Hyperbolic Tangent activation function, often known as TanH and behaves similarly to the sigmoid function, except that it limits the values between -1 and 1. The benefit of this function is that the negative inputs will be mapped strictly to negative and the only positive outputs are the positive inputs [11].

After the last hidden layer the output value is compared to the actual value. If the value is not what was expected then all the weights of bias and connections will be fine-tuned from the last layer till the first, this is called Back Propagation.

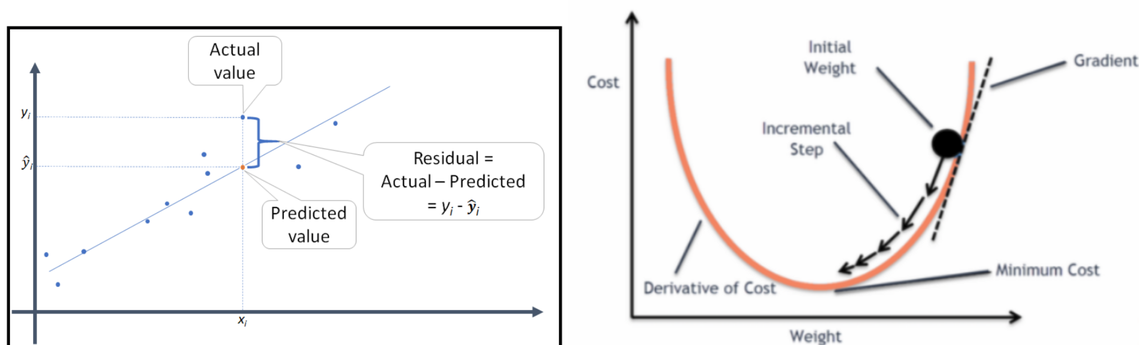


FIGURE 2.5: Cost and Gradient [12]

After the weights have been modified on the next iteration of the model train, a new cost (difference between the actual and predicted value) is determined, and this process is repeated until the cost is reduced, as illustrated in Figure 2.5. An NN's incremental step is one of its parameters. The training process will be substantially slower if the incremental step is too small, but the outcomes may not be optimal if the incremental step is too high, therefore, there must be a compromise between training time and results.

In this thesis, it is considered a regression problem and there are various functions to determine that cost (or loss function). Some of which are the Mean Squared Error

(MSE), Root Mean Squared Error ($RSME$) and the Mean Absolute Error (MAE) [8] which equations are the 2.1, 2.2 and 2.3, respectively.

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}-y_i)^2 \tag{2.1}$$

The MSE is obtained by summing the squared differences of between the observed value (y) and the predicted value ($\hat{y}$) and dividing by the number of test rows (N) [13].

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i-y_i)^2} \tag{2.2}$$

The $RMSE$ is obtained by using the square root of MSE.

$$\frac{1}{N}\sum_{i=1}^{N}|\hat{y}-y_i| \tag{2.3}$$

The MAE is similar to MSE but is instead obtained by summing the absolute value of the differences.

Unlike MAE, MSE and $RSME$ are more punishing of larger forecast errors [8]. This is due to the fact that the summation used is the squared differences, so these values will increase rapidly if the differences are larger.

While determining a minimum cost on a dataset with a small number of variables may be simple, this is not the case as the number of variables grows. This has a significant impact on the model's performance, known as the Hughe's Phenomenon and shown in Figure 2.6.

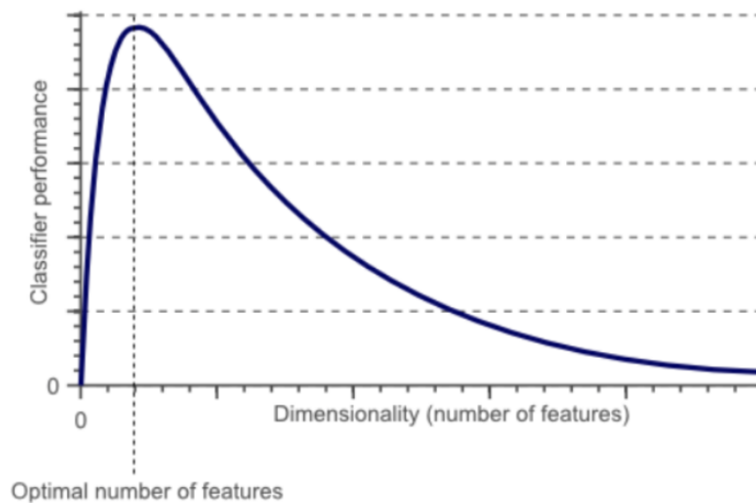


FIGURE 2.6: Hughe's Phenomenon [12]

A problem associated with ANNs is that they are not able to capture sequential information in the input data which is needed to analyse time-series [11].

2.3 RNN - Recurrent Neural Network

Recurrent Neural Networks (*RNNs*) address the challenge of sequential data through different feedback links in the architecture, enabling notion of state [9].

Figure 2.7 illustrates this difference, which is a recurrent link on each node. This loop ensures that sequential information is preserved, making RNN more appropriate for working with Time Series data.

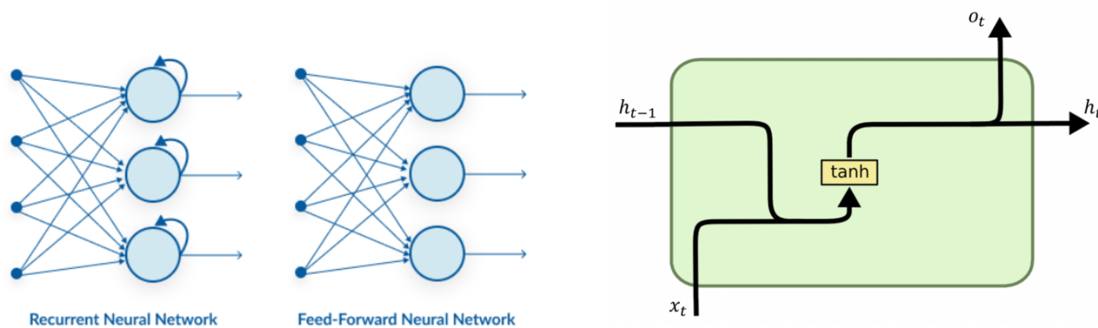


FIGURE 2.7: RNN vs ANN [9]

The data processed on each iteration is made up of the current iteration's input, x_t, and a combination of prior outputs, h_{t-1}. The output of this iteration is o_t. Figure 2.8 shows this impact as well.

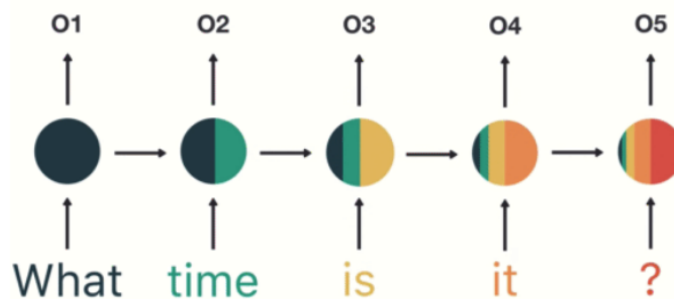


FIGURE 2.8: RNN data on each iteration [9]

Although RNNs are capable of perceiving sequential input, they are affected by the vanishing gradient problem [14], which causes short-term memory. The information from the first iteration gets smaller and smaller as the number of iterations increases, until it vanishes [11].

To tackle this problem, new types of RNN were created, LSTM (Long Short Term Memory) and GRU (Gated Recurrent Units).

As the name suggests, the Gated Recurrent Units have additional gates, compared to vanilla RNN, used to control the flow of information in the network. These gates are also present in LSTM models and are able to pass long sequences of information and use them to make predictions.

Figure 2.9 represents the basic LSTM Architecture, introduced by Hochreiter & Schmidhuber [15], that includes three additional gates. Each gate is a neural network with its own weights and biases. These new gates are the Forget Gate, Input Gate and Output Gate.

The cell state serves as the network's memory, storing information from previous iterations. The effects of short-term memory are reduced because information from earlier time steps is carried all the way [16]. As it goes through the gates, the data on this cell state can be added or erased. These gates are neural networks that decide whether or not to keep or forget information during training. Each gate has a Sigmoid activation function (red circles) that confines values between 0 and 1, which is what eliminates or adds information about cell states, values near 0 are forgotten or not contributed to the cell stated and values near 1 are added.

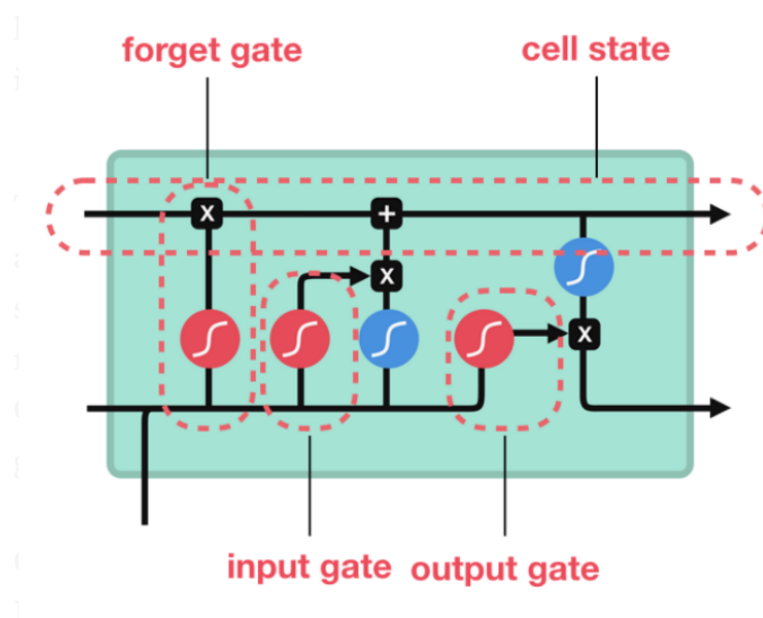


FIGURE 2.9: LSTM Architecture [17]

The data on the bottom black line is a combination of the prior hidden state (possible values for this time step) and the current input and it goes through all three gates.

When these values reach the Forget gate, they pass via the Sigmoid function; values near 0 indicate the information will be removed from the cell state, while values around 1 indicate that the information will be maintained.

The information will be processed through a Sigmoid function on the Input Gate to determine which information is significant (near 1) and which information is not important (near 0). A TanH activation function (blue circle) processes the same data, limiting the values to -1 and -1 in order to regulate the network. Then these regulated values are multiplied and essential values are picked and added to the cell state.

Finally, the Output gate determines what the next hidden state will be. The prior hidden state and the new input are passed through a Sigmoid state at this stage, after which the new cell state information (after the Forget and Input gates) is passed through a TanH function. Then, both TanH and Sigmoid output are multiplied in order to obtain the new hidden state which will be carried over to the next time step.

The GRU, introduced by Cho, et al. [18] is akin to the LSTM. GRUs does not have access to the cell state and instead relies on the hidden state to send data to the next time step. It contains two gates: the Update Gate and the Reset Gate, as shown in Figure 2.10.

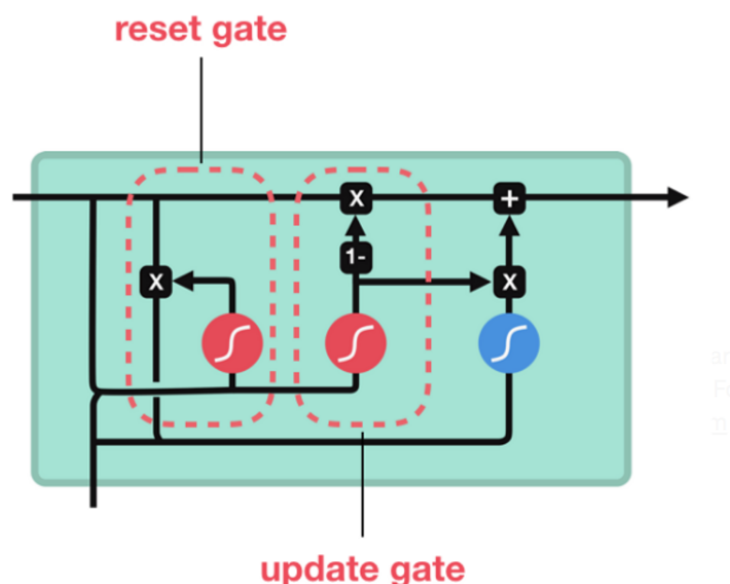


FIGURE 2.10: GRU Architecture [17]

This Update Gate serves as both a Forget and an Input Gate, deciding what information to erase and what information to add [16].

The Reset Gate is used to determine whether or not the prior hidden state is significant.

GRU is faster to train than LSTM because it has fewer gates, although in most cases, both are utilized and one is chosen based on performance. Both GRU and LSTM models have a Bidirectional model, the Bi-GRU and Bi-LSTM, that process the information in two directions: Backwards (future to past) and Forward (past to future).

2.4 CNN - Convolutional Neural Network

The Convolutional Neural Network, introduced by LeCun et al. [19], is another type of NN that is commonly used in image processing [8].

CNN neurons are filters, known as kernels, that extract relevant characteristics from input and produce a feature map. CNNs capture the spatial properties of an image and learn the relationship between the image and its arrangement of pixels, as demonstrated in Figure 2.11.

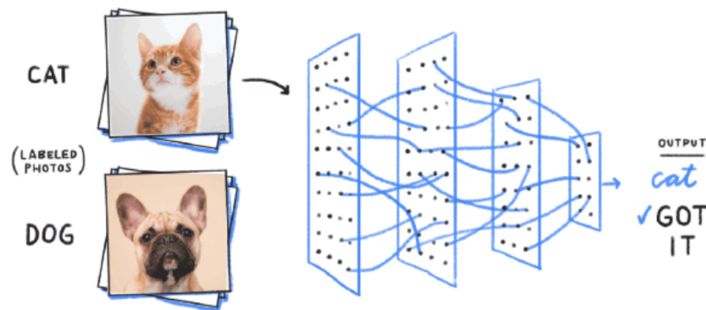


FIGURE 2.11: CNN – Image Classification [9]

In terms of image processing, CNNs outperform ANNs in determining whether an image belongs to a specific category, but is in a different location, slightly tilted, or simply zoomed (Figure 2.12). This happens because, unlike an ANN, CNN captures picture features with 2d mapping rather than processing an array of values.

The same filter is applied to various regions of an input in each layer [11]. Figure 2.13 shows a 5x5 image that has been turned into a 2x2 map by sliding the same 3x3 filter across the image by 2 pixels. The *CNN* parameters are the kernel, which determines the filter's window size and the stride, which determines the shift size as well as the padding.

To prevent reducing too much the image size in each layer, a padding is applied adding pixels to the data with 0s resulting in an output with higher dimensions as shown in Figure 2.14.

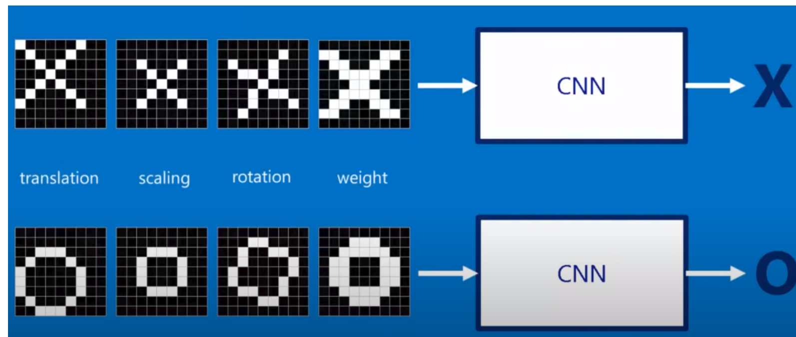


FIGURE 2.12: CNN – Image Classification

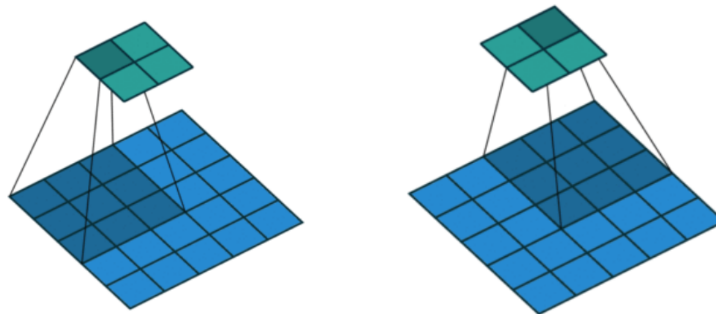


FIGURE 2.13: CNN – Convolving image 5x5 with a kernel = 2 and stride = 2 [9]

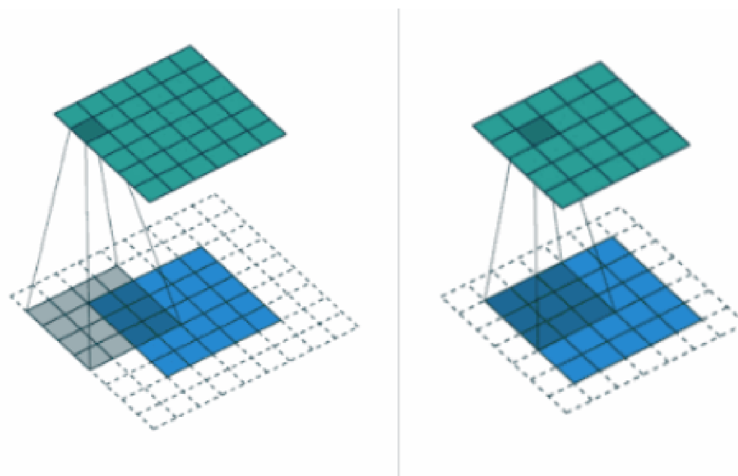


FIGURE 2.14: CNN – Convolving image 5x5 with a kernel = 2, stride = 1 and padding (2 on the left, 1 on the right) [20]

A clear example of this map is shown in Figure 2.15 where the convolution result is the sum of the values in each box multiplied by the values of the filter. As shown in the figure, the values on the first box, (1, 7, 11, 1) are element-wise multiplied by the filter (1, 1, 0, 1) and summed up (1 + 7 + 0 + 1). Then it shifts across the other values of the matrix and this process is repeated for each box in the input matrix [11].

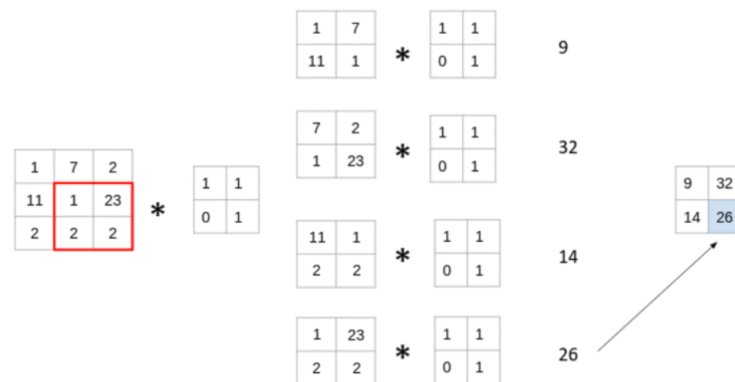


FIGURE 2.15: CNN – Convolving 3x3 image with a 2x2 filter Example [21]

The final type of layer (can be more than one layer), the Fully Connected Layers, are a traditional neural network that are applied after all of the mapping has been completed. The 2D matrix output is converted to an array, and each value is handled as a separate feature representing the entire image. As in an ANN, each feature is multiplied by the weight associated with each connection to each node, and then the bias of the neuron is added (Figure 2.16).

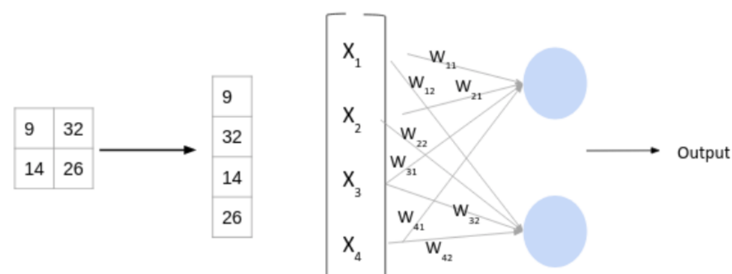


FIGURE 2.16: CNN – Fully Connected Layer [21]

Traditional CNNs, on the other hand, are not built for sequential data, so 1 Dimension Convolutional Neural Networks were created to overcome this problem [11]. The kernel (filter) of this new type of NN is made up of a set of sequential values centered around a point, as shown in Figure 2.17 (left). Standard 1D CNNs had an issue because they

relied on the input of x(1) and x(2) values to forecast the value of x(0), which is equivalent of attempting to predict the present while knowing the future. Causal CNN solves this problem by only using past values as input.

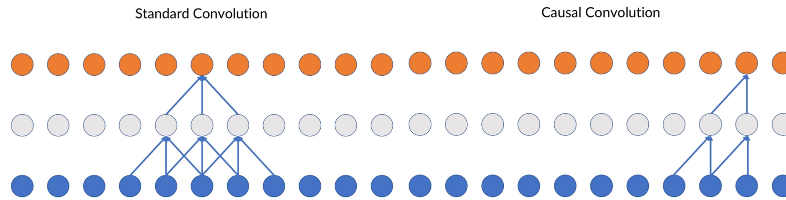


FIGURE 2.17: 1D CNN – Standard vs Causal [22]

It is possible to learn and train from the past in Causal CNN, but it is extremely difficult to learn and train from the distant past. In this case, a larger filter is required to capture information from more points, but this is not parameter efficient, or one additional layer for each point is required, which is a significant problem. Taking a music audio file with a sampling rate of 44.1 kHz as an example, this corresponds to 44,100 points per second of music, hence analyzing these files would demand a large number of layers. Because a large number of layers creates the vanishing/exploding gradient problem, the number of layers must be lowered, and Causal Dilated CNNs are adopted to do this [23]. These take into account points that are not immediately before x(0) but are separated by a dilation factor, d, and thus can take as input points from the distant past with fewer layers. However, it has no information about the values between, for example, the values of x(-1), x(-2), and x(-3), as shown in Figure 2.18.

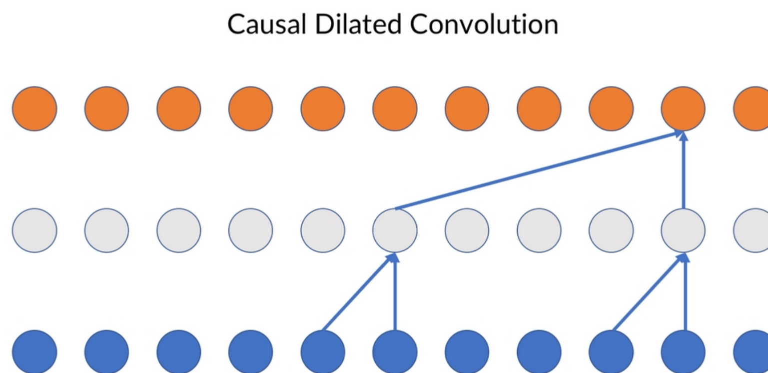


FIGURE 2.18: Causal Dilated CNN [22]

In order to have information of all points and learn from the far past, these Causal Dilated CNNs have several layers with different dilation factors as shown in Figure 2.19, this is called Temporal Convolutional Network (TCN).

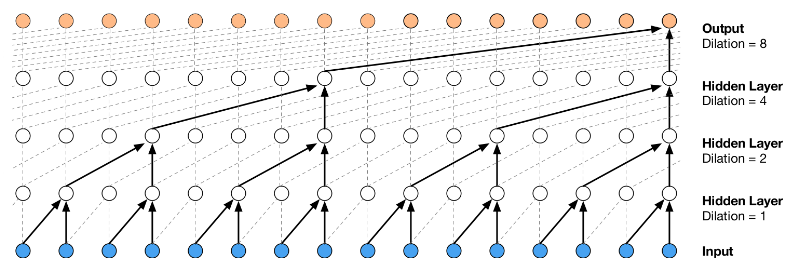


FIGURE 2.19: Temporal Convolutional Network [24]

The gradient flow is oriented vertically rather than horizontally in this sort of NN, hence the gradient is not time dependent like it is in RNN. This is significant because we can now treat this data in parallel rather than sequentially, resulting in a significant reduction in training time when utilizing GPUs [25]. They do not have the vanishing/exploding gradient problem since they have fewer layers, therefore they can learn from the far past without difficulty.

TCNs are not perfect, even with these advantages, because their receptive field is fixed *a priori* when the number of layers is defined, therefore values beyond this range are ignored when calculating the output at a certain position [25].

Most of these models will be tested in order to find the model that best predicts our dataset.

Chapter 3

Preprocessing

This chapter will cover the loading, exploration, preprocessing and merge of each variable's data. The variables of this dataset have different time intervals, different number of rows and some patients have more variables than the others which makes it difficult to gather every variable in a single dataframe for each patient. To do so, every variable needs to be preprocessed according to its characteristics.

3.1 Loading Dataset

The data provided by the OhioT1DM dataset is in XML format, so in order to work with this dataset the XML's data was parsed into CSV files.

As the data is originally split in train and test datasets, the same split is maintained after the parsing. The output of the parser consists in, for both test and train datasets, 12 folders corresponding to the 12 patients, each folder containing one CSV file for each variable. There are 18 variables for 2018 patients and 16 variables for 2020 patients.

In Figure 3.1 it is shown that the number and size of files for each patient is different as some patients may not contain information on some variables and some CSV files have more rows than others. This discrepancy in rows is explained by the fact that some variables like "*basis_gsr*" or "*glucose_level*" have rows for each interval of 1 or 5 minutes and variables like "*illness*" or "*exercise*" only have rows if the patient was ill or did exercise.

Before loading all CSV files, a dictionary called *patients* is made. The key of the dictionary is the number of each patient followed by "_testing" or "_training".

Then, for each key in the dictionary we save a list of pair of strings containing the name of each variable and the corresponding CSV file as presented on Figure 3.2.

540-ws-training.xml	19 items
544-ws-training.xml	19 items
552-ws-training.xml	19 items
559-ws-training.xml	18 items
559-ws-training-example.xml	18 items
563-ws-training.xml	18 items
567-ws-training.xml	19 items
570-ws-training.xml	18 items
575-ws-training.xml	18 items
584-ws-training.xml	19 items
588-ws-training.xml	18 items
591-ws-training.xml	18 items
596-ws-training.xml	17 items

acceleration.csv	1,3 MB
basal.csv	569 bytes
basis_gsr.csv	1,0 MB
basis_heart_rate.csv	39 bytes
basis_skin_temperature.csv	900,3 kB
basis_sleep.csv	2,0 kB
bolus.csv	10,6 kB
exercise.csv	1,3 kB
finger_stick.csv	4,3 kB
glucose_level.csv	259,2 kB
hypo_event.csv	563 bytes
illness.csv	57 bytes
meal.csv	8,1 kB
sleep.csv	1,8 kB
stressors.csv	44 bytes
temp_basal.csv	331 bytes
work.csv	24 bytes

FIGURE 3.1: CSV data for one patient in the train folder

```
key              value
'584_testing' : [ ( 'glucose_level' ,   'test/584-ws-testing.xml/glucose_level.csv' ) ,
                  ( 'basis_gsr'     ,   'test/584-ws-testing.xml/basis_gsr.csv' )      ,
                  ( 'meal'          ,   'test/584-ws-testing.xml/meal.csv' )           ,
                  ( ... )
'588_testing' : [ ( 'glucose_level' ,   'test/588-ws-testing.xml/glucose_level.csv' ) ,
                  ( 'basis_gsr'     ,   'test/588-ws-testing.xml/basis_gsr.csv' )      ,
                  ( 'meal'          ,   'test/588-ws-testing.xml/meal.csv' )           ,
                  ( ... )
( ... )
```

FIGURE 3.2: Example of patients dictionary

Using the *patients* dictionary to read each CSV file, the new dictionary, df_dict, is made. df_dict is a dictionary of dictionaries, each key corresponds to one patient ("584_*testing*") and the value is a dictionary containing a pandas dataframe for each variable as shown in Figure 3.3.

This structure of dataframes will be useful for the preprocessing since it is easy to specify which dataframe is used by using `df_dict['patient_id']['variable']`.

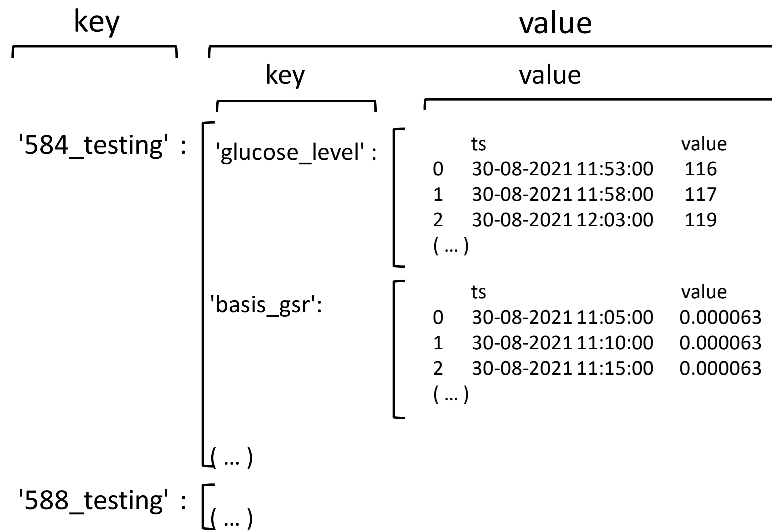


FIGURE 3.3: Example of df_dict dictionary.

3.2 Preprocessing

As said previously, different variables have different time intervals and, therefore, different timestamps. These time intervals are laid-out in Table 3.1, "X" means that this variable is not present in this batch of patients and the other variables are not periodic, and thus the interval between values varies. Since the variable "acceleration" was only introduced in 2020, the CSV of this variable was not used.

TABLE 3.1: Time interval of each variable for patients of 2018 and 2020 in minutes

Variable	2018	2020
glucose_level	5	5
basis_skin_temperature	5	1
basis_gsr	5	1
basis_air_temperature	5	X
basis_steps	5	X
basis_heart_rate	5	X
basis_sleep	5	?
acceleration	X	1

This inconsistency of time stamps is one of the reasons why it is not possible to merge all CSV files into one dataframe since there is not a timestamp that can contain the information of all variables. In order for this to be possible, all time stamps in every variable will be re-scaled to time intervals of 30 minutes that will enclose all values within this temporal window. This process will be different for each variable.

Due to this inconsistency of time intervals some variables have more rows than the others as shown in Table 3.2 that contains information of the minimum, mean and maximum number of events for each variable across all patients for train and test dataset. Highlighted rows are the patient's manual entries

TABLE 3.2: Min, Mean and Max number of events for each variable for Train and Test dataset.

	Train			Test		
	min	mean	max	min	mean	max
glucose_level	9080	11232.5	12640	2364	2645.2	2896
meal	32	149.4	265	0	31.2	54
basal	23	86.5	163	8	21.2	32
temp_basal	2	39.4	232	0	8.2	55
bolus	134	251.9	347	36	59.2	102
work	0	14.6	35	0	3.2	6
exercise	0	15.6	46	0	2.8	10
sleep	0	33.5	46	0	8.2	12
basis_sleep	9	512.5	1087	5	115.4	237
basis_skin_temperature	11822	25111.2	53219	2612	6538.9	13796
basis_gsr	11769	25099.2	53219	2606	6536.9	13796
stressors	0	0.5	3	0	0.1	1
illness	0	0.9	4	0	0.2	2
hypo_event	0	9.9	37	0	1	4
basis_heart_rate	0	6164.6	12980	0	1337.1	2720
basis_air_temperature	0	6664.9	12948	0	1338.9	2716
basis_steps	0	6739.9	13297	0	1460.4	2727
finger_stick	137	328.7	592	25	68.2	143

Some variables like "*glucose_level*" have similar min, mean and max because these values are obtained every 5 minutes for all patients and so the number of events depends only on the number of missing values and the duration of the collection of these values.

Other variables like "*basis_skin_temperature*" or "*basis_gsr*" have a great discrepancy between min, mean and max number of events due to the difference in time interval of these variables. As shown in Table 3.1, 2020 patients collected information about these variables every 1 minute while 2018 patients did so every 5 minutes thus 2020 patients will have much more rows than the others.

The "*basis_air_temperature*", "*basis_heart_rate*" and "*basis_steps*" have different values because these variables were not available in 2020 patients.

There is also a discrepancy in the number of rows in the "*basis_sleep*" variable, this is due to the fact that in 2020 there is only one row per night of sleep with the initial and wake-up time and in 2018 each night of sleep is divided into several rows.

Most of the other variables are self-reported so the difference in the number of rows depends on the frequency that the patient reported each variable.

For variables that have multiple values within 30 minutes, a mean or mode function needs to be applied in order to obtain only one value. For variables that do not always have at least one value in each 30 minute interval a filling function is applied. Thus, for the preprocessing of each variable at least two functions are used, a re-sample function and a fill function.

3.3 Re-sampler

In this section the several re-samplers are explained for each variable. Some variables can use the same preprocessor while others may need a specific function, depending on how the information of that variable is saved.

3.3.1 preprocessing_ts

One of the most simple re-sampler used was the `preprocessing_ts` that is applied on some variables such as *"glucose_level"* or *"basis_heart_rate"*. This function, as all other preprocessors, transforms the values on the time columns (*"ts"*) into datetime format using the function `datetime.strptime()`. After that, the values are re-sampled to 30 minute intervals applying the mean of the original values in that interval. An example of this step is shown in Table 3.3.

TABLE 3.3: Example of heart rate dataframe before and after the re-sample using the mean, in bpm.

Before		After	
ts	heart rate	ts	heart rate
18:18	70	18:00	75
18:23	75	18:30	82
18:28	80	19:00	80
18:33	85		
18:38	85		
18:48	80		
18:53	85		
18:58	75		
19:03	80		

This function will be applied to the following variables:

- glucose_level

- basis_skin_temperature
- basis_gsr
- basis_heart_rate
- basal
- basis_air_temperature
- finger_stick

3.3.2 preprocessing_ts_sum

In these kinds of variables like the heart rate, using the mean of the values within the 30 minutes interval is the best way to represent the value but this is not true for all variables. For example, in the case of "*basis_steps*" or "*meal*" variables, using the mean would result in obtaining the mean steps every 5 minutes or the mean of carbs of two inputs within 30 min. So, to re-sample these variables the function `.sum()` is used instead. If there is no value in every 30 minute interval, that time stamp will be added with a NaN that will be replaced later with a filler. An example of the function `preprocessing_ts_sum` is in Table 3.4

TABLE 3.4: Example of meal dataframe before and after the re-sample using the sum, in carbs.

Before		After	
ts	meal	ts	meal
18:03	20	18:00	30
18:25	10	18:30	NaN
20:09	60	19:00	NaN
21:48	15	19:30	NaN
		20:00	60
		20:30	NaN
		21:00	NaN
		21:30	15

Unlike all other variables, "*illness*", "*hypo_event*" and "*stressors*" do not have a "*value*" or "*intensity*" column, instead they have only a time column and time stamps at which the patient was ill, stressed or had an hypo_event. So, in each of these variables a new column "*value*" was added and set to 1 and the function `preprocessing_ts_sum` was applied to the dataframe. If there is no value in every 30 minute interval, that time stamp will be

added with a NaN that will be replaced later with a filler. In the *"illness"* dataframe there are two time columns but the *"ts_end"* is always empty, so this column is removed and the column *"ts_begin"* is renamed to *"ts"*. Since there is no more than one event within 30 minutes, the column *"value"* will then be a binary variable that states if a patient was ill, stressed or had an hypo_event in that time stamp. The Table 3.5 shows this transformation.

TABLE 3.5: Example of stressors dataframe before and after the re-sample using the sum

Before	Transformed		After	
ts	ts	value	ts	value
2021-09-22 13:59:00	2021-09-22 13:59:00	1	2021-09-22 13:30:00	1
2021-09-24 14:06:00	2021-09-24 14:06:00	1	2021-09-22 14:00:00	NaN
2021-10-08 18:12:00	2021-10-08 18:12:00	1	...	...
			2021-10-08 18:00:00	1

The variables that will use the *preprocessing_ts_sum* function are the following:

- basis_steps
- meal
- stressors
- illness
- hypo_event

3.3.3 preprocessing_tbte

While most of the variables have only one column for time, *"ts"*, some variables like *"work"*, *"sleep"* and others have two columns, *"ts_begin"* and *"ts_end"*. These kinds of variables with two time columns need some kind of treatment in order to have only one so the re-sample can be applied.

To do this, a dummy dataframe is created with the same columns as the original but replacing both time columns with only one, *"ts"*. This *"ts"* column has datetime values, every 5 minutes, from the first time stamp on the column *"ts_begin"* until the last time stamp on the column *"ts_end"* ensuring that it contains the entire range of the original dataframe. Then, a for loop is made for each line in the dummy dataframe that checks if each time stamp in the *"ts"* column is contained in the intervals between *"ts_begin"* and *"ts_end"* of the original dataframe. If it is contained, then it means that in the time stamp

"*ts*", the patient is working or sleeping, therefore the value of the "*intensity*" column for this time stamp is the same as the original dataframe, otherwise the value is 0. There are some rows that have "*ts_begin*" = "*ts_end*" and since it is not possible to know how long the patient worked or slept, these rows are removed. Then, this dummy dataset is re-sampled to 30 minutes and a mode function is applied to select the most occurring value within 30 min. This whole process is shown in Table 3.6.

TABLE 3.6: Example of work dataframe before and after the re-sample using the mode.

(A) Transformation into dummy dataframe.

Before			Dummy	
ts_begin	ts_end	intensity	ts	intensity
2021-08-31 08:14:00	2021-08-31 17:15:00	4	2021-08-31 08:14:00	4
2021-09-01 07:15:00	2021-09-01 17:34:00	5	2021-08-31 08:19:00	4
...	...	...	2021-08-31 08:24:00	4
2021-10-14 08:06:00	2021-10-14 17:00:00	5	...	...
			2021-08-31 17:14:00	4
			2021-08-31 17:19:00	0
			2021-08-31 17:24:00	0
			...	...
			2021-09-01 07:14:00	0
			2021-09-01 07:19:00	5
			2021-09-01 07:24:00	5
			...	...
			2021-10-14 16:54:00	5
			2021-10-14 16:59:00	5

(B) Re-sample of dummy dataframe.

Dummy		After	
ts	intensity	ts	intensity
2021-08-31 08:14:00	4	2021-08-31 08:00:00	4
2021-08-31 08:19:00	4	2021-08-31 08:30:00	4
2021-08-31 08:24:00	4	2021-08-31 09:00:00	4
...	...	...	...
2021-08-31 17:14:00	4	2021-08-31 17:00:00	4
2021-08-31 17:19:00	0	2021-08-31 17:30:00	0
2021-08-31 17:24:00	0	2021-08-31 18:00:00	0
...	...	...	...
2021-09-01 07:14:00	0	2021-09-01 07:00:00	0
2021-09-01 07:19:00	5	2021-09-01 07:30:00	5
2021-09-01 07:24:00	5	2021-09-01 08:00:00	5
...	...	...	...
2021-10-14 16:54:00	5	2021-10-14 16:00:00	5
2021-10-14 16:59:00	5	2021-10-14 16:30:00	5

The "*basis_sleep*" variable had an additional step because of the distinction between the dataframes of batch of patients. In 2018, patients wore Basis Peak fitness bands, these bands saved information about sleep quality and the time stamps were irregular as shown in Table 3.7 in contrast with 2020 patients that wore Empatica Embrace that did not save any information about quality and saved only the beginning and the end of the sleep. So, to make it easier to compare the dataframe of both batches, the quality value was always set to 1, this way the "*basis_sleep*" variable will be a binary variable that defines if the patient was asleep or not.

TABLE 3.7: Comparing basis_sleep dataframe between patients from 2018 and 2020.

2018			2020		
ts_begin	ts_end	qual	ts_begin	ts_end	qual
30-08-2021 22:24:00	30-08-2021 22:26:00	93	14-05-2025 07:02:00	14-05-2025 08:34:00	0
30-08-2021 22:26:00	30-08-2021 22:45:00	93	14-05-2025 21:59:00	15-05-2025 04:31:00	0
...	...	...	...	...	...
14-10-2021 23:47:00	15-10-2021 00:03:00	92	28-06-2025 22:11:00	29-06-2025 04:06:00	0

The `preprocessing_tbte` function will be applied on the variables "*work*", "*sleep*", "*basis_sleep*", "*temp_basal*" and "*exercise*".

3.3.4 preprocessing_ex

One other variable that needs preprocessing in terms of the time columns is the variable "*exercise*". This variable, unlike the others, instead of having a pair "*ts_begin*" and "*ts_end*" has a column "*ts*" and a column "*duration*". To solve this problem, the "*ts*" column was renamed to "*ts_begin*" and a empty "*ts_end*" column was added. To fill the new column, the values on "*duration*" were added to the time stamp of "*ts_begin*". Afterwards the column "*duration*" is removed. After this step, the `preprocessing_ex` function, the transformed "*exercise*" dataframe has now the same format as the variable "*meal*" and can be re-sampled the same way. This whole process is displayed on Table 3.8.

TABLE 3.8: Example of exercise dataframe before and after the transformation and re-sample using the mode.

Before			Transformed			After	
ts	duration	intensity	ts_begin	ts_end	intensity	ts	intensity
16:45:00	61	10	16:45:00	17:46:00	10	16:30:00	10
						17:00:00	10
						17:30:00	0

3.3.5 preprocessing_bolus

Lastly, the function for the variable *"bolus"* which sample is presented on Table 3.9.

TABLE 3.9: Sample of bolus dataframe from a 2018 patient.

ts_being	ts_end	type	dose	bwz_carb_input
07-12-2021 07:36:54	07-12-2021 07:36:54	normal dual	8.0	102
07-12-2021 07:41:58	07-12-2021 08:11:58	square dual	6.4	0
07-12-2021 18:31:52	07-12-2021 18:31:52	normal dual	4.0	65
07-12-2021 18:34:32	07-12-2021 19:04:32	square dual	4.1	0
07-12-2021 22:19:46	07-12-2021 22:19:46	normal	0.7	0

The format of this dataframe is different between patients as the 2018 patients have the column *"bwz_carb_input"* and 2020 patients do not, however this column can be removed as it has the exact same information as the *"meal"* dataframe. For this variable there are 4 distinct values for *"type"*: normal, square, normal dual and square dual. These types of bolus are shown in Figure 3.4, the normal bolus is an instant injection of insulin, in the square option the insulin is injected over a period of time and finally the normal dual and square dual that is a combination of the previous ones.

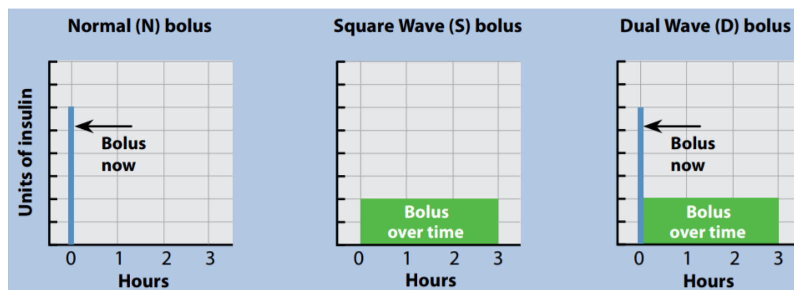


FIGURE 3.4: Types of bolus [26]

Figure 3.5 is a print screen of the Medtronic 630G [26] insulin pump and explains how the Dual wave bolus works. First the dose of insulin units is selected, then the patient selects the percentage of insulin to be taken now as normal dual bolus and the rest is taken over a period of time as squared dual bolus. This is also shown in Table 3.9, normal and normal dual bolus have *"t_begin"* = *"t_end"* since the insulin is taken instantly and square dual has *"t_begin"* $\neq$ *"t_end"*.

Since all other variables are re-scaled to 30 minutes time intervals, if the square or square dual duration were 30 minutes or bellow there would be no need to do a preprocessing. This is due to the fact that the insulin units taken within 30 minutes would be the same if it was instantaneously or over 30 minutes. In fact, most of the duration for

FIGURE 3.5: Dual Wave bolus [26]

the square and square dual is 30 minutes, however that is not always so a preprocessing needs to be done.

An example of this process is demonstrated in Figure 3.6. The first step is to check the duration of the rows with "*type*" = square and divide by 30 minutes, this will give the number of "full cycles" (1 cycle on the left example / 3 on the right example) and the duration of the "incomplete cycle" (15 / 0 minutes) if the duration is not multiple of 30. With these values it is possible to obtain the value of the dose injected during the incomplete cycle (3 / 0 insulin units). This value is then subtracted to the total insulin dose (9 / 18 iu) which is then divided by the number of complete cycles (1 / 3 cycles) giving the insulin dose injected each full cycle (6 / 6 iu). After the update of the dose value on the original row, the new rows are added. On the left example, since there is only one full cycle there is no need to duplicate this row, instead a new row with the incomplete cycle's dose is added and the time *ts_1* of this row is obtained by adding 30 minutes to the original time stamp *ts_0*. In the right example there is no incomplete cycles so no new rows are added, the original row is duplicated 2 times (number of full cycles minus 1) and the time stamps of these rows are updated adding 30 and 60 minutes respectively. After this split of the square and square dual inputs, these doses are already separated by 30 minute intervals and so the columns "*ts_end*" and "*type*" can be removed. Finally, this dataframe is re-sampled to 30 minutes using the `.sum()` function aggregating values within each time stamp. Table 3.10 shows both transformations of the "*bolus*" dataframe during the `preprocessing_bolus` function.

TABLE 3.10: Example of bolus dataframe before, after the split and after sum.

(A) Bolus Split.

Before				After Split	
ts_begin	ts_end	type	dose	ts	dose
15:01:30	15:01:30	normal	3	15:01:30	3
15:10:00	15:55:00	square	9	15:10:00	6
15:12:50	15:42:50	square	1.2	15:12:50	1.2
18:16:20	19:46:20	square	18	15:40:00	3
				18:16:20	6
				18:46:20	6
				19:16:20	6

(B) Re-sample with sum.

After Split		After Sum	
ts	dose	ts	dose
15:01:30	3	15:00	10.2
15:10:00	6	15:30	3
15:12:50	1.2	16:00	NaN
15:40:00	3	...	...
18:16:20	6	18:00	6
18:46:20	6	18:30	6
19:16:20	6	19:00	6

3.3.6 preprocessing basal

The only variables left are the "*basal*" and "*temp_basal*". Since they contain information about the basal insulin these two variables will be merged into one. When inside the timestamp of the "*temp_basal*", the "*basal*" will be overwritten. To do this, each variable will be re-sampled into 5 minute intervals instead of 30. For the variable "*temp_basal*" the function `preprocessing_tbte` was used as shown in Table 3.11. That function turns the temporal scale dataset of this variable into a 5 minute interval that contains NaN if this timestamp was outside the "*ts_begin*" and "*ts_end*" or the temporary basal value if it was inside that range.

The function `preprocessing_ts` was applied to the "*basal*" variable with a 5 minute time interval. Since the basal value is constant until the next basal value is set, the values of each will be filled with the forward filler as shown in Table 3.13. After this step both variables are re-scaled to 5 minutes and if, for a given timestamp there is a value on the "*temp_basal*" (other than NaN), then for that timestamp the "*basal*" value will be updated

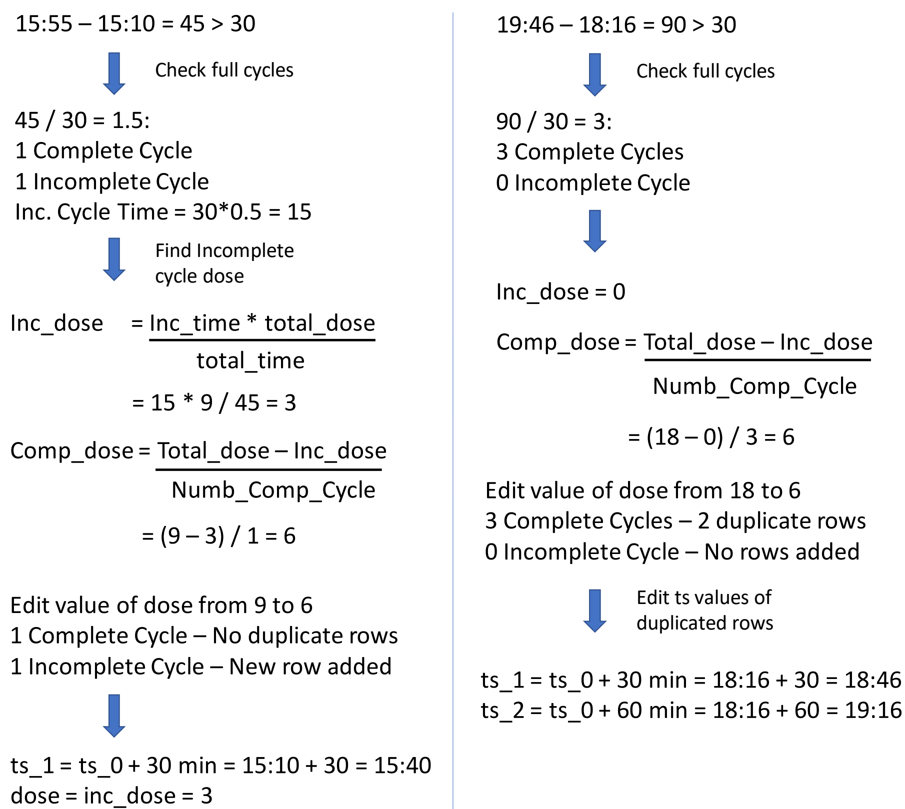


FIGURE 3.6: Bolus split function for the 2nd (left) and 4th (right) row of Table 3.10

TABLE 3.11: Sample of the *"temp_basal"* dataframe for patient 544.

Before			After	
ts_begin	ts_end	value	ts	value
24-06-2027 12:11:39	24-06-2027 12:37:28	0	2027-06-24 12:10:00	NaN
25-06-2027 07:58:30	25-06-2027 08:19:09	0	2027-06-24 12:15:00	0
...	...	...	2027-06-24 12:20:00	0
			2027-06-24 12:25:00	0
			2027-06-24 12:30:00	0
			2027-06-24 12:35:00	0
			2027-06-24 12:40:00	NaN
			...	...

with the temporary one. After this merge, the *"basal"* variable is re-sampled to 30 minute time intervals using the mean. This whole process is demonstrated in Table 3.12.

TABLE 3.12: Example of basal dataframe before, after the 5 min re-sample, after the merge with temp_basal dataframe and after the 30 min re-sample for patient 544.

Before		5 min Resample		After temp merge		Final	
ts	value	ts	value	ts	value	ts	value
24-06-2027 00:00:00	1.7	00:00:00	1.7	00:00:00	1.7	00:00:00	1.7
24-06-2027 03:00:00	1.8	00:05:00	1.7	00:05:00	1.7	00:00:00	1.7
24-06-2027 08:00:00	1.5	...	...	...	...	...	...
24-06-2027 11:00:00	1.0	12:00:00	1	12:00:00	1	11:30:00	1.0
24-06-2027 16:00:00	1.3	12:05:00	1	12:05:00	1	12:00:00	0.3
...	...	12:10:00	1	12:10:00	0	12:30:00	0.8
		12:15:00	1	12:15:00	0	13:00:00	1
		...	...	...	...	...	...

3.4 Filler

After re-sampling all variables the NaN values need to be substituted. To fill variables like *"basis_skin_temperature"* or *"basis_heart_rate"* the missing values will be filled with the last known value. In these cases, even though some values might be missing, it does not make sense to fill these values with 0 because this would mean that the heart rate at that specific time is 0, which is not possible. To counter this, in the re-sampled time intervals that have no values, it is assumed that the value is the last known value. An example of this fill function is presented on Table 3.13.

TABLE 3.13: Example of heart rate dataframe before and after the Fill Function $ffill$ in bpm.

t	Before	After
18:00	80	80
18:30	NaN	80
19:00	NaN	80
19:30	77	77
20:00	NaN	77
20:30	75	75
21:00	80	80

The other option for the filler, which is the most common one, is filling the missing values with 0. Taking as example the variable *"meal"* in which a patient ate 30 carbs at 18:30 and 60 carbs at 20:00, if the NaN were replaced with the last known value it would mean that this patient ate 30 carbs at 18:30, 30 carbs at 19:00, 30 carbs at 19:30 and 60 carbs at 20:00 and this is not what really happened. In these cases the correct way to fill the missing values is replacing them with 0 as shown in Table 3.14.

TABLE 3.14: Example of meal dataframe before and after the Fill Function *fillna* in carbs.

t	Before	After
18:00	30	30
18:30	NaN	0
19:00	NaN	0
19:30	NaN	0
20:00	60	60
20:30	NaN	0
21:00	NaN	0

3.4.1 Dataframe merge

A summary of the re-samplers and fillers used is in Table 3.15, these dataframes will then be stored in a new dictionary, *df_new* with the exact same format as the *df_dict*.

TABLE 3.15: Summary of functions used in each variable.

Variable	Prepro_func	Fill_func
glucose_level	ts	last value
basis_skin_temperature	ts	last value
basis_gsr	ts	last value
basis_heart_rate	ts	last value
basal	ts	last value
basis_air_temperature	ts	last value
finger_stick	ts	fillna
basis_steps	ts_sum	fillna
meal	ts_sum	fillna
stressors	ts_sum	fillna
illness	ts_sum	fillna
hypo_event	ts_sum	fillna
work	tbte	fillna
sleep	tbte	fillna
basis_sleep	tbte	fillna
temp_basal	tbte	none
exercise	ex + tbte	fillna
bolus	bolus	fillna

After these functions were applied, the only important columns in each dataframe are the time column *"ts"* and the columns containing the values like *"value"*, *"intensity"* and *"quality"* which will be renamed to the name of the variable. All other columns in all dataframes are either empty like *"illness"*:*"description"* or contain unnecessary information like *"bolus"*:*"bwz_carb_input"* so several columns from certain dataframes will be removed. The removed columns for each dataframe are the following:

- *"meal"* : *"type"*

- *"basis_sleep"* : *"type"*
- *"stressors"* : *"type"* ,*"description"*
- *"illness"* : *"type"*, *"ts_end"*, *"description"*
- *"bolus"* : *"type"*, *"ts_end"*, *"bwz_carb_input"*
- *"exercise"* : *"type"*, *"duration"*, *"competitive"*

Then, a new dictionary, *df_all*, is made that has a key for each patient and as value a copy of *"glucose_level"* dataframe. Then all other variables' dataframes are merged into this one according to the time stamps of *"glucose_level"*. Only rows within the first and last *"ts"* of this dataframe will be added. In case a variable has no rows within these time stamps the value NaN will be given.

This merge is detailed in Table 3.16, the first and last row of *"meal"* dataset do not cover the entire range of *"glucose_level"* dataframe so some rows are filled with NaN values. In the case of *"basis_sleep"*, the range of this dataframe is bigger than *"glucose_level"* so some rows are cut-off.

If a dataframe is empty or does not exists the join will fail, so, before merging, non existing dataframes like *"basis_steps"* for 2020 patients will be created. Some patients have empty dataframes with the wrong structure such as *"basis_heart_rate"* for patient "596" as shown in Table 3.17 and to solve this, all empty dataframes will have their columns changed to match the format of the original non-empty dataframes.

After this step, a new row will be added to all empty dataframes, otherwise the join will fail. This row will have a time stamp of $'01-01-2000\ 00:00:01'$ and all other values as 0, as demonstrated in Table 3.18.

The prepocessing functions will also be applied to these dataframes so the columns that are not needed are dropped and the time columns are transformed to datetime format.

Since these time stamps are before the first *"ts"* of *"glucose_level"*, the merge will succeed but add only NaN values to the new dataframe.

After the merge and as seen in Table 3.16, some variables will have NaN so these new dataframes will have values filled with either 0 or using `.ffill()` and `.bfill()` functions according to the fill functions on Table 3.15. The function `.bfill()` backward fills the NaN values with the first know values in case the first known value of a variable

TABLE 3.16: Example of "*glucose_level*", "*meal*" and "*basis_sleep*" dataframes from a patient in the dictionary *df_new* and after the merge in the dictionary *df_all*.

(A) Example of dataframes in the dictionary *df_new*.

Before					
ts	glucose	ts	meal	ts	basis_sleep
2021-08-30 11:30	116.5	2021-08-30 13:00	60	2021-08-30 7:30	1
2021-08-30 12:00	113.3	2021-08-30 13:30	0	2021-08-30 8:00	0
2021-08-30 12:30	124.2	2021-08-30 14:00	0	2021-08-30 8:30	0
2021-08-30 13:00	147.0	...	...	...	...
2021-08-30 13:30	151.7	2021-08-30 17:30	15	2021-08-30 11:30	0
...	...	...	...	...	...
2021-10-14 22:00	199.3	2021-10-14 13:00	50	2021-10-14 23:00	0
2021-10-14 23:00	168.7	...	...	2021-10-14 23:30	1
2021-10-14 23:30	140.6	2021-10-14 18:00	20	2021-10-15 00:00	1

(B) Example of the final dataframe in the dictionary *df_all*.

After			
ts	glucose	meal	basis_sleep
2021-08-30 11:30	116.5	NaN	0
2021-08-30 12:00	113.3	NaN	0
2021-08-30 12:30	124.2	NaN	0
2021-08-30 13:00	147.0	60	0
2021-08-30 13:30	151.7	0	0
...	...	...	...
2021-10-14 22:00	199.3	NaN	0
2021-10-14 23:00	168.7	NaN	0
2021-10-14 23:30	140.6	NaN	1

TABLE 3.17: Example of "*basis_heart_rate*" dataframe from patient "596" and "588".

596					588	
ts	intensity	type	duration	competitive	ts	value
					30-08-2021 11:04:00	83
					...	...

is at a time stamp later than the first "*ts*" of "*glucose_level*". For variables that contain only NaN values, `.ffill()` and `.bfill()` functions will not work, in these cases the variables will be filled with 0.

The final structure of the *df_all* dictionary is displayed in Figure 3.7. This whole process takes about 6 minutes for the train dataset and 1 minute for the test dataset.

The Table 3.19 shows the total duration of the dataset for each patient for train and

TABLE 3.18: Example of the updated dataframes for patient "596".

basis_heart_rate		work		
ts	value	ts_begin	ts_end	intensity
01-01-2000 00:00:01	0	01-01-2000 00:00:01	01-01-2000 01:00:00	0

key value

'584_testing' :

ts	glucose	meal	basis_sleep	work
2021-08-30 11:30	116.5	0	0	0
2021-08-30 12:00	113.3	0	0	0
2021-08-30 12:30	124.2	0	0	4
2021-08-30 13:00	147.0	60	0	4
2021-08-30 13:30	151.7	0	0	4
(...)				

'588_testing' : (...)

FIGURE 3.7: Example of df_all dictionary for the test folder.

test dataset. These numbers were obtained subtracting the timestamp for the first row to the last row. The datasets contain values corresponding to 38 days up to 47 days in the Train dataset and 9 to 13 days in the Test dataset. Patient 552 has the smallest Train dataset and the biggest Test dataset but this will not be changed in order to preserve the same information in the original CSVs.

TABLE 3.19: Duration of the collection of values for each patient for train and test dataset.

Patient	Train Duration	Test Duration
540	45 days 12:00:00	10 days 15:00:00
544	43 days 23:30:00	10 days 21:30:00
552	38 days 12:30:00	13 days 17:00:00
567	46 days 23:30:00	9 days 23:00:00
584	45 days 23:30:00	10 days 09:30:00
596	47 days 07:30:00	10 days 00:00:00
559	41 days 22:30:00	9 days 23:30:00
563	45 days 11:00:00	9 days 08:00:00
570	40 days 07:30:00	9 days 23:30:00
575	45 days 11:30:00	9 days 10:30:00
588	45 days 12:00:00	10 days 00:00:00
591	44 days 06:30:00	9 days 21:00:00

To check the amount of missing values affected by the filling functions some variables were preprocessed and then the number of NaN was counted. With the number of NaN it is possible to obtain the percentage of missing values in each variable. These values were

only obtained for 5 variables: *"glucoselevel"*, *"skin_temperature"*, *"gsr"*, *"heart_rate"* and *"air_temperature"*. The other variables are self report and it makes no sense to check the amount of rows affected by the filling functions because it is impossible to know if an NaN is a missing value or if the patient did not eat, exercised or was ill at that time stamp. The amount of missing values will not be obtained for variables like work or sleep because if a patient did not self reported the *ts_begin* and *ts_end* it is not possible to know how many rows would be affected because the patient could have slept any number of hours or even did not go to work that day. The Table 3.20 contains the percentage of missing values for the 5 variables for the 2018 patients and for the 3 variables for 2020 patients for both train and test datasets.

TABLE 3.20: Percentage of missing values of train and test datasets after 30 min re-sample and before the filling functions.

(A) Train missing values.

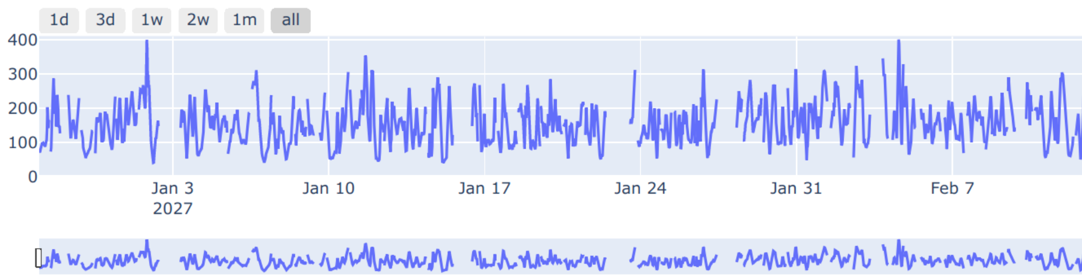
variable	559	563	570	575	588	591	540	544	552	567	584	596
glucose level	9.1	6.7	4.8	7	3.2	14.2	7.7	15.3	16.5	18	6.4	19.3
skin temperature	1.6	1.7	0.9	4.9	1.1	7.7	51.8	7.6	50.4	36.5	26.1	38.7
basis gsr	1.6	1.7	0.9	4.9	1.1	7.8	51.8	7.6	50.3	36.5	26.1	38.7
heart rate	1.6	1.7	0.9	4.9	1.1	7.8						
air temperature	1.6	1.7	0.9	4.9	1.1	7.7						

(B) Test Missing Values.

variable	559	563	570	575	588	591	540	544	552	567	584	596
glucose level	10.4	4	3.1	3.3	2.7	2.5	4.3	12.8	39	15	8.6	7.6
skin temperature	3.8	4.2	0.4	0.4	0.9	5.6	36.7	6.4	35.3	29	12.6	31
basis gsr	3.8	4.2	0.4	0.7	0.9	5.6	36.7	6.4	35.3	29	12.6	31
heart rate	3.8	4	0.2	0.2	0.7	5.6						
air temperature	3.8	4.2	0.4	0.4	0.9	5.6						

Overall the patients from 2020 contain much more missing values than the previous batch of patients. For the training dataset of 2018 only one patient had more than 10% of missing values for the glucose level while for 2020 4 patients had more than 15% of missing values.

The distribution of some of these missing values is presented in Figures 3.8 and 3.9. As shown in the first figure, the intervals of missing values for the glucose level for the train dataset of patient 567 are usually short with a duration of a few hours. The missing values of the variable *"basis_gsr"* are exactly the same as the skin temperature. In this case



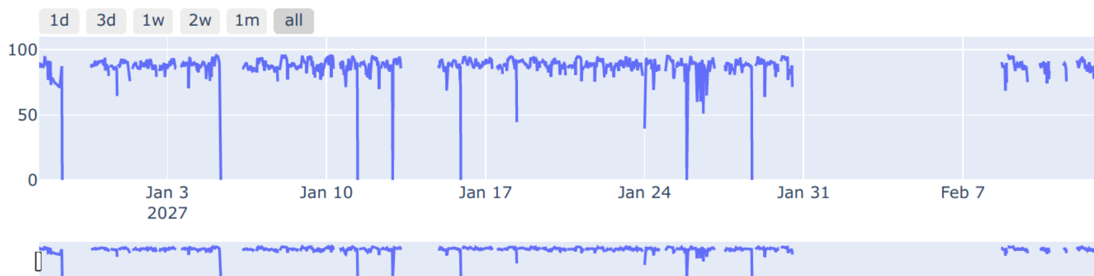
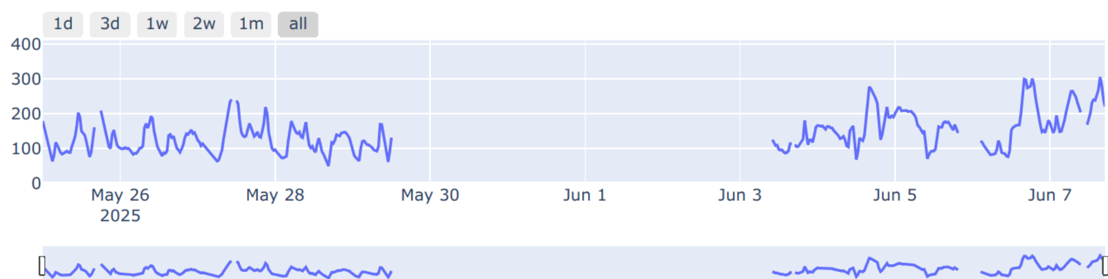


FIGURE 3.8: Time series of glucose level and skin temperature for train dataset of patient 567.

there is an interval greater than one week with missing values but since the values do not have a great variation, this week of missing values will be replaced with the last known missing value.

For the case of the test dataset of patient 552, all 3 variables have a long period of missing values as shown in the second figure. This is also the reason that the Test duration for this patient is greater than the others as previously shown in Table 3.19. If this was a train dataset, it would be worrisome to have models train with such a long period of missing values in the glucose level, in that case this dataframe would be split into two different dataframes removing the whole week of missing values. However this is a test dataframe, although it will lower the values of the metrics while predicting the glucose level of this patient, it will not have any impact on the performance of the models so the split will not be made and the missing values are filled with the previous known value.

The other patients' missing values had similar behavior as the patient 567 so all missing values are filled with the `ffill()` function.



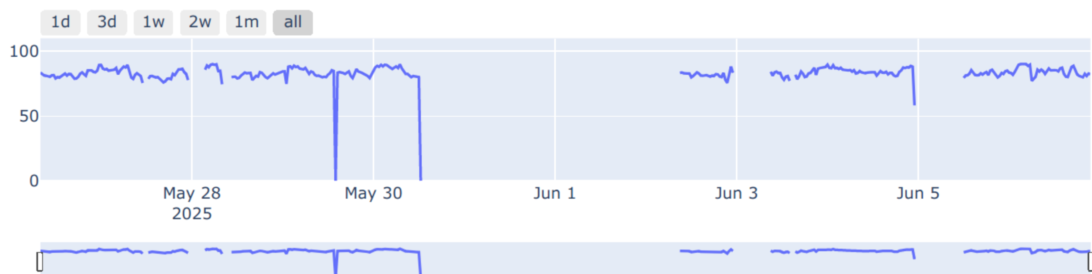


FIGURE 3.9: Time series of glucose level and skin temperature for test dataset of patient 552.

Chapter 4

Modeling

After the preprocessing of the datasets the data needs to be transformed into a format that can be used by the deep learning models, and to do so, a few rows containing all information of some hours will be gathered as X data to predict the glucose value of the next timestamp, the Y data. After the transformation, deep learning models will be tested to check if everything runs as expected and finally build more complex models.

4.1 Data Transformation

After preprocessing, the data is in the form of two dictionaries of dataframes, one for the training folder, df_all_tr and one for the testing folder, df_all_te. If the models need to be tested without some columns, a bin map is made as shown next.

Then, a new pair of dictionaries, df_select_tr and df_select_te, is created by making a copy of df_all dictionaries selecting the wanted columns using the bin_map.

Deep learning models are sensitive to the scale of the input data and so it is good practice to re-scale the data to the range -1 to +1 or 0 to 1 [13].

One of the possible scalers would be the `StandardScaler()`[27]. Equation 4.1 is the formula for the standard scaler where μ is the mean of the data and σ the standard deviation.

$$x_{scaled} = \frac{x - \mu}{\sigma} \tag{4.1}$$

This turns the data into values with mean = 0 and standard deviation = 1. This results in na output that has values outside the range -1 to +1.

The other option is the `MinMaxScaler()`[27].

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.2}$$

As shown in equation 4.2 this scaler subtracts the minimum value and this value is divided by the range ($x_max - x_min$) of the data. This scaler preserves the shape of the original distribution of data and does not reduce the importance of outliers [28].

After the selection of columns, the `MinMaxScaler()` function will be applied to all dataframes that will turn all the values in each column into the range of 1 (max value in that column) and 0 (min value).

The format that will be used as input for the models will be, the "*X_data*", an array with all values for rows within a *window_size* and the "*y_data*" that is the glucose value of the *window_size* + 1 row. The value of this window size will be set to 48, since each row has a time interval of 30 minutes, this means that the models will use values within 24 hours to predict the glucose value of the next 30 minutes. This transformation, the `df_to_X_y` function, is displayed on Figure 4.1 and this function is used on both training and test dataframes. Using the `.shape` function for the "*X_data_tr*", for the training data, it returns (2137, 48, 17), 2137 arrays containing 48 arrays of 17 values while "*y_data_tr*" returns (2137,), a single array with 2137 values.

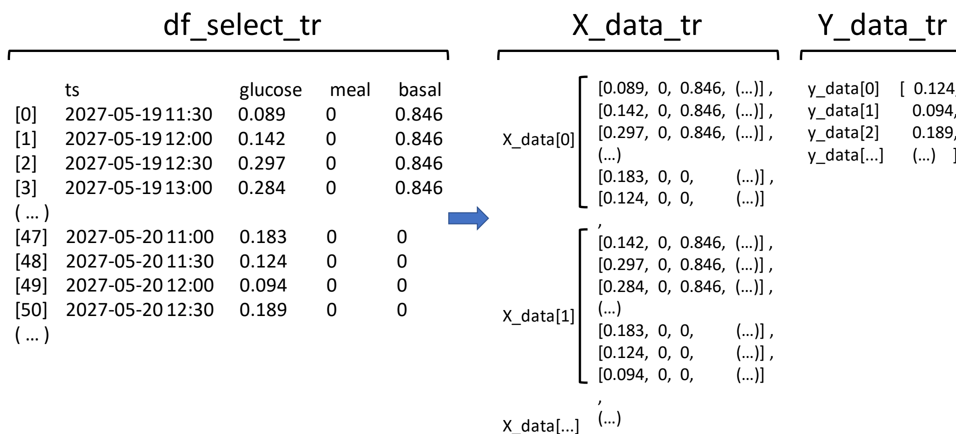


FIGURE 4.1: Data transformation from training dataframe to arrays for patient 540.

Then, a dictionary *df_final*, as shown in Figure 4.2, containing all this data is made that has 8 keys for each patient:

- "*X_tr*" - X Training data containing the first 1700 arrays of "*X_data_tr*".
- "*y_tr*" - y Training data containing the first 1700 values of "*y_data_tr*".

- *"X_val"* - X Validating data containing the remaining arrays of *"X_data_tr"*.
- *"y_val"* - X Validating data containing the remaining values of *"y_data_tr"*.
- *"X_te"* - X Testing data containing all arrays of *"X_data_te"*.
- *"y_te"* - y Testing data containing all values of *"y_data_te"*.
- *"df_tr"* - Copy of the training dataset
- *"df_te"* - Copy of the testing dataset

key value

'540' :

key	value
'X_tr' :	['X_data_tr'][1700:]
'y_tr' :	['y_data_tr'][1700:]
'X_val' :	['X_data_tr'][:1700]
'y_val' :	['y_data_tr'][:1700]
'X_te' :	['X_data_te']
'y_te' :	['y_data_te']
'df_tr' :	df_select_tr['540_training']
'df_te' :	df_select_te['540_testing']

FIGURE 4.2: df_final dictionary for patient 540.

The initial train data is split into train and validation data because a separate test set is needed in order to evaluate the model on unseen data. The model is trained using *"X_tr"* and *"y_tr"*, then validated on *"X_val"* and *"y_val"* and then tested on *"X_te"* and *"y_te"*. Most of the test data had between 400 and 500 rows and most of the train data had between 2100 and 2200. Splitting the train data by the 1700th row will result in a validation set with almost the same size as the test set.

The data is now ready to be used in the deep learning models.

4.2 Model Functions

The model compile function, `mod_compile`, is used to configure the learning parameters of the models and train the models. It has 4 required arguments and 5 arguments that have a default value when not used, these arguments are:

- model_name – String – Contains the name of the model currently running.

- df_data – Dictionary – The dictionary with all needed data, *df_final*.
- df_metric – Dataframe – Data frame where some output information will be saved.
- model – Keras Model – The deep learning model used.
- l_r – Float – Model's Learning Rate – Default: 0.001.
- epo – Int – Epochs, the number of times each model will train with a dataset – Default: 50.
- min_d – Float – Min-Delta, the minimum variation of the metric – Default: 0.0002.
- pat – Int – Patience, max number of epochs without improvement on the metric – Default: 4.
- metric – String – Metric used for EarlyStopping – Default: 'loss'.

The first step on model training is to configure the model, this is done using [29]:

```
model.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=l_r),
            metrics = [RootMeanSquaredError()])
```

This step will select the metric used for the loss function (*MSE*), the optimizer used (*Adam*) [11], the optimizer learning rate (*l_r*) and the list of metrics to be evaluated by the model during training (*RMSE*).

A loss function, also known as cost function, is a function that saves the differences of a predicted value and its actual value. The goal of an optimization problem is to minimize the loss function with a certain learning rate as shown in Figure 2.5.

Then a early stopping configuration is created using [29]:

```
tf.keras.callbacks.EarlyStopping(monitor=metric, patience = pat,
            min_delta = min_d, restore_best_weights = True)
```

Using this configuration the model will stop training earlier if there is no improvement on the metric selected on the *monitor* argument. The goal is to minimize the loss, so, at the end of every epoch the model will check if the loss decreased at least the *min_delta* value and if there is no improvement within *patience* epochs, the training terminates. The loss of the model can rise so, if the model was stopped using this method the best weights for the model are restored.

This rise in the loss is due to an overfit to the training dataset. As the epochs go by, the model learns and its error decreases both on training and validation data but after a certain number of epochs the validation loss will rise and enter the overfitting zone of the Figure 4.3. While in the underfitting zone the model can still learn and improve but once it starts overfitting the algorithm cannot perform accurately against unseen data so the training needs to be stopped.

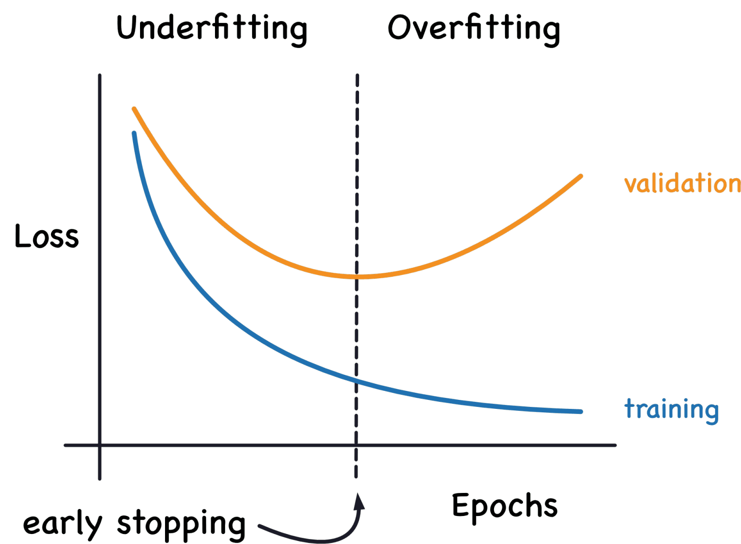


FIGURE 4.3: Early Stopping.

This effect is also shown on Figure 4.4 that contains the train and validation loss for two models with 200 and 1000 epochs. The values of *patience* and *min_delta* were selected as 4 and 0.0002 respectively after testing a few models and looking at the *loss* plots and then deciding what was the order of magnitude at which the *loss* stopped improving.

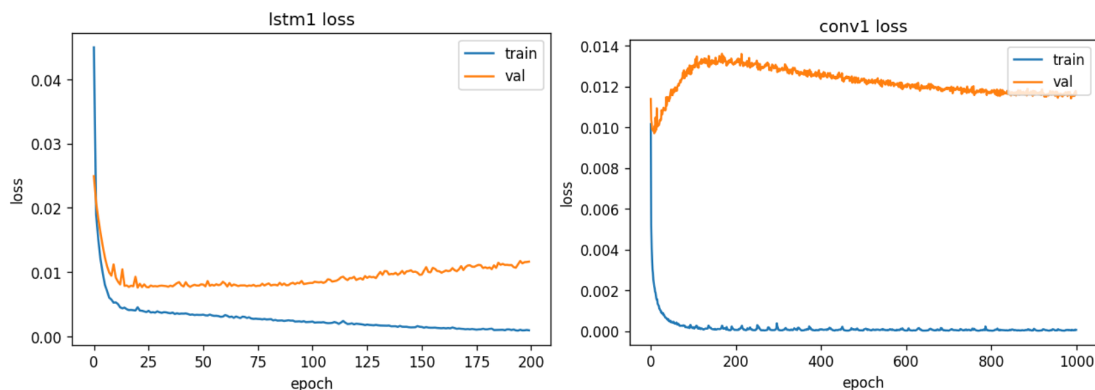


FIGURE 4.4: Train and Validation Loss for two different models with 200 and 1000 epochs.

Finally, the function [29]:

```
history = model.fit( df_data['X_tr'], df_data['y_tr'],
    validation_data = (df_data['X_val'], df_data['y_val']),
    epochs = epo, callbacks = [earlystop]
    )
```

that trains the model over a certain number of epochs. The output information about the train and validation loss and *rmse* in each epoch is saved in a new variable, *history*, so plots such as Figure 4.4 can be made. The `mod_compile` function returns the trained model and the variable *history*.

After the models are trained it is time to use the test data and predict these values using the `mod_pred` function which has 4 required arguments, the same ones of the `mod_compile` function. Using `model.predict(df_data['X_te']).flatten()`[29] a list of the predicted glucose values is obtained and this list is saved in a dataframe, *test_res*, alongside the actual values of the glucose (`df_data['y_te']`). The index of this dataframe is changed to the index of the test dataframe (`df_data['df_te'].index[w_s:]`), starting after "*w_s*" rows since these rows are used to start the predictions.

A sample of this dataframe is in Table 4.1 and with this dataset it is possible to obtain the *MSE* and *MAE* values as well as plot the time-series of both predicted and actual values.

TABLE 4.1: Sample of the *test_res*" dataframe for patient 540.

ts	test_pred	actual
2027-07-05 00:00:00	0.721	0.862
2027-07-05 00:30:00	0.865	0.982
2027-07-05 01:00:00	0.956	1.000
2027-07-05 01:30:00	0.854	0.984

4.3 1st Models

The first batch of models to be tested were simple versions of the *LSTM*, *GRU* and 1*DCNN*. For this test, the value of the `patience` will be 3 epochs instead of 4 and the `min_delta` will be 0.0005 instead of 0.0002. These values will trigger the earlystop earlier, reducing the train time, and since these are not the final values, it does not matter if the metrics are a bit worse than they should be.

The shape of the input layer of the first model, an LSTM model, is a 2D matrix with *"windowsize"* and variables used. This is the input shape used for all models. The following layer is a LSTM layer with 64 units (dimensionality of the output space) using a *TanH* activation function (default value). This is followed by a Dense layer that transforms the 64 outputs into 8 using a *relu* activation function. Since the value that needs to be predicted is a single value of the glucose, another Dense layer is used that transforms the 8 outputs into a single value.

```
1 def m_lstm1():
2     lstm1 = Sequential()
3     lstm1.add(InputLayer((w_s, n_col)))
4     lstm1.add(LSTM(64))
5     lstm1.add(Dense(8, 'relu'))
6     lstm1.add(Dense(1, 'linear'))
7     return lstm1
```

LISTING 1: First *LSTM* model.

The second and third models are very similar to the first one but using *GRU* and 1*D CNN* layer.

In the 1*D CNN* the *kernel_size* needs to be specified and the output shape of this layer depends on this value. Since the type of data used is a timeseries, the type of padding needs to be causal as shown in Section 2.4. Before the Dense layer a Flatten layer is added to turn the output into 1D shape.

```
1 def m_conv1():
2     conv1 = Sequential()
3     conv1.add(InputLayer((w_s, n_col)))
4     conv1.add(Conv1D(64, kernel_size=2, padding="causal", activation = 'relu'))
5     conv1.add(Flatten())
6     conv1.add(Dense(8, 'relu'))
7     conv1.add(Dense(1, 'linear'))
8     return conv1
```

LISTING 2: First 1*D CNN* model.

The last model is a slightly upgraded version of the first model. This model has an LSTM layer with 32 units and the information on the sequences will be passed on to a new LSTM layer with 64 units. After this step, the Dropout layer drops 20% of the units which helps prevent overfitting.

```
def m_lstm2():
    lstm2 = Sequential()
    lstm2.add(InputLayer((w_s, n_col)))
    lstm2.add(LSTM(32, return_sequences=True))
    lstm2.add(LSTM(64))
    lstm2.add(Dropout(0.20))
    lstm2.add(Dense(8, 'relu'))
    lstm2.add(Dense(1, 'linear'))
    return lstm2
```

LISTING 3: Second *LSTM* model.

A new dictionary, *model_dict*, is created as demonstrated in Figure 4.5 which has as key the "*model_name*" and each model contains 1 up to 6 keys. The only required key is the "*model*" that will contain the keras model, all other keys are the parameters used in the *mod_compile* function. When using `mod_compile` with `mod_compile(model_name, patient, df_final[patient], **model_dict[model_name])` it will check if the model_dict contains any key with the same name as the other parameters and if so, those values will be used as arguments instead of the default values. This dictionary is useful to pass to the function all models and their names and if some models need specific parameters they can be altered in this dictionary. Later on all that is needed is to create new models, add them to the dictionary, and with a for loop, all models can be tested easily.

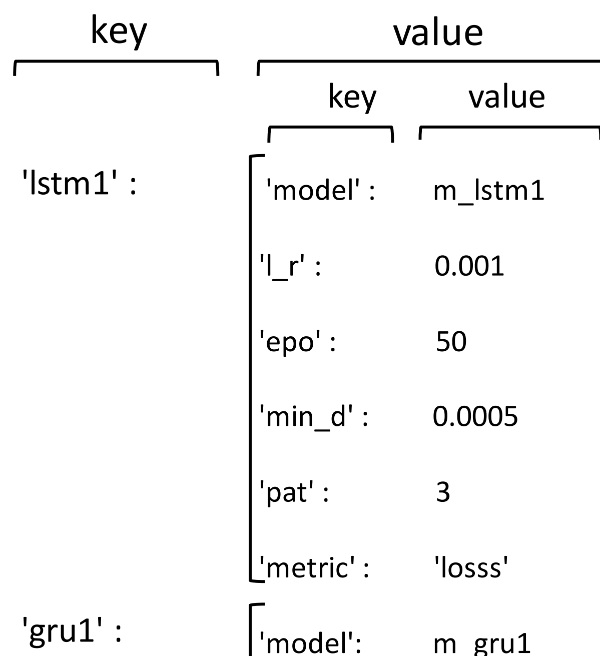


FIGURE 4.5: Model dictionary.

For each model, a new version with 48 units instead of 64 was tested to see if having the same units as the number of inputs would help in the performance of the models.

In the first test with these initial models, each model was trained with only one patient and made predictions on the corresponding test dataset. Using all "*test_res*" dataframes containing information on the actual and predicted values for each patient, metrics such as MSE, MAE and $RMSE$ were obtained.

For each model, two versions were tested, a personalized version in which a model is only trained with that patient train dataset and a generalized version in which each model is trained with all train datasets from all patients.

For all the initial models, more layers were added with different parameters. All models will be named as the following "model_type" + "order of complexity", this way it is easier to see if more complex models (with more layers), such as lstm4, outperform simpler models (with less layers) of the same type such as lstm1.

For the $LSTM$, 7 models are tested, 3 of these are $Bi-LSTM$ models whose structure is displayed in Tabel 4.2. $Bi-LSTM$ Are Bidirectional LSTM that process the information in two directions: Backwards (future to past) and Forward(past to future).

TABLE 4.2: Structure of $LSTM$ models.

lstm1	lstm2	lstm3	lstm4	bi-lstm1	bi-lstm2	bi-lstm3
LSTM(64)	LSTM(32)	LSTM(64)	LSTM(32)	BiLSTM(32)	BiLSTM(64)	BiLSTM(32)
Dense(8)	LSTM(64)	LSTM(128)	LSTM(64)	BiLSTM(64)	BiLSTM(128)	BiLSTM(64)
Dense(1)	Dense(8)	Dropout(0.2)	LSTM(128)	Dense(8)	Dropout(0.2)	BiLSTM(128)
	Dense(1)	Dense(8)	Dropout(0.2)	Dense(1)	Dense(8)	Dropout(0.2)
		Dense(1)	Dense(64)		Dense(1)	Dense(64)
			Dropout(0.2)			Dropout(0.2)
			Dense(8)			Dense(8)
			Dense(1)			Dense(1)

Higher numbered models have more $LSTM$ layers with a different number of units that pass the information into other $LSTM$ layers. To prevent overfit some of the information is removed using dropout layers.

Similar to the previous models, there are 6 models for GRU which 2 of these are $Bi-GRU$ as shown in Table .

The next batch of models are the $1D$ CNN models. As illustrated in Table 4.4 there are 5 different models.

Each convolutional layer has 2 parameters, the number of units and the kernel size and all layers have causal padding. The first 3 models have a kernel size of 2, the conv4 is

TABLE 4.3: Structure of *GRU* models.

gru1	gru2	gru3	gru4	bi-gru1	bi-gru2
GRU(64)	GRU(32)	GRU(64)	GRU(32)	BiGRU(64)	BiGRU(32)
Dense(8)	GRU(64)	GRU(128)	GRU(64)	BiGRU(128)	BiGRU(64)
Dense(1)	Dense(8)	Dropout(0.2)	GRU(128)	Dropout(0.2)	BiGRU(128)
	Dense(1)	Dense(8)	Dropout(0.2)	Dense(8)	Dropout(0.2)
		Dense(1)	Dense(64)	Dense(1)	Dense(64)
			Dropout(0.2)		Dropout(0.2)
			Dense(8)		Dense(8)
			Dense(1)		Dense(1)

TABLE 4.4: Structure of 1*D CNN* models.

conv1	conv2	conv3	conv4	conv5
Conv1D(64, 2)	Conv1D(64, 2)	Conv1D(64, 2)	Conv1D(64, 4)	Conv1D(32, 8)
Flatten()	Flatten()	MaxPooling()	MaxPooling()	MaxPooling()
Dense(8)	Dropout(0.2)	Conv1D(128, 2)	Conv1D(128, 4)	Conv1D(64, 4)
Dense(1)	Dense(32)	Flatten()	Flatten()	Pooling()
	Dense(8)	Dense(64)	Dropout(0.2)	Conv1D(128, 2)
	Dense(1)	Dense(1)	Dense(64)	Flatten()
			Dense(8)	Dropout(0.2)
			Dense(1)	Dense(64)
				Dense(8)
				Dense(1)

similar to conv3 but with higher kernel size. The last 1*D CNN* has 3 convolutional layers with increasing number of units and decreasing kernel size. To reduce the features after a convolution layer the MaxPooling layer condenses these features to 25% of their size selecting the most salient [8]. These features are then passed into a new convolution layer.

These models can be combined [8], the next batch mixed model of 1*D CNN* + *LSTM* and 1*D CNN* + *GRU* were tested as demonstrated in Table 4.5. These models are similar to some of the models on Tables 4.2 and 4.3 but have an additional first layer that is a convolutional layer with 64 units and kernel size 2.

TABLE 4.5: Structure of mixed models.

convlstm1	convlstm2	convlstm3	convgru1	convgru2
Conv1D(64, 2)	Conv1D(64, 2)	Conv1D(64, 2)	Conv1D(64, 2)	Conv1D(64, 2)
LSTM(64)	LSTM(64)	LSTM(32)	GRU(64)	GRU(32)
Dense(8)	LSTM(128)	LSTM(64)	Dense(8)	GRU(64)
Dense(1)	Dropout(0.2)	LSTM(128)	Dense(1)	Dense(8)
	Dense(8)	Dropout(0.2)		Dense(1)
	Dense(1)	Dense(64)		
		Dropout(0.2)		
		Dense(8)		
		Dense(1)		

The last models tested are the Temporal Convolutional Networks - TCN. These 3 models have several dilation values as shown in Figure 2.19 and as presented in Table 4.6 all TCN layers have 64 units but different kernel sizes. The dilation values are: 1, 2, 4, 8, 16, 32 and 64.

TABLE 4.6: Structure of TCN models.

tcn1	tcn2	tcn3
TCN(64, 2)	TCN(64, 3)	TCN(64, 6)
Dense(1)	Dense(1)	Dense(1)
Dropout(0.2)	Dropout(0.2)	Dropout(0.2)
Dense(1)	Dense(1)	Dense(1)

Chapter 5

Results

In this section the results of the models will be displayed and compared. The first comparison is using the initial models and compares the Personalized version of the models against the Generalized version. In the Personalized version each model is trained with one patient and tested with the same patient while the generalized version uses all patients datasets.

Then after selecting the best version, new and more complex models are trained with different amounts of variables. The first case will train with all 17 variables of the dataset while the second case will train with only a selection of variables. These values are also compared in terms of training time and metrics.

5.1 Personalized vs Generalized models

The first version tested was the personalized version. The Table 5.1 shows, for each model, the total time training for all 12 patients and the mean of the metrics for this version. Both GRU and $1D\ CNN$ models with 64 units outperformed their versions with 48 units, while in the $LSTM$ model, the best version is the one with 48 units. All 64 units versions are slightly slower but this is to be expected as they have more neurons.

In order to avoid overfit to only one patient and to train with more data, in the second version of these models, the generalized version, each model was trained with all patients and only then predictions were made.

The mean of the metrics for the predictions of each patient with the models trained with all patients and the total time of training is shown in Table 5.2.

TABLE 5.1: Total time in seconds spent on models, mean MSE, mean MAE and mean RMSE sorted by lowest RMSE - Personalized Version.

model	total_time	mean_mse	mean_mae	mean_rmse
gru1	199	0.0058	0.0543	0.075
gru1_48	182	0.0061	0.0568	0.0771
lstm1_48	232	0.0064	0.0568	0.0785
lstm1	252	0.0072	0.0614	0.0831
lstm2	466	0.0093	0.0676	0.089
conv1	46	0.0184	0.1007	0.1292
conv1_48	41	0.0252	0.1114	0.1416

TABLE 5.2: Total time spent on models, mean MSE, mean MAE and mean RMSE sorted by lowest RMSE - Generalized Version.

model	total_time	mean_mse	mean_mae	mean_rmse
gru1	117	0.0039	0.0434	0.0621
lstm1	119	0.0039	0.0441	0.0623
gru1_48	107	0.0041	0.0442	0.0636
lstm2	230	0.0044	0.0464	0.0657
lstm1_48	108	0.0049	0.0506	0.0695
conv1	33	0.0091	0.0738	0.0943
conv1_48	32	0.0123	0.0865	0.1084

With the generalized version all models with 64 units had better performance (lower metrics) and so, in the next models, layers with multiples of 64 units will be used.

The comparison between the personalized and the generalized versions is presented in Table 5.3. These values were obtained by subtracting the values of Table 5.2 to Table 5.1.

All times are positive, which means that training with each patient one at a time took more time than training with all patients sequentially. This happens because as the models are trained with more patients, with each patient it will not learn as much as the first patients and so the *loss* value improves less and less and the `EarlyStop` terminates the training with that patient and proceeds to the next dataset. All differences in the metrics are positive, indicating that the generalized version outperforms personalized versions. Using this information, all the new models will only be trained with all patients.

TABLE 5.3: Differences in time, mean MSE, mean MAE and mean RMSE sorted by Model name: Personalized − Generalized.

model	time_dif	mean_mse_dif	mean_mae_dif	mean_rmse_dif
lstm1	133	0.0033	0.0173	0.0208
lstm1_48	124	0.0015	0.0062	0.0090
lstm2	236	0.0049	0.0212	0.0233
conv1	13	0.0093	0.0269	0.0349
conv1_48	9	0.0129	0.0249	0.0332
gru1	82	0.0019	0.0109	0.0129
gru1_48	75	0.0020	0.0126	0.0135

5.2 All vs Selected variables

The new models were tested with all 17 variables of the dataset and these results are in Table 5.4.

TABLE 5.4: Time and error statistics sorted by RMSE - Generalized version with All variables.

model_name	time	mean_mse	mean_mae	mean_rmse
lstm1	192	0.0039	0.0440	0.0622
gru4	734	0.0040	0.0449	0.0632
gru3	388	0.0041	0.0450	0.0633
gru1	217	0.0041	0.0453	0.0639
lstm4	782	0.0042	0.0448	0.0643
bigru1	648	0.0043	0.0464	0.0650
convlstm1	259	0.0044	0.0462	0.0655
convlstm3	987	0.0044	0.0472	0.0664
bilstm2	944	0.0045	0.0482	0.0671
bilstm1	617	0.0046	0.0484	0.0675
bilstm3	1506	0.0046	0.0483	0.0677
convlstm2	696	0.0048	0.0502	0.0690
gru2	306	0.0049	0.0503	0.0695
convgru1	296	0.0049	0.0504	0.0698
lstm3	541	0.0050	0.0511	0.0701
convgru2	524	0.0050	0.0506	0.0706
lstm2	348	0.0053	0.0540	0.0727
conv1_48	63	0.0072	0.0630	0.0845
bigru2	1150	0.0074	0.0632	0.0859
conv2	80	0.0075	0.0646	0.0865
tcn1	522	0.0076	0.0660	0.0871
conv1	77	0.0088	0.0715	0.0933
tcn2	1097	0.0088	0.0706	0.0937
tcn3	1101	0.0094	0.0732	0.0965
conv3	116	0.0117	0.0818	0.1076
conv4	123	0.0120	0.0816	0.1086
conv5	119	0.0149	0.0933	0.1216

We can see that the worse models are the $1D$ CNN and TCN models and that in these cases more complex models not only have higher training time but also have lower performance. It is also shown that the Bidirectional versions of the $LSTM$ and GRU usually have lower performance and higher training time than the original versions.

Among the top 6 models (with mean_rmse greater or equal than 0.0650) 3 are GRU, 2 are $LSTM$ and one is $Bi - GRU$. It also shows that simpler models are not worse, as 2 of the top 6 models are the lstm1 and gru1.

In order to have lower training time a new training will be made but without some variables.

The removed variables are the following:

- Sleep - Since there is already information about the sleep using the basis_sleep, the self-reported version will be removed.
- Stressors and Illness - As shown in Table 3.2 there are very few events of this self-reported variable so it will be removed.
- Heart Rate, Air Temperature and Steps - These variables will be removed since they are not available for half of the patients.
- Finger Stick - This variable is removed since it either contains 0 or the glucose value at that time stamp so it does not add any relevant new information.

The results for all the models trained with the selected variables are displayed in Table 5.5.

Once again the worse models are the $1D$ CNN and TCN. This time, although the order changed, the top 6 best models are almost the same and there are more models with mean_rmse lower than 0.0650. Since many of the removed variables had mostly 0s, these variables might have lowered the performance of the models.

Table 5.6 contains the comparison of the models that were in the top 6 models in either Table 5.4 or 5.5. These values are obtained by subtracting the version with the selected variables from the version with all variables.

All time differences are positive, that means that all models with the selected variables had a faster training time. Although they were faster, it is noticeable that differences of the metrics for models lstm1 and gru4 are negative which implies that these models had a better performance while training with all variables.

TABLE 5.5: Time and error statistics sorted by RMSE - Generalized version with Selected variables.

model_name	time	mean_mse	mean_mae	mean_rmse
lstm3	452	0.0039	0.0433	0.0618
bigru1	591	0.0039	0.0428	0.0618
gru3	377	0.0039	0.0426	0.0618
gru1	173	0.0039	0.0429	0.0620
gru2	269	0.0041	0.0450	0.0635
lstm4	651	0.0041	0.0450	0.0636
convlstm3	699	0.0042	0.0458	0.0645
bilstm2	761	0.0042	0.0468	0.0646
convlstm1	134	0.0042	0.0449	0.0648
lstm1	167	0.0042	0.0449	0.0648
lstm2	280	0.0043	0.0458	0.0650
gru4	613	0.0043	0.0471	0.0654
bigru2	1028	0.0044	0.0463	0.0657
convgru1	227	0.0045	0.0479	0.0668
convgru2	317	0.0045	0.0473	0.0669
convlstm2	518	0.0046	0.0490	0.0675
bilstm1	481	0.0053	0.0532	0.0720
bilstm3	1204	0.0054	0.0527	0.0727
tcn2	884	0.0078	0.0670	0.0880
tcn3	952	0.0087	0.0703	0.0930
tcn1	405	0.0088	0.0729	0.0933
conv1	60	0.0094	0.0729	0.0953
conv2	78	0.0114	0.0809	0.1046
conv3	108	0.0134	0.0890	0.1150
conv4	130	0.0138	0.0901	0.1171
conv5	115	0.0177	0.1011	0.1311

TABLE 5.6: Differences in time and error statistics sorted by Model name - Generalized version: All - Selected variables.

model_name	time	mean_mse	mean_mae	mean_rmse
lstm1	25	-0.0003	-0.0009	-0.0026
lstm3	89	0.0011	0.0078	0.0083
lstm4	131	0.0001	-0.0002	0.0007
gru1	44	0.0002	0.0024	0.0029
gru2	37	0.0008	0.0053	0.0060
gru3	11	0.0002	0.0024	0.0015
gru4	121	-0.0003	-0.0022	-0.0022
bigru1	57	0.0004	0.0036	0.0032

Looking at Tables 5.5 and 5.2 it can be seen that the top values of mean_rmse are around 0.0620 in both cases. Although the later models were trained using a patience of 4 and min_delta of 0.0002 for the earlystop values and the models in Table 5.2 used the value 3 and 0.0005 respectively, they had similar results.

So, as a last test the new models will be trained with patience = 3 and min_delta = 0.0005 reducing the train time. The models that will be tested with these parameters are the best model of each model type in Table 5.5. In the case *LSTM* both model 3 and 1 will be tested because both were the best model in Tables 5.4 and 5.5. For *GRU* the models trained are gru1 and gru3 since they have the same value of mean_rmse. So the models tested with the new parameters are the following:

- lstm1
- lstm3
- bilstm2
- gru1
- gru3
- bigru1
- conv1
- convlstm1
- convgru1

TABLE 5.7: Time and error statistics sorted by Model - Generalized version with Selected variables - Pat = 3, min_delta = 0.0005.

model_name	time	mean_mse	mean_mae	mean_rmse
lstm1	116	0.0038	0.0426	0.0612
convlstm1	93	0.0038	0.0426	0.0612
gru3	256	0.0039	0.0429	0.0619
bigru1	421	0.0040	0.0452	0.0629
bilstm2	506	0.0041	0.0452	0.0634
gru1	101	0.0041	0.0453	0.0635
lstm3	322	0.0041	0.0457	0.0638
convgru1	134	0.0044	0.0476	0.0657
conv1	30	0.0080	0.0674	0.0883
tcn2	534	0.0102	0.0765	0.1005

With the new parameters the train time was reduced and some models like lstm1, convlstm1 had better performance than with the previous parameters when comparing Tables 5.7 and 5.5. There are also models such as conv1 and tcn2 that performed worse, maybe these models need more epochs for the training. The results for the predictions of each patient individually for for these models can be found on the Tables A.1 up to A.10

Usually the best overall best performing patient is patient 552, this is the patient with almost 40% missing values in the glucose level. Figure 5.1 shows the comparison of the predictions (black line) and the actual glucose values (blue line) for this patient using the model lstm1.

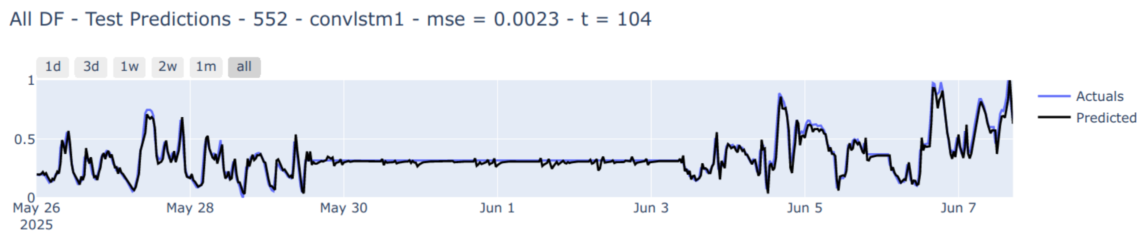


FIGURE 5.1: Plot of predictions for patient 552 model lstm1.

Most models seem to do a fine job predicting the constant values that were filled with the forward fill function so the results for this patient might be a bit biased positively. Due to this bias, the plots presented in Figures A.1 up to A.7 are the predictions for the patient 570 that is the second best performing patient overall using the models of Table 5.7. These figures also contain the plots of the predictions for patient 591 that is usually the worst performing patient.

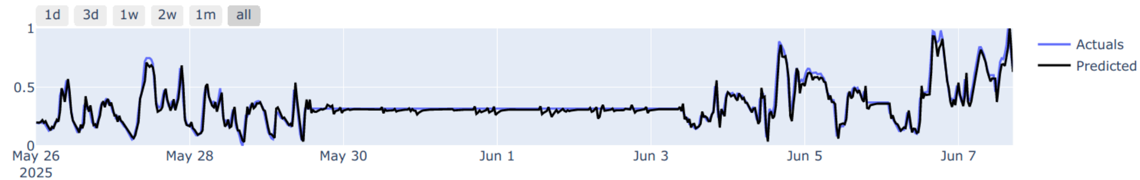
Each time the models are tested, even with the same parameters, sometimes there are slight fluctuations on the training time ($\pm$ 10s) and on the metrics (mse $\pm$ 0.002). To avoid these slight fluctuations on the final models, they will be trained 2 more times. Table 5.8 contains the results for the mean of the 3 outputs for the best models, convlstm1, lstm1, gru1 and gru3. The information for other models is not shown due to poor performance or higher training time.

TABLE 5.8: Time and error statistics sorted by Model - Generalized version with Selected variables - Pat = 3, min_delta = 0.0005 - Mean of 3 different trains.

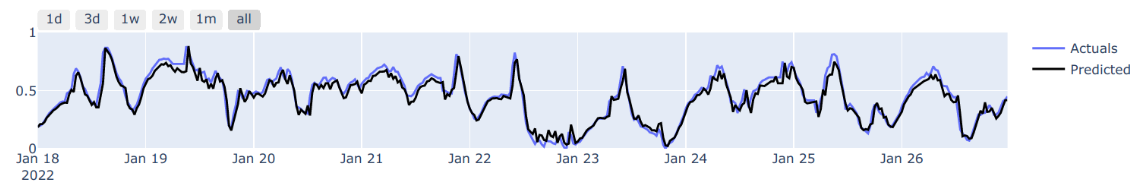
model_name	time	mean_mse	mean_mae	mean_rmse
convlstm1	97	0.0039	0.0431	0.0617
lstm1	118	0.0039	0.0431	0.0617
gru1	109	0.0040	0.0444	0.0630
gru3	264	0.0041	0.0449	0.0635

All models had similar performance, but gru3 training time was significantly higher. Among the three other models, the convlstm1 was the fastest and the one with lower mean_rmse. The plots for the 2 best and 2 worse performing patients for this model are shown in Figures 5.2a and 5.2b respectively. The same plots for the two other models, lstm1 and gru1, are shown in Figures A.8 and A.9

All DF - Test Predictions - 552 - convlstm1 - mse = 0.0023 - t = 104

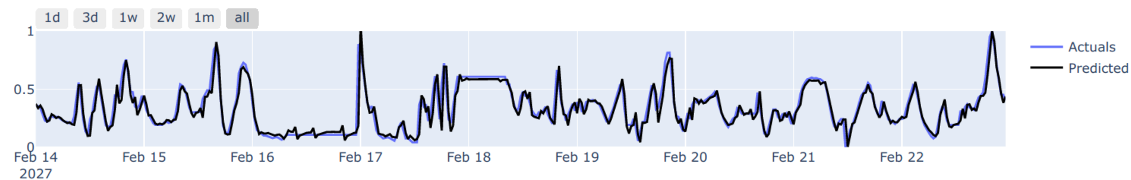


All DF - Test Predictions - 570 - convlstm1 - mse = 0.00301 - t = 104

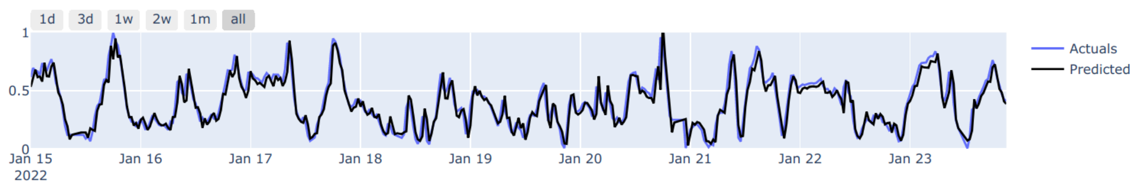


(A) Plot of predictions for patient 552 and 570 using model convlstm1.

All DF - Test Predictions - 567 - convlstm1 - mse = 0.00523 - t = 104



All DF - Test Predictions - 591 - convlstm1 - mse = 0.0058 - t = 104



(B) Plot of predictions for patient 567 and 591 using model convlstm1.

FIGURE 5.2: Plot of predictions for top 2 and bottom 2 performing patients using model convlstm1.

5.3 Comparing Other Methods

In the second Blood Glucose Level Prediction (BGLP) Challenge, concluded on August 30, 2020, several AI researchers proposed different approaches to predict glucose levels, using the OhioT1DM dataset [6].

In this section, two of these approaches will be analysed and discussed. Additionally, other two approaches that use the OhioT1DM dataset for glucose value prediction, will also be examined.

In this challenge all predictions were made using a time interval of the data of 30 and 60 minutes.

Approaches such as Bhimireddy et al. [30] and Freiburghaus et al. [31], used Deep Learning [32].

Bhimireddy et al. used $LSTM$, $Bi-LSTM$, $1D\ CNN-LSTM$ and TCN models. The authors however, did not use all the available variables. The variables used were the: glucose values, finger stick, basal, bolus, meal, sleep, exercise, gsr, skin temperature, basis sleep, and acceleration. Despite removing several variables, they considered as a variable every column in the variables above. This resulted in having columns such as $basis_sleep_type$ or $exercise_type$ that contained no information at all. Thus, the final dataframe has many columns with only 0s, columns that should have been removed. Although some features' columns did not contain data, e.g. $basis_sleep_type$ or $exercise_type$, the authors preserved the empty columns in the dataset. The final dataframe, given this methodology, contains columns filled with values equal to zero, that should not exist. The variable $temp_basal$ was not combined with the $basal$ leading to wrong basal values. The only preprocessing made was a forward fill of all values after the 30 / 60 minute re-sample. The authors make predictions on the next glucose value with only the information of the previous 30 / 60 minutes. In this work, the best preforming model was the $Bi-LSTM$ with a mean $RSME$ of 21.8 when trying to predict the glucose value of the next 30 minutes.

Freiburghaus et al., contrary to the previous approach, filtered empty columns, but used less features. The only variables used in this work were the meal, bolus, basal and glucose level records. The column bwz_carb_input, contains redundant meal information in the 2018 patient datasets and is empty in the 2020 patient datasets. Additionally, the authors did not consider the type of bolus for each time stamp nor the variable $temp_basal$. In this work the data was re-sampled to 1 second. Then, a forward fill was applied to all variables. Finally, the data was re-sampled to 5 minute intervals. The glucose value predictions are based on the previous two hours of data using a $1D\ CNN$ model. The 30 minute glucose level prediction of this work had an $RMSE$ mean of 17.45. This result is lower than the one obtained in the work by Bhimireddy et al. [30].

Beside the BGLP Challenge, other authors have used the OhioT1DM dataset to do a deep learning analysis.

The the work by T. Zhu et al. [33] used TCN to predict glucose values. In this work, the authors only used the Ohio datasets from 2020. The only variables used were the glucose value, meal and bolus. The bolus type was not considered, thus the considered bolus data may lead to incorrect conclusions. Besides other removed variables, the authors removed the variables *basal* and $temp_basal$. Using only the bolus variable the insulin intake by each patient is incomplete. The dataset set used for training was a combination of 50% of the current subject, and the other five patients contribute to the other half of the training data with 10% each [33]. They use a time interval of data of 5 minutes and make predictions of the glucose levels on the next 30 minutes using the previous 30 minutes of data. The $RMSE$ value for this study was 21.72, almost the same as the work by Bhimireddy et al. [30].

The work by J. Daniels et al. [34] consists on the use of a $1D\ CNN - LSTM$ model to make predictions of the glucose level in the next 30 / 60 minutes. Glucose levels, bolus, meals and exercise were the considered variables. Once again the variables *basal* and $temp_basal$, as well as other variables, were not used. To deal with missing values the authors use a linear interpolation. The glucose values are scaled down by a factor of 120, the insulin bolus is scaled by 100 and meal intake values are scaled by 200. The bolus type is not used. To predict the next glucose value the method uses the previous two hours of data [34]. The result for the 30 minute predictions was a $RSME$ of 19.79.

M. Rabby et al. [35] used yet a different method with the same dataset. Instead of selecting a fixed number of variables *a priori*, they tested their $LSTM$ model with only the glucose value and obtained the corresponding $RMSE$. Then, they added one variable and tested if the $RMSE$ was lower and if so, the variable is kept and more variables are added. If, when adding a variable, the $RMSE$ rises, that variable is removed. The final selected variables are the glucose level, meal, bolus and step count. They used the information of the previous two hours to predict the glucose value of the next 30 minutes of the six 2018 patients. This resulted in a $RMSE$ of 20.07

After reviewing the considered approaches, it is interesting to see that some authors do not consider certain variables in the data important. Most of the analysed works do not include the basal insulin rate, and none of the works include the temporary basal rate

feature. On top of this, most of the approaches do not consider the bolus type. Neglecting these features, may lead to a wrong input of the actual insulin taken by the patient. Additionally, most of the authors only consider one type of neural network. Two of the reviewed analysis used 30 minutes of data to predict the next 30 minutes, while the other three used data from the previous two hours. Finally, most of the analysis did not use any type of data scaling, which is considered to be a bad practice [13].

Chapter 6

Conclusions and Future Work

In this work we used data from the OhioT1DM dataset [2], which has 12 patients in 2018 and 2020 in order to predict the glucose value of a patient in the next 30 minutes using information of 17 variables in the previous 24 hours. Comparing the 2 versions of the initial models, the personalized and generalized versions, it is noticeable that training the models with all patients sequentially takes less time than training the models 12 times individually. On top of being faster, the generalized version also produced better results in terms of the predictions.

Using the generalized version, the train time and the metrics of two new versions were compared. In the first version the models were trained using all available variables, while in the second version, the variables sleep, stressors, illness, heart rate, air temperature, steps and finger stick were removed. As expected, the version trained with a selection of variables was faster than the other version, and most of the models had better performance. This indicates that for most models the removed variables had no impact on the predictions.

The parameters of the `EarlyStop` were also changed in order to reduce the number of times the dataset of each patient is trained that reduced the training time of all models. Although some models performed worse, some had better performance indicating that depending on the type of the model it might need to train each dataset more times than other models.

After all these tests, the best performing models were the simpler versions (with the lowest amount of layers) of the models $LSTM$, GRU and $ConvLSTM$ having not only the lowest metrics but also lowest training time. This shows that for this dataset, maybe due to the dataset size or number of variables, simple deep learning models outperform

more complex models. The best model of all was the combined model of $1DCNN$ and $LSTM$, having the best values of the metrics and having the lowest training time of the best performing models.

As future work, several cases can be tested. One of those is re-scaling the dataset to a different time intervals such as 5 or 15 minutes where the training time will be slower but the values of the metrics can change.

Removing more variables or trying different combinations of variables can also be tested and compare the results to see if there is a better combination.

The window size of 24 hours for the X_data can also be tested with new values like using only 12 hours of values or even using bigger window sizes such as 48 hours and see if the metrics improve.

New values for the compiler can also be tested such as reducing or increasing the learning rate or reducing the min_delta to search the lowest training time with the best metrics.

New versions of the simpler models can also be tested with higher or lower number of neurons of the deep learning layers.

Finally more information about future patients can also be added in order to have even more training data for the models. Having data from more patients will reduce the overfiting to each patient.

Bibliography

[1] I. D. Federation. (2021) About diabetes. [Online]. Available: https://idf.org/aboutdiabetes/what-is-diabetes/facts-figures.html [Cited on page 1.]

[2] B. R. Marling C, "The ohiot1dm dataset for blood glucose level prediction: Update 2020," 2020. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7881904/ [Cited on pages 1, 3, 4, and 69.]

[3] I. D. Federation. (2022) What is diabetes. [Online]. Available: https://www.idf.org/aboutdiabetes/what-is-diabetes.html [Cited on page 2.]

[4] C. for Disease Control and Prevention. (2022) What is diabetes. [Online]. Available: https://www.cdc.gov/diabetes/basics/diabetes.html [Cited on page 2.]

[5] I. D. Federation. (2020) Type 2 diabetes. [Online]. Available: https://www.idf.org/aboutdiabetes/type-2-diabetes.html [Cited on page 2.]

[6] Blood glucose level prediction challenge. [Online]. Available: https://sites.google.com/view/kdhd-2018/bglp-challenge [Cited on pages 3 and 64.]

[7] N. I. of Diabetes, Digestive, and K. Diseases. (2017) Continuous glucose monitoring. [Online]. Available: https://www.niddk.nih.gov/health-information/diabetes/overview/managing-diabetes/continuous-glucose-monitoring [Cited on page 3.]

[8] J. Brownlee, *Deep Learning for Time Series Forecasting*. Independently Published, 2019. [Cited on pages 6, 9, 13, 17, and 54.]

[9] A. Pai. (2020) Cnn vs rnn vs ann analyzing 3 types of neural networks in deep learning. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/ [Cited on pages ix, 10, 11, 14, 17, and 18.]

[10] N. Kang. (2017) Multi-layer neural networks with sigmoid function. [Online]. Available: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f [Cited on pages ix, 10, and 11.]

[11] I. Gridin, *Time Series Forecasting using Deep Learning*. BPB Publications, 2022. [Cited on pages ix, 11, 12, 14, 17, 19, and 48.]

[12] P. S. Artificial neural networks – better understanding. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/artificial-neural-networks-better-understanding/ [Cited on pages ix, 12, and 13.]

[13] N. D. Lewis, *Deep Time Series Forecasting with Python*. CreateSpace Independent Publishing Platform, 2016. [Cited on pages 13, 45, and 67.]

[14] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994. [Cited on page 14.]

[15] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735 [Cited on page 15.]

[16] C. Olah. Understanding lstm networks. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ [Cited on pages 15 and 16.]

[17] M. Phi. (2018) Illustrated guide to lstm's and gru's: A step by step explanation. [Online]. Available: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 [Cited on pages ix, 15, and 16.]

[18] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014. [Online]. Available: https://arxiv.org/abs/1406.1078 [Cited on page 16.]

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, 1989. [Cited on page 17.]

[20] J. Jeong. (2019) The most intuitive and easiest guide for convolutional neural network. [Online]. Available: https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480 [Cited on pages ix and 18.]

[21] A. Singh. Demystifying the mathematics behind convolutional neural networks (cnns). [Online]. Available: https://www.analyticsvidhya.com/blog/2020/02/mathematics-behind-convolutional-neural-network/ [Cited on pages ix and 19.]

[22] Dilated and causal convolution. [Online]. Available: https://subscription.packtpub.com/book/data/9781789136364/4/ch04lvl1sec59/dilated-and-causal-convolution [Cited on pages ix and 20.]

[23] A. Ayodeji, Z. Wang, W. Wang, W. Qin, C. Yang, S. Xu, and X. Liu, "Causal augmented convnet: A temporal memory dilated convolution model for long-sequence time series prediction," *ISA Transactions*, vol. 123, pp. 200–217, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019057821002810 [Cited on page 20.]

[24] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016. [Online]. Available: https://arxiv.org/abs/1609.03499 [Cited on pages ix and 21.]

[25] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," 2018. [Online]. Available: https://arxiv.org/abs/1803.01271 [Cited on page 21.]

[26] Medtronic 630g user guide. [Online]. Available: https://www.medtronicdiabetes.com/download-library/minimed-630g-system [Cited on pages ix, 32, and 33.]

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011. [Cited on page 45.]

[28] F. Shahid, A. Zameer, and M. Muneeb, "Predictions for covid-19 with deep learning models of lstm, gru and bi-lstm," *Chaos, Solitons & Fractals*, vol. 140, p.

110212, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0960077920306081 [Cited on page 46.]

[29] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://keras.io [Cited on pages 48, 49, and 50.]

[30] A. R. Bhimireddy, P. Sinha, B. Oluwalade, J. W. Gichoya, and S. Purkayastha, "Blood glucose level prediction as time-series modeling using sequence-to-sequence neural networks," in *KDH@ECAI*, 2020. [Cited on pages 65 and 66.]

[31] J. Freiburghaus, A. Rizzotti-Kaddouri, and F. Albertetti, "A deep learning approach for blood glucose prediction and monitoring of type 1 diabetes patients," in *KDH@ECAI*, 2020. [Cited on page 65.]

[32] Blood glucose level prediction challenge results. [Online]. Available: http://smarthealth.cs.ohio.edu/bglp/bglp-results.html [Cited on page 65.]

[33] T. Zhu, K. Li, P. Herrero, and P. Georgiou, "Personalized blood glucose prediction for type 1 diabetes using evidential deep learning and meta-learning," *IEEE Transactions on Biomedical Engineering*, pp. 1–12, 2022. [Online]. Available: https://doi.org/10.1109/tbme.2022.3187703 [Cited on page 66.]

[34] J. Daniels, P. Herrero, and P. Georgiou, "A multitask learning approach to personalized blood glucose prediction," *IEEE Journal of Biomedical and Health Informatics*, vol. 26, no. 1, pp. 436–445, Jan. 2022. [Online]. Available: https://doi.org/10.1109/jbhi.2021.3100558 [Cited on page 66.]

[35] M. F. Rabby, Y. Tu, M. I. Hossen, I. Lee, A. S. Maida, and X. Hei, "Stacked LSTM based deep recurrent neural network with kalman smoothing for blood glucose prediction," *BMC Medical Informatics and Decision Making*, vol. 21, no. 1, Mar. 2021. [Online]. Available: https://doi.org/10.1186/s12911-021-01462-5 [Cited on page 66.]

Appendix A

Further results

A.1 Additional Tables

TABLE A.1: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model lstm1 - Train Time = 116s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0023	0.0304	0.0475
570	0.0026	0.0362	0.0512
559	0.0030	0.0369	0.0547
588	0.0035	0.0415	0.0590
544	0.0037	0.0416	0.0605
584	0.0037	0.0427	0.0610
596	0.0037	0.0429	0.0611
540	0.0039	0.0472	0.0622
563	0.0040	0.0445	0.0630
575	0.0046	0.0473	0.0675
567	0.0052	0.0454	0.0722
591	0.0056	0.0541	0.0749

TABLE A.2: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model lstm3 - Train Time = 322s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0026	0.0342	0.0505
559	0.0031	0.0388	0.0555
570	0.0033	0.0433	0.0575
588	0.0036	0.0440	0.0603
584	0.0038	0.0438	0.0613
544	0.0039	0.0437	0.0622
596	0.0041	0.0453	0.0643
540	0.0041	0.0502	0.0644
563	0.0044	0.0473	0.0662
575	0.0051	0.0525	0.0711
567	0.0053	0.0471	0.0729
591	0.0062	0.0579	0.0790

TABLE A.3: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model bilstm2 - Train Time = 506s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0023	0.0317	0.0475
559	0.0031	0.0385	0.0558
570	0.0033	0.0432	0.0573
588	0.0036	0.0439	0.0601
544	0.0037	0.0411	0.0611
596	0.0039	0.0450	0.0626
540	0.0040	0.0489	0.0630
584	0.0041	0.0466	0.0638
563	0.0048	0.0486	0.0691
575	0.0052	0.0531	0.0720
567	0.0053	0.0460	0.0728
591	0.0057	0.0556	0.0758

TABLE A.4: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model gru1 - Train Time = 101s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0025	0.0356	0.0502
559	0.0031	0.0383	0.0558
570	0.0033	0.0433	0.0575
588	0.0037	0.0443	0.0612
596	0.0038	0.0437	0.0614
544	0.0038	0.0422	0.0618
540	0.0040	0.0482	0.0629
584	0.0042	0.0473	0.0648
563	0.0044	0.0477	0.0665
575	0.0050	0.0502	0.0709
567	0.0052	0.0459	0.0718
591	0.0060	0.0565	0.0776

TABLE A.5: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model gru3 - Train Time = 256s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0024	0.0324	0.0490
570	0.0025	0.0365	0.0500
559	0.0027	0.0343	0.0524
588	0.0035	0.0423	0.0587
584	0.0037	0.0420	0.0609
596	0.0038	0.0434	0.0614
544	0.0039	0.0427	0.0622
540	0.0039	0.0470	0.0627
563	0.0042	0.0461	0.0649
575	0.0048	0.0477	0.0692
567	0.0057	0.0457	0.0755
591	0.0057	0.0546	0.0756

TABLE A.6: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model bigru1 - Train Time = 421s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0021	0.0310	0.0463
559	0.0030	0.0382	0.0548
570	0.0034	0.0448	0.0579
544	0.0035	0.0415	0.0594
588	0.0035	0.0439	0.0596
540	0.0037	0.0476	0.0607
584	0.0038	0.0451	0.0620
596	0.0042	0.0475	0.0649
563	0.0044	0.0474	0.0662
567	0.0052	0.0460	0.0723
591	0.0056	0.0545	0.0746
575	0.0058	0.0552	0.0761

TABLE A.7: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model convlstm1 - Train Time = 93s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0023	0.0304	0.0475
570	0.0026	0.0362	0.0512
559	0.0030	0.0369	0.0547
588	0.0035	0.0415	0.0590
544	0.0037	0.0416	0.0605
584	0.0037	0.0427	0.0610
596	0.0037	0.0429	0.0611
540	0.0039	0.0472	0.0622
563	0.0040	0.0445	0.0630
575	0.0046	0.0473	0.0675
567	0.0052	0.0454	0.0722
591	0.0056	0.0541	0.0749

TABLE A.8: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model convgru1 - Train Time = 134s.

patient	mean_mse	mean_mae	mean_rmse
552	0.0028	0.0395	0.0533
559	0.0034	0.0397	0.0581
588	0.0036	0.0439	0.0601
570	0.0037	0.0469	0.0607
540	0.0041	0.0482	0.0641
596	0.0043	0.0472	0.0653
544	0.0043	0.0492	0.0659
584	0.0044	0.0481	0.0665
563	0.0047	0.0488	0.0683
575	0.0052	0.0521	0.0720
567	0.0058	0.0501	0.0760
591	0.0061	0.0574	0.0782

TABLE A.9: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model conv1 - Train Time = 30s.

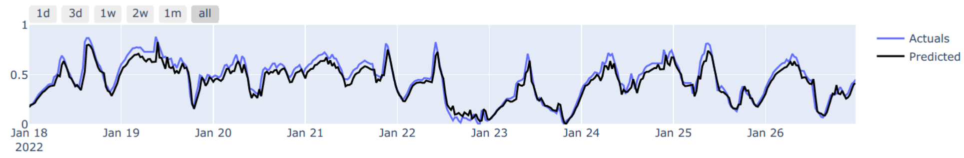
patient	mean_mse	mean_mae	mean_rmse
552	0.0049	0.0518	0.0703
559	0.0051	0.0541	0.0715
544	0.0063	0.0596	0.0796
540	0.0068	0.0644	0.0823
596	0.0068	0.0620	0.0823
591	0.0071	0.0624	0.0843
588	0.0075	0.0676	0.0863
567	0.0080	0.0650	0.0893
584	0.0084	0.0694	0.0919
575	0.0100	0.0736	0.1002
563	0.0117	0.0849	0.1084
570	0.0128	0.0941	0.1130

TABLE A.10: Time, mean MSE, mean MAE and mean RMSE sorted by RMSE - Generalized Verison with Selected variables - Model tcn2 - Train Time = 534s.

patient	mean_mse	mean_mae	mean_rmse
559	0.0062	0.0604	0.0788
552	0.0067	0.0554	0.0820
588	0.0086	0.0720	0.0927
540	0.0089	0.0726	0.0943
596	0.0094	0.0710	0.0968
591	0.0101	0.0750	0.1005
544	0.0101	0.0768	0.1007
575	0.0111	0.0794	0.1052
567	0.0125	0.0819	0.1119
570	0.0127	0.0939	0.1125
584	0.0129	0.0867	0.1134
563	0.0138	0.0934	0.1174

A.2 Additional Figures

All DF - Test Predictions - 570 - lstm3 - mse = 0.00428 - t = 354



All DF - Test Predictions - 591 - lstm3 - mse = 0.00813 - t = 354

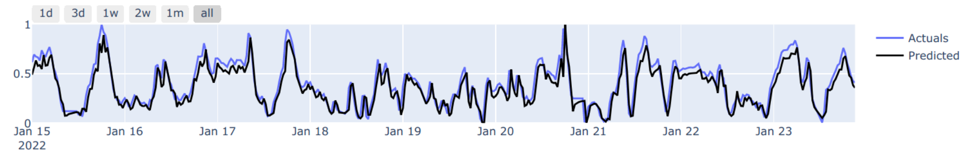
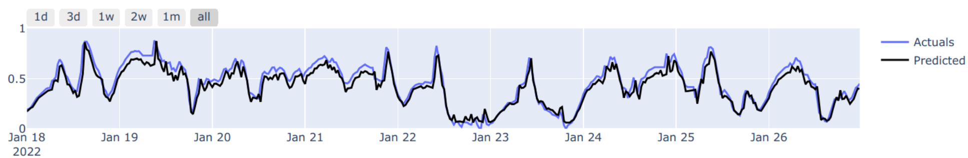


FIGURE A.1: Plot of predictions for patient 570 and 591 using model lstm3.

All DF - Test Predictions - 570 - bigru1 - mse = 0.00435 - t = 533



All DF - Test Predictions - 591 - bigru1 - mse = 0.00662 - t = 533

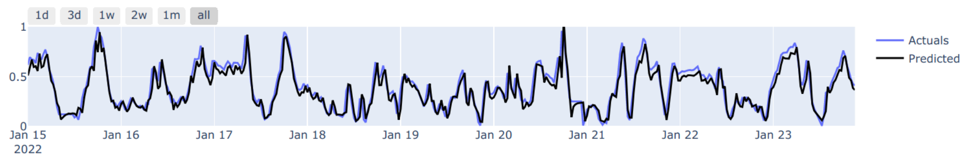
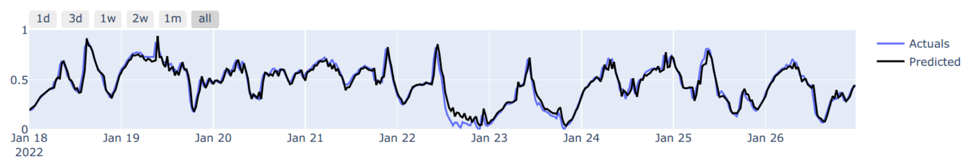


FIGURE A.2: Plot of predictions for patient 570 and 591 using model bigru1.

All DF - Test Predictions - 570 - gru3 - mse = 0.0028 - t = 310



All DF - Test Predictions - 591 - gru3 - mse = 0.00595 - t = 310

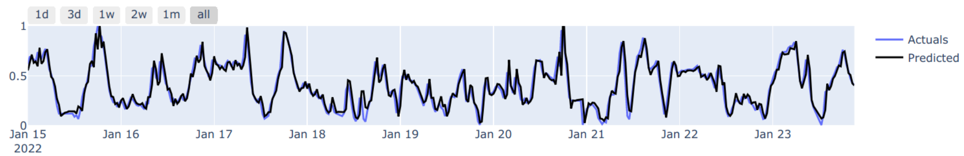
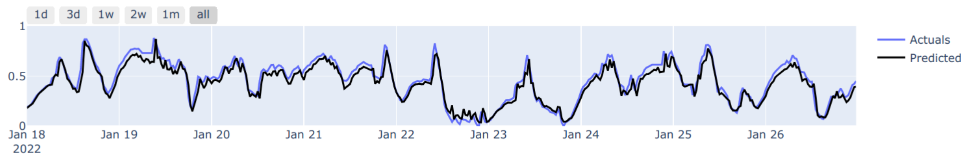


FIGURE A.3: Plot of predictions for patient 570 and 591 using model gru3.

All DF - Test Predictions - 570 - bilstm2 - mse = 0.00388 - t = 633



All DF - Test Predictions - 591 - bilstm2 - mse = 0.00639 - t = 633

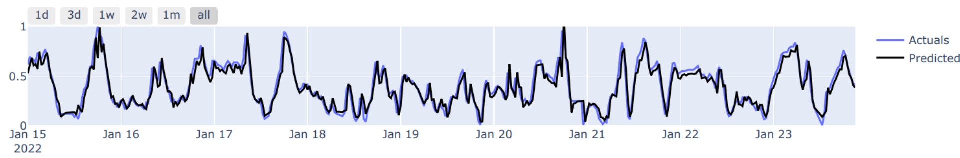


FIGURE A.4: Plot of predictions for patient 570 and 591 using model bilstm2.

All DF - Test Predictions - 570 - convgru1 - mse = 0.00289 - t = 171

All DF - Test Predictions - 591 - convgru1 - mse = 0.00576 - t = 171

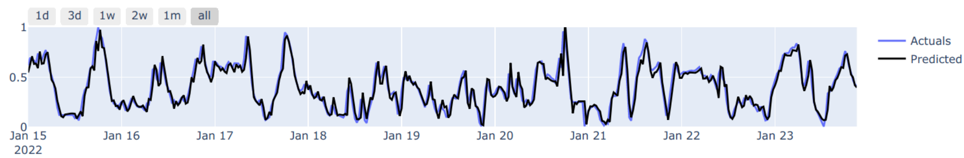
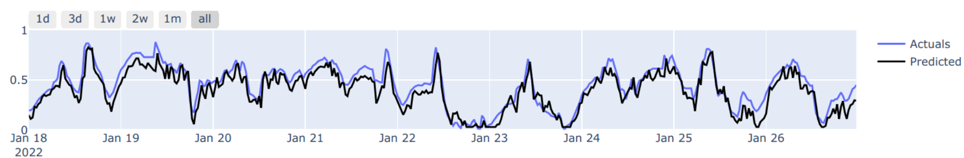


FIGURE A.5: Plot of predictions for patient 570 and 591 using model convgru1.

All DF - Test Predictions - 570 - conv1 - mse = 0.00856 - t = 31



All DF - Test Predictions - 591 - conv1 - mse = 0.00646 - t = 31

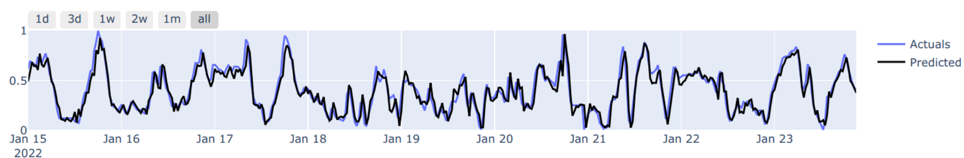


FIGURE A.6: Plot of predictions for patient 570 and 591 using model conv1.

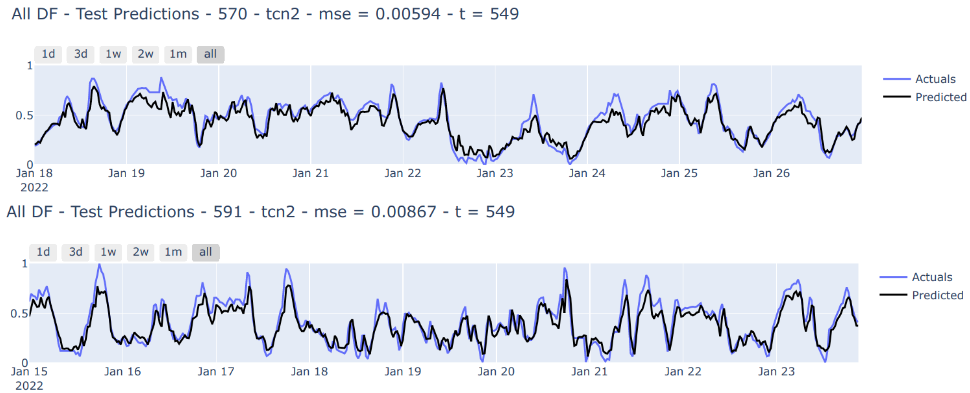
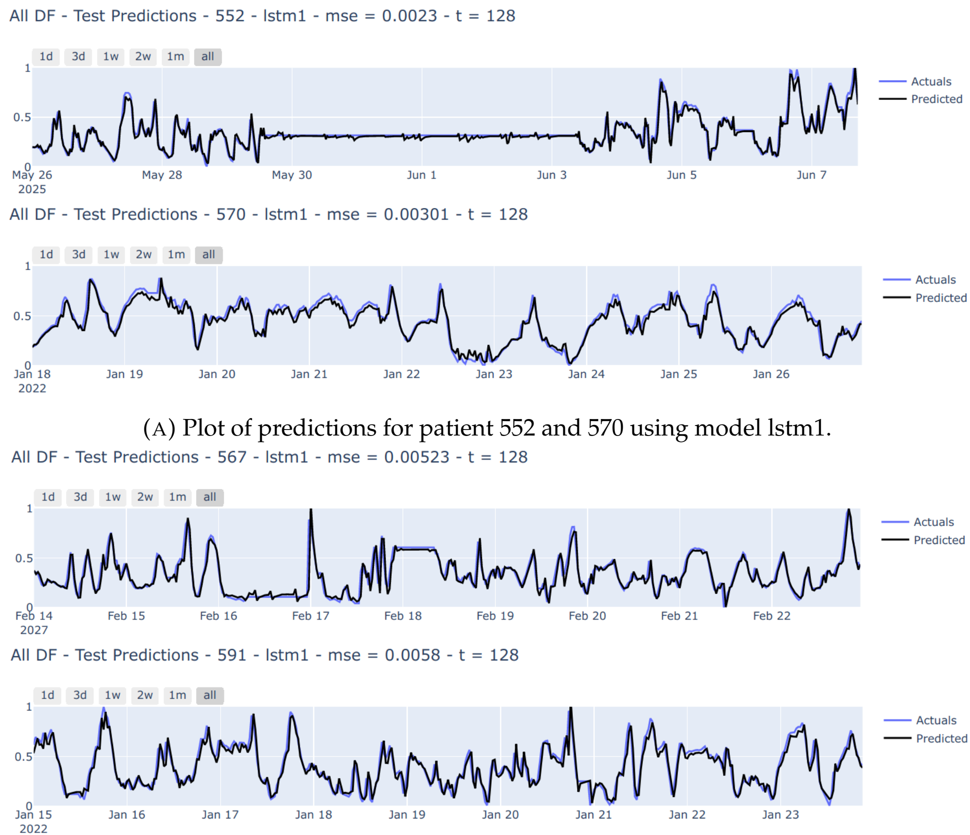


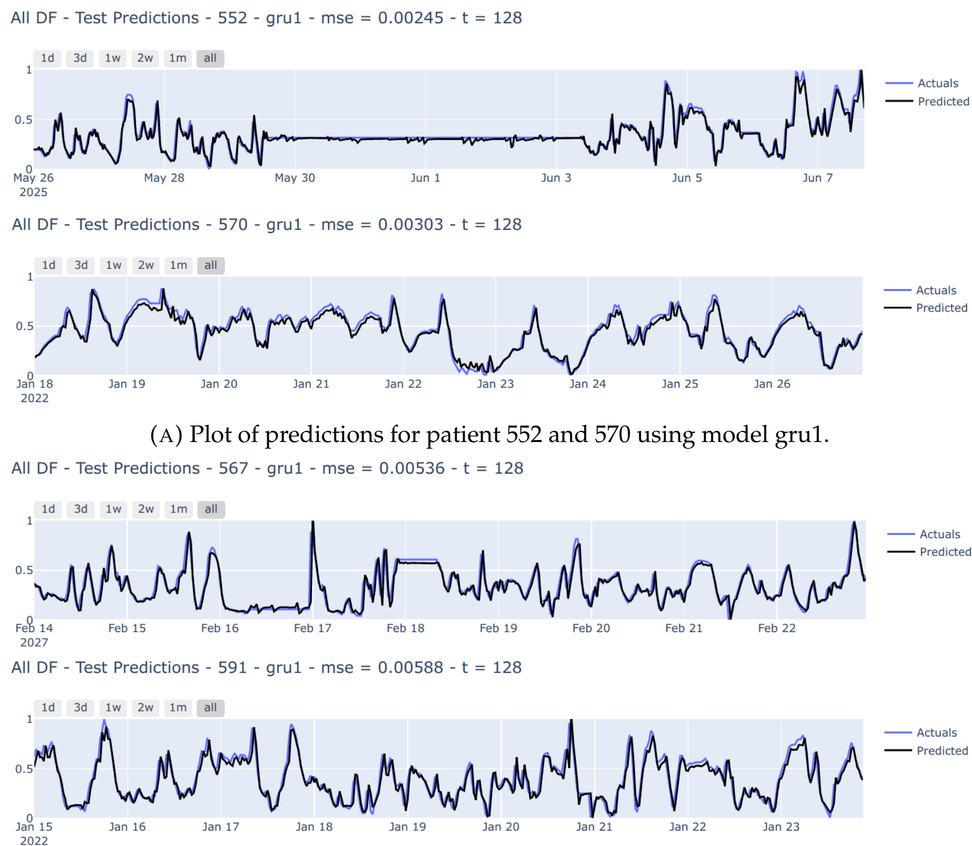
FIGURE A.7: Plot of predictions for patient 570 and 591 using model tcn2.



(A) Plot of predictions for patient 552 and 570 using model lstm1.

(B) Plot of predictions for patient 567 and 591 using model lstm1.

FIGURE A.8: Plot of predictions for top 2 and bottom 2 performing patients using model lstm1.



(A) Plot of predictions for patient 552 and 570 using model gru1.

(B) Plot of predictions for patient 567 and 591 using model gru1.

FIGURE A.9: Plot of predictions for top 2 and bottom 2 performing patients using model gru1.