Universidade do Minho
Escola de Ciências

Bárbara Daniela Oliveira da Silva

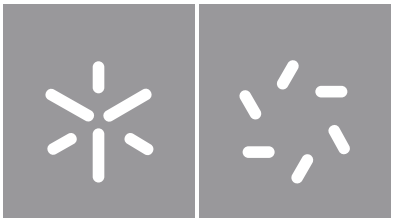
Time series forecasting with machine learning models informed by domain knowledge

janeiro de 2025

Time series forecasting with machine learning models informed by domain knowledge

Bárbara Silva

UMinho | 2025



Universidade do Minho
Escola de Ciências

Bárbara Daniela Oliveira da Silva

Time series forecasting with machine learning models informed by domain knowledge

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho efetuado sob a orientação de:
Professora Doutora Maria Fernanda Pires da Costa
Professor Doutor Vaibhav Shah

janeiro de 2025

Direitos de Autor e Condições de Utilização de Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos. Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada. Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



Atribuição-NãoComercial-CompartilhaIgual CC BY-NC-SA

https://creativecommons.org/licenses/by-nc-sa/4.0/

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Professor Doctor Fernanda Costa and Professor Doctor Vaibhav Shah, who were always available throughout this project. Their guidance and support were invaluable. I would also like to especially thank Cecília Martins for all the help and availability during the development of this project.

I would like to extend my heartfelt thanks to DTx for the opportunity to collaborate and develop this project within their organization. The resources, insights, and professional environment provided by DTx were fundamental to the progress and success of this work.

And last but not least, I would like to sincerely thank my family for their encouragement and support, which motivated me throughout this journey.

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

A previsão de séries temporais é uma tarefa crítica em muitos setores, particularmente no setor financeiro. Neste setor a previsão de cashflow é indispensável, uma vez que permite a uma empresa planear e gerir as entradas e saídas de dinheiro de uma forma eficaz. No entanto, esta tarefa pode ser desafiante quando os dados são limitados e/ou de pouca qualidade. Desta forma, esta dissertação tem como objetivo melhorar a previsão de séries temporais, particularmente o cashflow, através da integração do conhecimento de domínio em modelos de *machine learning*.

Inicialmente, foram exploradas várias abordagens para integrar o conhecimento de domínio nos modelos de previsão. Estas técnicas incluíam a integração de conhecimento, ou alterar a função de perda, ou treinar um modelo para aprender as relações hierárquicas entre variáveis, ou utilizar propriedades adicionais acerca das variáveis, denominadas por *meta-features*. No entanto, devido à complexidade e incerteza envolvidas na tradução desse conhecimento para uma forma que o modelo pudesse aprender, foi adotada uma abordagem mais prática. Foi utilizada uma arquitetura de *Neural ODE (Ordinary Differential Equation)* na tentativa de aprender as dinâmicas temporais dos dados; no entanto, mostrou-se insuficiente por si só ao aprender dinâmicas mais complexas.

Face a estes resultados, foi necessária uma arquitetura mais complexa. A arquitetura ODE-GRU, que combina a *Neural ODE* com a estrutura recorrente da *GRU (Gated Recurrent Unit)*. Esta arquitetura foi capaz de aprender dinâmicas mais complexas, capturando também dependências a longo prazo nos dados.

O objetivo final foi comparar os modelos com e sem conhecimento de domínio, de forma a avaliar a eficácia da incorporação do conhecimento de domínio na melhoria do desempenho dos modelos.

Os resultados indicaram que, no contexto do conjunto de dados utilizado, a integração do conhecimento do domínio não conduziu a melhorias significativas no desempenho do modelo. Este resultado pode dever-se ao facto de o histórico de dados disponíveis ser limitado, salientando a importância de um conjunto de dados com um registo histórico mais longo para se obter melhores resultados em estudos futuros.

Palavras-chave: Séries temporais, Previsão de Cashflow, Conhecimento do domínio, *Neural ODE*, *ODE-GRU*.

Abstract

Time series forecasting is a critical task in many fields, particularly finance. In this sector, cashflow forecasting is indispensable, as it allows a company to plan and manage cash inflows and outflows effectively. However, this task can be challenging when data is limited and/or of poor quality. This dissertation therefore aims to improve the forecasting of time series, particularly cashflow, by integrating domain knowledge into machine learning models.

Initially, various approaches were explored to integrate domain knowledge into the forecasting models. These techniques included integrating knowledge either by changing the loss function, training a model to learn the hierarchical relationships between variables, or by using additional properties about the variables, referred to as meta-features. However, due to the complexity and uncertainty involved in translating this knowledge into a form the model could learn from, a more hands-on approach was taken. A Neural ODE (Ordinary Differential Equation) architecture was employed as an attempt to capture the temporal dynamics of the data, however it proved to be insufficient by itself when trying to capture more complex dynamics.

Given these results, a more complex architecture was required. The ODE-GRU architecture, which combines the strengths of the Neural ODE with the recurrent structure of the GRU (Gated Recurrent Unit). This architecture was capable of learning more complex dynamics while also capturing long-term dependencies in the data.

The ultimate goal was to compare the models with and without domain knowledge, in order to evaluate the effectiveness of incorporating domain knowledge in enhancing predictive performance.

The results indicated that, in the context of the dataset used, the integration of domain knowledge did not lead to significant improvements in model performance. This outcome may be influenced by the limited historical data available, highlighting the importance of a dataset with a longer historical record for achieving better results in future studies.

Keywords: Time series, Cashflow forecasting, Domain knowledge, Neural ODE, ODE-GRU

Contents

1 Introduction **1**

1.1 Problem statement 1

1.2 Time Series Forecasting 2

1.3 Objectives 3

1.4 Structure of the Dissertation 4

2 Domain Knowledge Integration **6**

2.1 Previous Work on Relation Knowledge Integration 7

2.2 Previous Work on Logical Knowledge Integration 11

2.3 Previous Work on Scientific Knowledge Integration 17

3 Neural Networks **19**

3.1 RNN, LSTM and GRU 20

3.2 Neural ODE 26

3.3 ODE-RNN and ODE-GRU 29

4 Methodology and Dataset **31**

4.1 Programming Language 31

4.2 Dataset 32

4.3 DK Integration 35

4.4 Data Preprocessing 36

5 Experiments and Results Analysis **38**

5.1 Transformation of monthly data to daily data 38

5.2 Neural ODE Training and Results 40

5.3 GRU Training and Results 49

5.4 ODE-GRU Training and Results 58

5.5 Results Analysis 66

6 Conclusions and Future Work **70**

List of Figures

1 Example of Hierarchical Structure 9
2 Architecture of the perceptron 19
3 Short caption for list of figures 20
4 Short caption for list of figures 21
5 Distribution of number of observations per Group Id 33
6 Distribution of missing values per Group Id 34
7 Features distribution 35
8 Visualization of the transformation from monthly to daily data with dataset without DK . . 39
9 Visualization of the transformation from monthly to daily data with dataset with DK . . . 39
10 Neural ODE training results for group IDs G64 and G276 - no DK 41
11 Neural ODE testing results for group IDs G64 and G276 - no DK 42
12 Neural ODE training results - with DK 43
13 Neural ODE testing results - with DK 43
14 Neural ODE training results for group IDs G64 and G276 - no DK 44
15 Neural ODE testing results for group IDs G64 and G276 - no DK 45
16 Neural ODE training results - with DK 46
17 Neural ODE testing results - with DK 46
18 Neural ODE training results for group IDs G64 and G276 - no DK 47
19 Neural ODE testing results for group IDs G64 and G276 - no DK 48
20 Neural ODE training results - with DK 48
21 Neural ODE testing results - with DK 49
22 GRU training results for group IDs G64 and G276 - no DK 50
23 GRU testing results for group ids G64 and G276 - no DK 51
24 GRU training results - with DK 52
25 GRU testing results - with DK 52
26 GRU training results for group IDs G64 and G276 - no DK 53
27 GRU testing results for group IDs G64 and G276 - no DK 54

28 GRU training results - with DK 54
29 GRU testing results - with DK 55
30 GRU training results for group IDs G64 and G276 - no DK 56
31 GRU testing results for group IDs G64 and G276 - no DK 56
32 GRU training results - with DK 57
33 GRU testing results - with DK 57
34 ODE-GRU training results for group IDs G64 and G276 - no DK 59
35 ODE-GRU testing results for group IDs G64 and G276 - no DK 59
36 ODE-GRU training results - with DK 60
37 ODE-GRU testing results - with DK 61
38 ODE-GRU training results for group ids G64 and G276 - no DK 62
39 ODE-GRU testing results for group ids G64 and G276 - no DK 62
40 ODE-GRU training results - with DK 63
41 ODE-GRU testing results - with DK 63
42 ODE-GRU training results for group ids G64 and G276 - no DK 64
43 ODE-GRU testing results for group ids G64 and G276 - no DK 65
44 ODE-GRU training results - with DK 65
45 ODE-GRU testing results - with DK 66

List of Tables

1 Example of Hospital Admission Dataset 8
2 MSE for Different Models using Group ID G64 68
3 MSE for Different Models using Group ID G276 69

List of Abbreviations

CMVMC Custo das Mercadorias Vendidas e das Matérias Consumidas

COP Cost Of Personnel

DK Domain Knowledge

DPO Days Payable Outstanding

DSO Days Sales Outstanding

FOL First Order Logic

FSE Fornecimento e Serviços Externos

GRU Gated Recurrent Unit

KB Knowldge Base

LR Logistic Regression

LSTM Long-Short Term Memory

NN Neural Network

ODE Ordinary Differential Equation

PR Posterior Regularisation

RNN Recurrent Neural Network

SV Sales Volume

1 Introduction

1.1 Problem statement

Cashflow forecasting is an essential instrument that shapes economic strategies. This involves predicting the future cashflow of a business.

In theory, this is equivalent to a time series forecasting problem that machine learning models can handle. However, in multivariate scenarios, where the model is expected to learn causal dynamics among input variables, i.e., there are multiple factors (drivers) contributing to the outcome (driven) of a time series, the more traditional approaches require a significant volume of data. In some real use case of cashflow, the required volume of data is not always available due to low-frequency records and a relatively short history.

Without effective cashflow forecasting, businesses struggle with several critical financial aspects, such as estimating profits, recognising market trends, and protecting themselves against potential threats. The primary negative outcomes of inadequate forecasting include difficulties in managing cash flow, insufficient funds to operate, poor allocation of capital resources, and the accumulation of high levels of debt.

Cashflow forecasting involves predicting a bussiness's future financial position by predicting the expected cash inflows and outflows over a period of time. This process helps predict what cash on hand a business will have in its bank account [1].

When making strategic decisions, a company's cashflow forecasting is crucial since it provides insight into future cash positions, helping businesses make informed choices. From a strategic perspective, good cash flow forecasting is vital, as it will make it easier for a company to prepare for its operational needs, such as bond payments, debt repayment, and budgeting.

Despite its importance, real-world scenarios often present challenges when it comes to the quality and quantity of data available for making accurate cashflow forecasts. In many cases, historical data may be incomplete, inconsistent, or insufficient to capture the full financial dynamics of a business.

Given the limitations of available data, having an understanding about how the different variables, or drivers, are expected to influence the evolution of cashflow may help overcome the problems traditional models face when predicting cashflow. This understanding is the Domain Knowledge, or DK hereafter in this dissertation.

The Collaborative Laboratory for Digital Transformation Association – DTx, a non-profit organization focused on advancing digital solutions through applied research, proposed this project with the objective of developing an approach to use DK on a real dataset from one of DTx's associates, where several business were represented.

In cashflow forecasting problems, data is collected as a sequence of financial inflows and outflows that occur over time, meaning that the data are represented as time series.

1.2 Time Series Forecasting

In the context of cashflow forecasting, the ability to anticipate future events based on historical data is essential for strategic decision making. This type of analysis falls within the domain of time series, where data is a sequence of observations taken sequentially in time, and each observation depends on the previous ones. Using observations available at time t to forecast values at a future time $t + l$, can establish a solid foundation for economic and business planning. Assuming observations are available at discrete, equispaced intervals of time, in a cashflow forecasting problem, the cashflow c_t in the current month t and the cashflows in the previous months, $c_{t-1}, c_{t-2}, \ldots$, might be used to forecast cashflow c_{t+l} for $l = 1, 2, 3, \ldots, 12$ months ahead [2].

Time series forecasting can be challenging. Unlike problems involving unordered tabular data, time series problems involve the added complexity of sequence or temporal dependence among observations [3]. This complexity makes it necessary to handle the data in a specialized manner when fitting and evaluating models.

Linear models, such as ARIMA (Autoregressive Integrated Moving Average) models [4], have dominated time series forecasting due to their comprehensibility and effectiveness on numerous issues. Nevertheless, these traditional models have notable drawbacks:

1. Typically they do not handle missing or corrupted data;

2. They assume linear relationships between variables, potentially neglecting more intricate and nonlinear patterns existing in real-world scenarios;

3. They often concentrate on univariate data, ignoring the impact of multiple input variables prevalent

in real-life situations;

4. Many linear models are developed for one-step-ahead predictions, which may be insufficient for tasks demanding long-term forecasting.

However, in cases where traditional methods fall short and machine learning methods require elaborate feature engineering, deep learning techniques can be highly effective.

In deep learning, neural networks (NNs) [5] are capable of fitting a model to high-dimensional data. However, NNs might not always provide the best solution for every time series forecasting issue. Thus, due to the lack of perception of sequential data, a recurrence mechanism was introduced into NNs, originating Recurrent Neural Networks (RNNs) [6]. Due to the simplicity of RNNs, in the literature other more complex recurrence mechanisms were proposed, such as Gated Recurrent Units (GRU) [7] and Long-Short Term Memory (LSTM) [8]. Nevertheless, these NN architectures can only handle regularly sampled data, but often real time series data are irregularly-sampled or have missing data. To overcome this issue, preprocessing of the data is needed but this can be a source of error.

Neural Ordinary Differential Equations (Neural ODEs) are a NN architecture that approximates a time-dependent function to the dynamics of the data (an ODE), being able to perceive the sampling time of the data, overcoming the need to preprocess the data [9].

An ODE-RNN (ODE-GRU or ODE-LSTM) merges the capabilities of RNNs and Neural ODEs, creating a robust framework for managing time series data, regularly or irregularly sampled. Similar to Neural ODEs, ODE-RNNs (including ODE-GRU and ODE-LSTM) are designed to handle varying time intervals between data points, making them particularly well-suited for real-world scenarios where data collection is inconsistent or event-driven.

Thus, Neural ODEs, ODE-RNN, ODE-GRU and ODE-LSTM, are capable of autonomously capturing the temporal dependencies essential for complex time series forecasting tasks.

1.3 Objectives

Businesses often face challenges when it comes to forecasting cashflows, as reliable financial predictions are essential for decision-making and ensuring financial stability. However, a common issue encountered is the limited availability of historical cashflow data, which impairs the accuracy of more traditional approaches.

In response to this problem, this dissertation project, proposed by DTx, aims towards building a domain-informed machine learning model for cashflow forecasting and then use it on a real dataset from one of DTx's associates. The key objectives of this project are as follows:

1. Literature review and analysis to explore if and how DK can be integrated in machine learning models to improve time series forecasting;
2. Conduct thorough data analysis and preprocessing to ensure the data is properly prepared for the forecasting models;
3. Analysis of effectiveness of DK integration approaches in the context of one of DTx's associates;
4. Development of algorithm for DK integration to either facilitate the learning process or enhance model performance when data is insufficient;
5. Evaluate performance of domain-informed model and compare it with more traditional approaches.

1.4 Structure of the Dissertation

This dissertation is divided into 6 chapters: Introduction, DK Integration, Neural Networks, Methodology and Dataset, Numerical Experiments and Conclusions and Future Work. Each one of this chapters can be summarized as follows:

- Chapter 1: Introduction

  In this chapter the project's problem of cash flow forecasting is presented, detailing its context within the business and how it relates to a time series forecasting problem.

- Chapter 2: Domain Knowledge Integration

  In this chapter some of the state-of-art approaches for DK integration are presented;

- Chapter 3: Neural Networks

  In this chapter, a comprehensive overview of NNs is presented, including Recurrent NNs, Neural ODE, ODE-RNN, and ODE-GRU, which are the architectures used in this project;

- Chapter 4: Methodology and Dataset

  In this chapter, the programming language and tools employed in the development of all the architectures used, a descriptive analysis of the dataset used in this project, the formulation of the approach for DK integration based on the dataset, and all the preprocessing steps taken to prepare the data for the experiments described in the next chapter are presented.;

- Chapter 5: Experiments and Results Analysis

  In this chapter, the results of all the experiments conducted during this project are presented. The experiments were designed to evaluate the performance of different models, such as Neural ODE, GRU and ODE-GRU, under varying conditions and datasets. The analysis aimed to assess the models' ability to learn the dynamics of the data, and generalize to unseen data.

- Chapter 6: Conclusions and Future Work

  In this chapter the final conclusions and future work following up on this dissertation project are presented.

2 Domain Knowledge Integration

DK refers to the background knowledge of a specific area [10], such as finance. It involves more than just possessing information; it is about understanding how things function within a given sector. This understanding forms the foundation of data analysis, assisting in extracting pertinent insights and addressing practical problems.

In data science, domain expertise is essential, as it provides a deeper understanding of the specific context of the data, helping to make better analysis and decisions. By integrating DK into models, one can enhance the model's interpretability and improve prediction accuracy, as it allows the model to incorporate meaningful patterns and constraints that are often not captured by data-driven methods alone.

While machine learning techniques have evolved to improve model effectiveness, real-world challenges persist, such as limited, costly, or low-quality data. Integrating DK into machine learning models can help mitigate these challenges, enriching models with insights that enhance their reliability and performance in complex, practical applications.

The link between machine learning and human knowledge is rooted in the process of learning itself. As a matter of fact, NNs draw inspiration from the architecture of the brain. This connection was initially explored with the development of the perceptron [11], one of the pioneering computational models inspired by biological neurons. Until this day, researchers are striving to make machine learning as human-like as possible. In the context of machine learning, learning from data without prior knowledge represents one extreme, while incorporating manually driven hard-coded knowledge provides additional support [12]. The optimal approach for learning models lies in finding a balance between these two extremes.

DK can be categorized into three types: relation, logical and scientific knowledge [12]. The following brief definitions are provided:

1. **Relation Knowledge**: refers to information that describes how different objects are related to each other. The relation between these objects can be expressed in forms such as ordered pairs or relation symbols.

2. **Logical Knowledge**: logic can be expressed by propositional logic and first-order logic. Propositional logic is frequently applied in artificial intelligence (AI) for planning and decision making focusing

on functions and statements that go beyond true or false values to encompass elements of certainty and uncertainty. However, this form of knowledge has its limitations as it cannot see inside propositions and is unable to use the relationships between then.

3. **Scientific knowledge**: this type of knowledge encompasses principles from fields such as physics, mathematics and astronomy. All of these can be incorporated into a NN architecture.

Using DK when modeling data can be tricky, as the complexity of the knowledge often makes it difficult to translate into a form that can be integrated into data science models. This may lead to the creation of new features or the modification of existing ones.

DK can be expressed in various ways: through constraints, in terms of ranges and patterns or even on the models parameters; it can be used to transform the input data ; or even by modifying the way a model learns, through the adjustment of the loss function [13].

2.1 Previous Work on Relation Knowledge Integration

Integrating relational knowledge into machine learning models can improve their ability to generalize and increase accuracy when tackling complex tasks. Such knowledge can be presented as hierarchies, which represent the relationships among concepts, or as meta-features that provide additional information about the characteristics of the data.

2.1.1 Hierarchy

In [14] a framework that integrates DK in the form of hierarchies into machine learning models is proposed, particularly, logistic regression (LR) models, using a method called stacking [15], an operation that combines the outputs of several models to generate a more accurate final prediction. The method was tested on a binary classification problem to predict whether a pediatric patient would be readmitted into the hospital 30 days after being discharged.

LR is described as follows [14]:

$$\ln\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n = \theta^T x \tag{1}$$

where ln $\left(\frac{p}{1-p}\right)$ represents the logarithm of the odds ratio, with p being the probability of readmission to the hospital and $1-p$ being the probability of not being readmitted to the hospital, and the parameter $\theta = (\theta_0, \ldots, \theta_n)^T$ represents the weights associated with the features $x = (1, x_1, \ldots, x_n)^T$ of the LR model [16].

To optimize the model, the parameter θ is adjusted by minimizing the loss function (2):

$$L(\theta) = \sum_{i=1}^{m} (1 + \exp(-y_i(\theta^T x_i + c)) \tag{2}$$

where $x_i = (1, x_1^i, \ldots, x_n^i)$ represents the input feature, y_i is a binary label which takes values from $\{-1, 1\}$, θ is a weight vector, m is the number of examples and c is some random noise [14].

After optimizing the LR model, the probability of occurrence of a hospital readmission for any new test point $\tilde{z} = (1, z_1, \ldots, z_n)$, is computed as follows

$$p(\tilde{y}) = \frac{\exp(\theta^T \tilde{z})}{1 + \exp(\theta^T \tilde{z})} \tag{3}$$

Since this probability is a value between 0 and 1, to make a prediction a threshold term δ was introduced. If $p(\tilde{y}) \geq \delta$ the event is classified as hospital readmission within 30 days, otherwise not.

For illustrative purposes, let us consider the following table and figure, which were inspired by the structure described in [14]. The table (1) provides an overview of the hospital readmission dataset used in the experiment, where each row represents one hospital admission and a column represents a diagnosis, the binary values indicate whether a specific diagnosis was present during the patient's admission, and Figure (1) illustrates how the different diagnoses can be organized in a hierarchy.

Table 1: Example of Hospital Admission Dataset

Diagnosis 1	...	**Diagnosis k**	**Readmission**
1	...	0	0
0	...	1	1
...	...	...	...
1	...	0	1

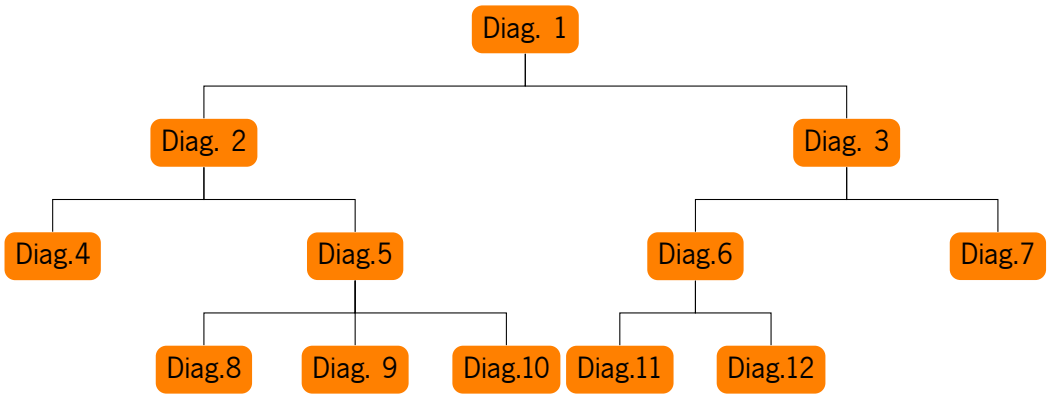


Figure 1: Example of Hierarchical Structure

In the experiment, numerous hierarchies were used, each of them would group clinically meaningful diagnoses.

Taking Figure (1) as an example, the top of the hierarchy, Diag.1 (meaning diagnosis 1), represents a more general concept, and as we go down the hierarchy, the concepts become more specific; the lowest levels of the hierarchy, the leafs, contain the most specific concepts.

Using the above example, the method can be resumed as follows. Having a hierarchy with parent-child elements select the group of concepts S that are non-leaf elements, in this case S={Diag.1, Diag.2, Diag.3, Diag.5, Diag.6}. For each concept $s \in S$, we would train an LR model using only its children C, for example, let s be Diag.5, we would train a model using only {Diag.8, Diag.9, Diag.10}. After training the model, a new column with the name s is added to the dataset and the model is saved in a list of models M. The process is repeated for every concept in S until the most general concept is learned, in this case until Diag.1 is learned. Then the most general concept is used for the prediction of hospital readmission.

Therefore, instead of trying to predict the outcome variable by having all attributes at the same level of the hierarchy, the idea is to arrange the attributes in a hierarchy and gradually predict the outcome attribute [14]. This is beneficial since relationships between attributes are often not simple or linear, and this hierarchical structure is better suited to capture more complex relationships.

2.1.2 Meta-features

Let's consider a setting where the input is represented by a vector of features.

In [17] a framework that incorporates additional information that we might have about each feature is proposed. This additional information is called meta-features. For this method, a training set $\{(x_i, y_i)\}_{i=1}^m$ was considered, where y_i is the label and $x_i \in \mathbb{R}^d$ is the raw input variable or features extracted from the inputs, being each of the d elements in the vector x a feature. An important distinction is that, in the context of the Support Vector Machine (SVM) approach, the labels y_i are either -1 or 1, whereas in logistic regression, they are typically 0 or 1. Let $\{u_1, \ldots, u_d\}$ be the vectors of meta-features, where $u_j \in \mathbb{R}^k$ are the meta-features of the j-th feature. An important point of these meta-features is that their value is fixed per feature. Due to this, the authors refer to the meta-features as prior knowledge.

Let w_j be the weight assigned to the feature j, with $j = 1, \ldots, d$. To incorporate the meta-features, the weights are interpreted as a function of the meta-features. Let G denote the class of mappings from meta-features to weights, then for $g \in G$, each weight satisfies $w_j = g(u_j)$ for $j = 1, \ldots, d$. The prior on w is induced by a prior over the functions in G, effectively defining a prior on the mapping from meta-features to weights.

The authors exploit the meta-features in order to create a better prior on w. For this purpose, they define the covariance between weight i and j as a function of the meta-features associated with the features i and j, i.e.

$$C_{ij} = C(u_i, u_j) \tag{4}$$

where C is a covariance function.

This function reflects the similarity between the weights and the meta-features. In general, it is expected that weights of features with similar meta-features are similar.

A learning algorithm uses the training set to select w. Assuming that the training set is linearly separable, there exists an infinite set of w such that all instances in training set $(w^T x_i)y_i \geq 1$. Once the covariance function is defined and in the context of the linear SVM approach, the optimization problem (5) is defined as:

$$w = \underset{w:(w^T x_i)y_i \geq 1, i=1,\ldots,m}{\text{argmin}} \; w^T C^{-1} w \tag{5}$$

This problem can be solved by applying the transformation $\tilde{x}_i = C^{1/2} x_i$ and defining $w = C^{1/2}\widetilde{w}$. From this transformation we get that $w^T C^{-1} w = (C^{1/2}\widetilde{w})^T C^{-1} C^{1/2}\widetilde{w} = \widetilde{w}^T (C^{1/2} C^{-1} C^{1/2})\widetilde{w} = \widetilde{w} I \widetilde{w} = \|\widetilde{w}\|^2$ and the constraints $(w^T x_i)y_i \geq 1$ can be rewritten as $((C^{1/2}\widetilde{w})^T x_i)y_i \geq 1$, i.e,

$(\widetilde{w}^T C^{1/2} x_i) y_i \geq 1$, i.e, $(\widetilde{w}^T \widetilde{x}_i) y_i \geq 1$. Therefore, the optimization problem is then rewritten as in (6)

$$\widetilde{w} = \underset{\widetilde{w}:(\widetilde{w}^T \widetilde{x}) y_i \geq 1}{\text{argmin}} \|\widetilde{w}\|^2 \tag{6}$$

This method was applied to a binary classification problem for handwritten digit recognition [17]. In this case, the meta-features u represented the position (x, y) of each feature (pixel) and the covariance function was defined as in (7)

$$C(u_i, u_j) = e^{-\frac{1}{2\sigma^2}\|u_i - u_j\|^2} \tag{7}$$

where u_i and u_j are the meta-features of weights associated with features i and j and σ is a parameter.

2.2 Previous Work on Logical Knowledge Integration

In situations where data is limited and of poor quality, integrating knowledge can be useful. The authors in [18] presented a way to incorporate loss terms for knowledge available as monotonic relationships between variables, monotonicity constraints, and a normal quantitative range where variables operate, approximation constraints [19].

2.2.1 Approximation constraint

The insights domain experts have about a quantitative range $[y_l, y_u]$, where a certain target variable Y operates, can be helpful in instances where noisy measurements would impact a model's performance. The violation of this constraint was represented in the function form as in (8)

$$g(\hat{Y}) = \begin{cases} 0, & \text{if } \hat{Y} \in [y_l, y_u] \\ y_l - \hat{Y}, & \text{if } \hat{Y} < y_l \\ \hat{Y} - y_u, & \text{if } \hat{Y} > y_u \end{cases} \tag{8}$$

This function was then incorporated into a loss function as shown in (9)

$$\text{Loss}_D(\hat{Y}) = \sum_{i=1}^{m} ReLU(y_l - y^i) + ReLU(y^i - y_u) \tag{9}$$

where

$$\text{ReLU}(z) = z^{+} = max(0, z) \tag{10}$$

The ReLU function was used because it allows to effectively model the violation of the constraint, making the output positive when the predictions do not satisfy the constraint imposed by the quantitative range of the variable.

For instance, the loss function (9), for the approximation constraint, is then incorporated into the generic loss function of the deep learning model as shown in (11)

$$\arg\min_{\theta} \text{Loss}(Y, \hat{Y}) + \lambda_D \text{Loss}_D(\hat{Y}) + \lambda R(\theta) \tag{11}$$

where $\text{Loss}(Y, \hat{Y})$ is the mean squared error between Y (true value) and $\hat{Y}$ (predicted value), $R(\theta)$ is an regularisation term, with θ representing the model parameters that are estimated during the training process. $\text{Loss}_D(\hat{Y})$ is the domain loss function used directly in the NN to enforce the model to learn from the training data accordingly with the range constraints. λ_D and λ are penalty hyper-parameters that determine the weight of the domain loss and the regularizer, respectively.

2.2.2 Monotonicity constraint

Let x_1 and x_2 be two variables that represent measurements of the same event. These variables share a monotonic relationship if, as the value of one variable increases (or decreases), the value of the other variable also increases (or decreases). Let us take for example a function $y = h(x)$ such that $x_1 > x_2 \Rightarrow h(x_1) > h(x_2)$. Then x_1 and x_2 are said to share a monotonic relationship.

This type of constraint was incorporated into a loss function as shown in (12)

$$\text{Loss}_D(\hat{Y}_1, \hat{Y}_2) = \sum_{i=1}^{m} \mathbb{I}\left((x_1^i < x_2^i) \wedge (\hat{y}_1^i > \hat{y}_2^i)\right) \times \text{ReLU}(\hat{y}_1^i - \hat{y}_2^i) \tag{12}$$

In the loss function (12) the constraint imposed is $\hat{Y}_1 < \hat{Y}_2$. The function $\mathbb{I}$ is the identity function, which returns TRUE if the logic value of the conjunction is TRUE, and returns FALSE otherwise.

In this context, the identity function ensures that the model respects the monotonicity constraint since it identifies instances in which the model's predictions violate the constraint.

An example of a monotonic relationship can be found in [20] where the authors attempted to model a lake's temperature (Y) using the physical relationships between the temperature, the density (ρ) and the depth of water (d). It was known that the water density increased monotonically with the depth. Thus, for two different depths d_1 and d_2 on the same time instant t the relationship was described as follows in (13)

$$\rho[d_1, t] - \rho[d_2, t] \leq 0 \quad \text{if} \quad d_1 < d_2 \tag{13}$$

The objective was to build a loss function that incorporated this relationship. Given a grid of input features with multiple depth levels and time instants, for each pair of consecutive depths d_i and d_{i+1} ($d_i < d_{i+1}$), the difference in density estimates at a specific time instant t is calculated. This difference is defined as in (14)

$$\Delta[i, t] = \hat{\rho}[d_i, t] - \hat{\rho}[d_{i+1}, t] \tag{14}$$

A positive value of $\Delta[i, t]$ can be seen as a violation of the relationship described in (13) on the depth d_i and time t. This can be evaluated as a non-zero occurrence of $ReLU(\Delta[i, t])$. Therefore, the mean of all violations across consecutive depth pair and time instant is considered as the proposed loss function (15), that incorporates the monotonic relationship between the density and the depth of water.

$$Loss(\hat{Y}) = \frac{1}{n_t(n_d - 1)} \sum_{t=1}^{n_t} \sum_{i=1}^{n_d - 1} ReLU(\Delta[i, t]) \tag{15}$$

where n_d is the number of depth values at n_t time steps. For more details see [20].

2.2.3 Structured DK

The integration of structured DK can sometimes require complex modeling. In [21] the authors used Posterior Regularization (PR), a technique introduced in [22] that provided a framework for incorporating structured constraints, to build a probabilistic model capable of learning constraints as rewards and flexible to arbitrary constraints.

Like in [21], let us consider a generative model that can be expressed as $x \sim p_\theta(x)$ with parameters θ. The original objective of the model p_θ was to minimize a certain loss function $L(\theta)$.

Let us now consider a constraint function $f_\theta(x) \in \mathbb{R}$ where a higher value of $f_\theta(x)$ indicates a stronger alignment between x and the particular knowledge. This constraint can be imposed on the model by maximizing the expectation $E_{p_\theta}[f_\theta(x)]$.

However, to ensure versatile application and efficient optimization, PR imposes the constraint on an auxiliary distribution q. This distribution is encouraged to stay close to p_θ using a term called Kullback-Leibler (KL) Divergence:

$$L(\theta, q) = \mathrm{KL}(q(x) \| p_\theta(x)) - \alpha E_q[f_\theta(x)] \tag{16}$$

where $\alpha \geq 0$ is the weight of the constraint term.

The PR objective for learning the model is written as:

$$\min_{\theta,q} L(\theta) + \lambda L(\theta, q) \tag{17}$$

where $\lambda \geq 0$ is the balancing hyperparameter. For details on how to solve the problem (17) see [21].

The method was applied on an image generation task. Given an image of a person and a different body pose, the goal was to produce an image of the same person in the specified new pose. For this example, the constraint function $f_\theta(x)$ included a human parsing module that classified each pixel of a person's image into possible body parts. The goal of this function was to encourage each body part in the generated image to match its corresponding part in the actual image [21].

2.2.4 Semantic Based Regularization

DK expressed as a collection of first-order logic (FOL) clauses can be integrated into a machine learning model using an approach called Semantic Based Regularization (SBR). SBR was presented in [23] as a Statistical Relation Learning (SRL) framework and in [24] served as a fundamental framework to represent prior knowledge expressed as a set of FOL clauses. SRL is especially used due to its ability to inject logic knowledge into learning, because it transforms the knowledge into a set of continuous constraints that can be incorporated into the loss functions that are usually used in machine learning.

In [24] the authors consider a knowledge base (KB) consisting of a set of FOL clauses, where each predicate, representing a class or category, corresponds to a task to be learned and where all possible groundings for these predicates are provided or some of them are estimated by the functions to be learned.

Let's consider a multi-task problem, where a set of T functions must be learned and let $f = [f_1, \ldots, f_T]$ be the vector of functions. It is assumed that a set of H functionals $\phi_h(f), 0 \leq \phi_h(f) \leq 1, h = 1, \ldots, H$ is provided to describe prior knowledge about a learning task. Each j-th function is associated to a set $\mathcal{X}_j$ of pattern representations. Each pattern in this set is represented by a vector of features. A set of examples $\varepsilon_j \subset \mathcal{X}_j$ is provided to express the desired function outputs over a subset of the available patterns. The vector of values obtained by applying the function f_k to the set of patterns $\mathcal{X}_k$ is denoted by $f_k(\mathcal{X}_k)$, while $f(\mathcal{X}) = f_1(\mathcal{X}_1) \cup f_2(\mathcal{X}_2) \cup \ldots$.

The generic loss function was defined as follows:

$$C_e[f(\mathcal{X})] = \sum_{k=1}^{T} \left(\|f_k\|^2 + \lambda_l \sum_{x \in \varepsilon_k} L(f_k(x), y_k(x)) \right) + \sum_{h=1}^{H} \lambda_h L_c(\phi_h(f(\mathcal{X})) \tag{18}$$

where the first term is a regularization term, $\|f_k\|^2$, that penalizes solutions that are nonsmooth, ε_k is the set of labelled data available for each k-function, $L(\cdot, \cdot)$ is a loss function, $y_k(x)$ the target output value for the x pattern for task k and λ_l a weight associated to the labelled portion of the generic loss function. $L_c(\cdot)$ is the loss function used to incorporate the constraints and λ_h is the weight associated to the h-th constraint.

In order for the formulas in the KB to be incorporated into a model they need to be converted into a continuous and differential form. For this purpose, fuzzy logic is used to enable these formulas to have a degree of truth that goes further than just being classified as true or false.

For each FOL formula, the basic logic operations $(\neg, \vee, \wedge, \Rightarrow)$ are replaced by a unit that computes the logic operation using a t-norm (details of the operations computed by the units are in a table in [24]). Therefore, given a formula F_i in the KB, its representation in fuzzy logic, $\phi_i(\cdot)$, depends on which t-norm is used.

For demonstrative purposes, let us consider the example given in [24], $\forall x[\neg A(x)] \vee [B(x) \wedge C(x)]$. Since the experiment in [24] was performed on an image classification task to determine the class of an animal and following the explanation in [23], let us think of A, B and C as animal classes. Let d be an animal to be classified and $A(d)$, $B(d)$ and $C(d)$ be the functions for the classes. The functions $f_A(d), f_B(d)$ and $f_C(d)$ must be estimated to approximate $A(d), B(d)$ and $C(d)$, respectively. For any given grounding of d, an example of the t-norm expression for the propositional part of the formula, i.e,

$[\neg A(x)] \vee [B(x) \wedge C(x)]$, is $t_E(f(d)) = 1 - f_A(d) + f_A(d) \times f_B(d) \times f_C(d)$, in this case the operation $x \wedge y$ was converted into $x \times y$, $x \vee y$ into $x + y - x \times y$ and $\neg x$ into $1 - x$. The only part left is the quantifier. For existential quantifiers we would take the maximum of the t-norm expression over the domain of the quantified variable, in the case of a universal quantifier, like the one in the example, we would take the average of the t-norm expression. In this way, we would get $\phi(f(\mathcal{D})) = \frac{1}{|D|} \sum_{d \in \mathcal{D}} t_E(f(d)) = \frac{1}{|D|} \sum_{d \in \mathcal{D}} 1 - f_A(d) + f_A(d) \times f_B(d) \times f_C(d)$, where $\mathcal{D}$ is the set of all possible groundings for d.

2.2.5 Semantic Loss Function

In [25] the authors present a method to incorporate symbolic knowledge, to be more specific, symbolic knowledge that connects the different outputs of a NN. This knowledge takes the form of a constraint (or sentence) in Boolean logic. The goal is to make a NN capable of making predictions while subjected to these constraints.

Let us consider a logical sentence α and a Boolean variable X; $\neg X$ represents its negation and x represents its instantiation ($X = 0$ or $X = 1$). The sets of variables are written in bold uppercase **X**. A state **x** is an instantiation of all variables **x**. A state **x** satisfies a sentence α, denoted $\mathbf{x} \vDash \alpha$, if the sentence is true.

The symbolic knowledge was incorporated into a semantic loss function, $L^s(\alpha, p)$, defined as follows:

$$L^s(\alpha, p) \propto -log \sum_{\mathbf{x} \vDash \alpha} \prod_{i : \mathbf{x} \vDash X_i} p_i \prod_{i : \mathbf{x} \vDash \neg X_i} (1 - p_i) \tag{19}$$

where α is a sentence defined over a set of variables $\mathbf{X} = \{X_1, \ldots, X_n\}$, p is a vector of probabilities for the variables **X**. Each element p_i is the predicted probability of the variable X_i and corresponds to a single output of the NN.

Intuitively, the semantic loss is directly related to the negative logarithm of the probability of generating a state that satisfies the constraint, based on values sampled according to p.

The proposed method is designed to be broadly applicable and seamlessly compatible with any feed-forward NN. The semantic loss is simply another regularization term that can be directly plugged into an existing loss function. For some weight w, the new loss function is defined as:

$$L(\theta, \alpha, p) = \text{existing loss} + w.\text{semantic loss}$$

where θ denotes the parameters of the NN.

Following the same idea as one of the examples given by the authors of [25]. Let us focus on a multiclass problem with two classes x_1 and x_2. The constraint for this problem requires that only one of the classes is true, therefore this constraint can be encoded as $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$. To be able to implement the constraint in the loss function, the logical operations are converted into arithmetic operations: the operation AND ($\wedge$) is replaced by the multiplication ($\times$) and the operation OR ($\vee$) is replaced by the sum ($+$). Then by getting the probabilities associated to each class we obtain $P(x_1)P(\neg x_2) + P(\neg x_1)P(x_2)$.

From this, the semantic loss function can be written as $L^s(\alpha, p) = -\log(P(x_1)P(\neg x_2) + P(\neg x_1)P(x_2))$.

2.3 Previous Work on Scientific Knowledge Integration

Prior DK such as laws of physics can be used as constraints in a NN. The work in [26] introduced a method for training NNs without explicit labels. Instead of directly use the labels, the network is trained so that the outputs obey the constraints. This is achieved by building a constraint function the penalizes inconsistencies with the DK, which is given by the laws of physics. This way, the goal of the method is to train a network f that maps inputs to the desired outputs without the need of direct examples of those outputs.

The full details of the method are available in the original paper [26], but its core idea can be described as follows.

Let us consider a training set $D = \{(x_1, y_1), \ldots (x_n, y_n)\}$ in which each pair (x_i, y_i) is formed by an instance $x_i \in X$ and the corresponding output label $y_i \in Y$. The goal is to learn a function $f : X \to Y$, which maps input values to the corresponding output values. The optimal solution for f is found via (20)

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{n} l(f(x_i), y_i) \tag{20}$$

where $l : Y \times Y \to \mathbb{R}$ is a loss function and $\mathcal{F}$ represents NNs parameterized by their weights.

In the experiments conducted in [26], the DK was modeled on the outputs by using a constraint function $g : X \times Y \to \mathbb{R}$ used to penalize inconsistencies with the DK. One experiment aimed to learn the height of an object in each frame using a recorded video of the object being thrown across the field of view. The goal was to build a model in which a sequence of N images produces N heights. Given a vector of images **x**, the model was trained not using the direct labels, **y**, but by finding an object that obeys the laws of physics

of a free-falling object.

Under the influence of gravity, an object experiences a constant acceleration of $a = -9.8m/s^2$ and its height over time follows a parabolic trajectory. The height of the object over time forms a parabola that can be described as in (21):

$$\mathbf{y}_i = y_0 + v_0(i\Delta t) + a(i\Delta t)^2 \tag{21}$$

where $\Delta t = 0.1s$ represents the time interval between frames. This equation acts as a constraint that the mapping f^* must satisfy. The model is trained in a way that given any trajectory of N height predictions, $f(\mathbf{x})$, we fit a parabola, given by the equation above, to this predictions, this way we ensure that the predicted heights of an object align with the physical laws.

To build the function g that will act as the constraint function, in the experiment, the authors started by defining the fixed acceleration term $\mathbf{a}$ as a vector $\mathbf{a} = [a(\Delta t)^2, a(2\Delta t)^2, \ldots, a(N\Delta t)^2]$. The adjusted height, $\hat{\mathbf{y}}$ was then calculated by subtracting the contribution of the fixed acceleration from $f(\mathbf{x})$, i.e ,$f(\mathbf{x}) - \mathbf{a}$, then a line is fitted to account for the initial height and velocity, i.e, $\mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T(f(\mathbf{x}) - \mathbf{a})$ where $\mathbf{A}$ is a matrix $N \times 2$ with each row representing $[i\Delta t, 1]$. Finally, the acceleration term is added back to reconstruct the full trajectory. Formally, this was expressed as in (22)

$$\hat{\mathbf{y}} = \mathbf{a} + \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T(f(\mathbf{x}) - \mathbf{a}) \tag{22}$$

The inconsistency between the model predictions $f(\mathbf{x})$ and the parabola is then quantified using the constraint function g defined as follows in (23).

$$g(\mathbf{x}, f(\mathbf{x})) = \sum_{i=1}^{N} |\hat{\mathbf{y}}_i - f(\mathbf{x})_i| \tag{23}$$

Then, the optimal solution, $\hat{f}^*$, could be found via (24).

$$\hat{f}^* = \underset{f \in \mathcal{F}}{\text{argmin}} \sum_{i=1}^{n} g(x_i, f(x_i)) \tag{24}$$

In a real-world scenario involving one of DTx's associates, the available DK contained a degree of uncertainty, making it difficult to transform into explicit, precise information that could be integrated using the previous studied methods. Consequently, instead of applying one of these standard methods, a more hands-on approach was adopted, leveraging NNs to more effectively address the forecasting challenge.

3 Neural Networks

NNs are widely used for learning highly complex functions and often yield outstanding predictions for challenging machine learning tasks. These networks are called neural because they simulate the learning process of the brain.

In 1957, Frank Rosenblatt evolved the idea of replicating how biological neurons worked in the human brain by coming up with the simplest NN, which consisted of a single input layer and a single node in the output layer, called the Perceptron. The architecture of the perceptron is shown in Figure (2)

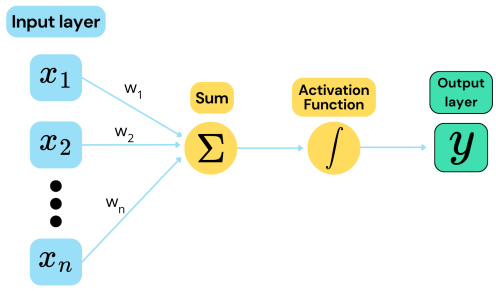


Figure 2: Architecture of the perceptron

A NN performs three main operations to map inputs to outputs:

1. **Weighted sum**: Each input x_i is multiplied by the corresponding weight w_i and the results are summed. In many situations, it is necessary to incorporate an additional variable b referred to as bias, that helps adjust the outputs of the neuron;

$$z = \sum_{i=1}^{n} w_i x_i \tag{25}$$

2. **Activation function**: In order to make a prediction, the value z is passed through an activation function, σ, which introduces non-linearity into the output, allowing the network to learn and represent complex patterns;

3. **Final prediction**: The output/predicted value, $\hat{y}$, is computed after applying the activation function. In the case of binary classification, the *sign* function, which returns $+1$ if z is greater then 0 and

-1 otherwise, is commonly used.

$$\hat{y} = sign\{z\} \tag{26}$$

NNs can include additional intermediate layers, called hidden layers, located between the input and output layers, such networks are called **multilayer NNs**. The architecture of a multilayer network can be seen in Figure (3). These networks are feedforward NNs, which means that information travels in a single direction, from the input nodes, through the hidden nodes, to the output nodes, without forming any loops or cycles. The standard structure of feedforward networks assumes that every node in one layer is connected to each node in the following layer [27].

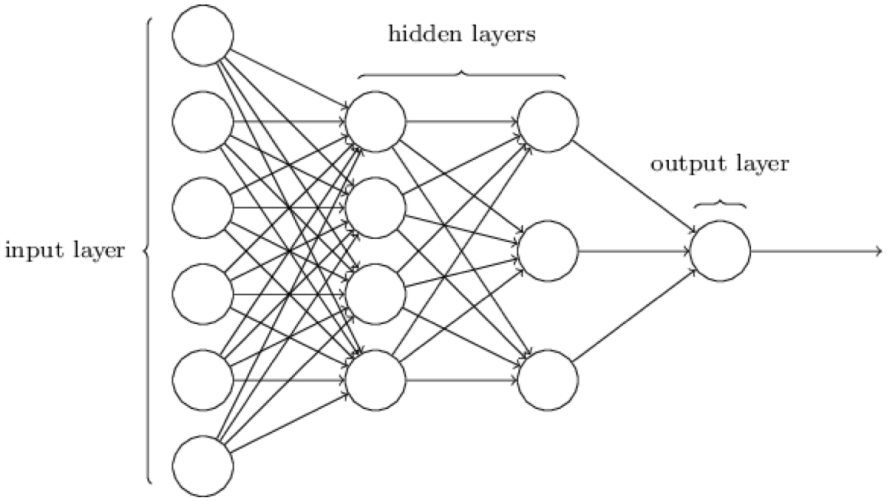


Figure 3: Illustration of an Multilayer NN, reproduced from [28].

NNs excel at processing noisy or incomplete data, enabling accurate learning and prediction even when certain information is absent. They provide direct support for multivariate inputs, facilitating multivariate forecasting, and can handle multiple output values, aiding in multi-step forecasting [3] .

3.1 RNN, LSTM and GRU

3.1.1 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are designed to handle sequential data, such as time-series. These networks essentially have a memory component, enabling them to retain information from previous inputs.

This capacity to recall past data is crucial for forecasting time-related outcomes [29].

In traditional NNs, the inputs and outputs operate independently. However, when forecasting the next observation in a sequence, it is crucial to take into account the preceding observations. An RNN fulfills this need by using processing units, more specifically a series of fixed units, each corresponding to a specific time step. Each unit possesses an internal state, known as the hidden state, which denotes the past information that the network retains at any particular time step [30] . This hidden state is updated at every time step to signify the change in knowledge of the network about the past. The update of the hidden state can be expressed as follows:

$$h_t = f(W_x x_t + W_h h_{t-1} + b) \tag{27}$$

where $h_t \in \mathbb{R}^m$ is the current hidden state, with m representing the number of units in the hidden layer, $h_{t-1} \in \mathbb{R}^m$ is the previous hidden state, $x_t \in \mathbb{R}^n$ is the input at time step t, with n representing the number of features, $W_x \in \mathbb{R}^m \times \mathbb{R}^n$ is the weight matrix for the input, $W_h \in \mathbb{R}^m \times \mathbb{R}^m$ is the weight matrix for the hidden state, $b \in \mathbb{R}^m$ is a bias vector and f is a non-linear activation function.

Figure (4) shows an RNN in vertical format, where the layers are stacked vertically (a) and the RNN with the layers unfolded in time (b).

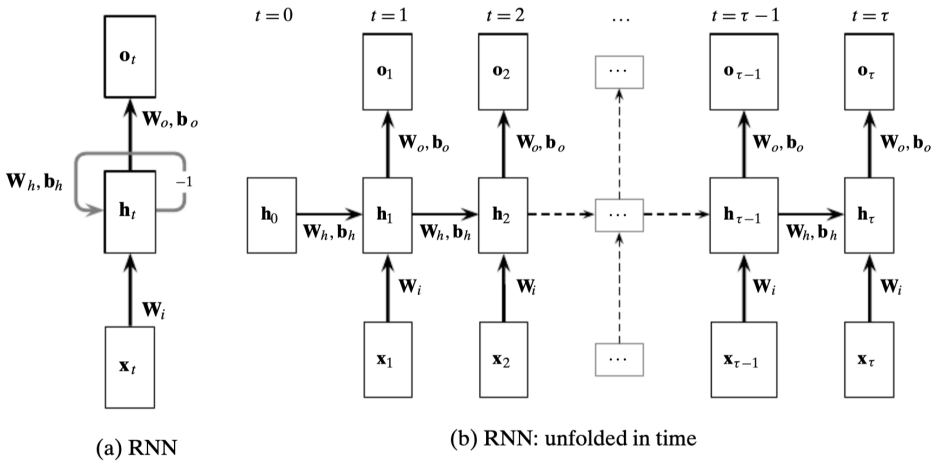


Figure 4: Illustration of an RNN, reproduced from [16].

One significant limitation of conventional RNNs is their dependence on evenly spaced data points. When

dealing with irregular timesteps, as in common in real-world datasets, pre-processing becomes necessary. This added step can increase complexity and potentially cause errors in the model's predictions. Also since RNNs work by handling sequences in discrete time steps, they might not effectively capture the inherent continuous dynamics of some datasets.

3.1.2 Long-Short Term Memory

One of the problems of training RNNs is that the values used to update the weights of the network can become either extremely small (vanishing gradient) or excessively large (exploding gradient). Long-Short Term Memory (LSTM) networks help mitigate these problems by using gate neurons to control access to the hidden states.

RNNs update the hidden state as shown in (27). To update the hidden state, LSTM networks use a vector c_t called internal memory vector and three gate vectors - input gate vector, forget gate vector and output gate vector [16]. Each gate vector plays a different role in the LSTM network. The input gate vector i_t controls how much of the candidate update vector u_t influences c_t, the forget gate vector f_t controls how much of the previous memory vector to forget and the output gate vector, o_t, controls how much of the memory state is retained for the hidden state.

Given an input vector x_t and a previous hidden state h_{t-1}, at each time step t, the LSTM network first computes the **candidate update vector** u_t as follows:

$$u_t = tanh(W_u x_t + U_u h_{t-1}) \tag{28}$$

where $x_t \in \mathbb{R}^n$ is the input at timestep t, with n representing the number of features, $h_{t-1} \in \mathbb{R}^m$ is the previous hidden state with m representing the number of hidden units, $W_u \in \mathbb{R}^m \times \mathbb{R}^n$, $U_u \in \mathbb{R}^m \times \mathbb{R}^m$ are learnable weight matrices and $tanh$ represents the hyperbolic tangent which is defined as in (29).

$$tanh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}} \tag{29}$$

Thereafter, all three gates are updated as follows:

1. **Input Gate**: $i_t = \sigma(W_i x_t + U_i h_{t-1})$

2. **Forget Gate**: $f_t = \sigma(W_f x_t + U_f h_{t-1})$

3. **Output Gate**: $o_t = \sigma(W_o x_t + U_o h_{t-1})$

where $W_i, W_f, W_o \in \mathbb{R}^m \times \mathbb{R}^n$, $U_i, U_f, U_o \in \mathbb{R}^m \times \mathbb{R}^m$ are learnable weight matrices and σ represents the sigmoid function which is defined as in (30).

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{30}$$

After updating all gates, the network computes the internal memory and hidden state vector as follows:

1. **Internal memory**: $c_t = i_t \odot u_t + f_t \odot c_{t-1}$

2. **Hidden state**: $h_t = o_t \odot tanh(c_t)$

where c_{t-1} is the previous internal memory and $\odot$ represents element-wise product, meaning each element in a specific position in one matrix is multiplied by the element in the corresponding position in the other matrix.

3.1.3 Gated Recurrent Unit

While LSTMs effectively mitigate the vanishing gradient problem through their gating mechanisms, Gated Recurrent Units (GRUs) simplify this structure by combining certain gates, reducing the computational complexity while maintaining the ability to capture long-term dependencies.

GRU network, introduced by Cho et al. in 2014, consists mainly of two gates: the reset gate and the update gate, which control the hidden state of the network at each time step. The reset gate dictates the portion of the previous hidden state that should be forgotten, while the update gate decides how much of the new input should be used to update the hidden state [31]. The output of the network is calculated based on the updated hidden state. The hidden state is calculated as follows:

1. **Reset Gate**: $r_t = \sigma(W_r x_t + U_r h_{t-1})$

2. **Update Gate**: $z_t = \sigma(W_z x_t + U_z h_{t-1})$

3. **Candidate hidden state**: Estimate of the new hidden state, $\tilde{h}_t = tanh(Wx_t + r_t \odot (Uh_{t-1}))$

4. **Hidden state**: $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$

where $x_t \in \mathbb{R}^n$ is the input at timestep t, with n representing the number of features, $h_{t-1} \in \mathbb{R}^m$ is the previous hidden state and $h_t \in \mathbb{R}^m$ is the current hidden state, with m representing the number of hidden units, $W_r, W_z \in \mathbb{R}^m \times \mathbb{R}^n$, $W \in \mathbb{R}^m \times \mathbb{R}^m$ are learnable weight matrices, σ represents the sigmoid function (30) , $tanh$ represents the hyperbolic tangent (29) and $\odot$ represents element-wise product.

In a typical setting, the update of the formulas enumerated above is performed by a GRUCell which is represented by $h_i = GRUCell(h_{i-1}, x_i)$. However, in the experiments executed in this project, the goal is to predict future values based on past observations. To align with this objective, the update of the formulas was modified so that to obtain the hidden state at time step i, we use the previous hidden state h_{i-1} and previous input x_{i-1}, i.e, $h_i = GRUCell(h_{i-1}, x_{i-1})$. This is known as Teacher Forcing [32], a method for training RNN models using the ground truth from a prior time step as input.

The following algorithm outlines the training algorithm employed for the GRU model.

Algorithm 1 Training of the GRU

Input: Data points $\{x_i\}_{i=0,\dots,N}$

Initialize:

    $\theta_0 \leftarrow$ Initialize parameters # initialize model parameters

    Optimizer $\leftarrow$ Adam(θ) # choose optimizer for minimising $L(\theta)$

for n = 1,..., epochs **do** # epochs specifies the number of training iterations

    $h_0 = \mathbf{0}$ # initial hidden state, initialized to zero

    for i = 1,...,N **do**

        $h_i = GRUCell(h_{i-1}, x_{i-1})$

        $o_i = LinearLayer(h_i)$

    end for

    Evaluate loss function $L(\theta_{n-1}; \{o_i\}_{i=1,\dots,N}, \{x_i\}_{i=1,\dots,N})$ # compute loss for all predictions

    $\nabla L \leftarrow$ Compute gradient $\nabla L(\theta_{n-1}; \{o_i\}_{i=1,\dots,N}, \{x_i\}_{i=1,\dots,N})$ #backpropagation

    $\theta_n \leftarrow$ Optimizer.Step($\theta_{n-1}, \nabla L$) # update model parameters

end for

return $\{o_i\}_{i=1,\dots,N}$; h_N

To make predictions we use the final hidden state and the last input from the training data as the starting point. Using these, the model generates the first prediction for the next time step. This prediction is then recursively fed back into the model as input for the subsequent time step, along with the updated hidden state, to produce the next prediction. This process continues iteratively until predictions are made for all desired future time steps. The following algorithm illustrates the process used to generate predictions.

Algorithm 2 GRU Forecasting

Input: Set of test points $\{x_i\}_{i=N+1,...,M}$ Last hidden state h_N and last data point from training set x_N

$h_{N+1} = GRUCell(h_N, x_N)$

$o_{N+1} = LinearLayer(h_{N+1})$

for i = N+1,...,M **do**

    $h_i = GRUCell(h_{i-1}, o_{i-1})$

    $o_i = LinearLayer(h_i)$

end for

return $\{o_i\}_{i=N+1,...,M}$

3.2 Neural ODE

An Ordinary Differential Equation (ODE) is an equation that contains one or more functions of a single independent variable and their derivatives. It relates the function to its derivatives [33]. The general form of an ODE is shown in (31).

$$F(t, x(t), x'(t), x''(t), ..., x^{(n)}(t)) = 0 \tag{31}$$

where $x(t)$ is the unknown function of t, $x'(t), x''(t), ..., x^{(n)}(t)$ are its derivatives and n is the order of the highest derivative. For instance, a first-order ODE can be written as in (32)

$$\frac{dx(t)}{dt} = f(t, x(t)) \tag{32}$$

where f is a known real-valued function that defines the ODE.

A system of ODEs consists of multiple ordinary differential equations that involve multiple dependent variables, each being a function of the same independent variable t.

For instance, the general form of a system of first-order ODEs is defined as follows:

$$\begin{cases} \dfrac{dx_1(t)}{dt} = f_1(t, x_1(t), x_2(t), \ldots, x_n(t)) \\ \dfrac{dx_2(t)}{dt} = f_2(t, x_1(t), x_2(t), \ldots, x_n(t)) \\ \vdots \\ \dfrac{dx_n(t)}{dt} = f_n(t, x_1(t), x_2(t), \ldots, x_n(t)) \end{cases}$$

where $x_1(t), x_2(t), \ldots, x_n(t)$ are the unknown real-valued functions and $f_1, f_2, \ldots, f_n$ are known given functions that define the ODE system.

Neural ODEs are a recent NN architecture introduced by Chen et al. in 2018 [9]. A neural ODE is essentially a network that fits the solution of an ODE or a system of ODEs, with a given initial condition, to the data. Unlike RNNs, which define a discrete sequence of hidden layers, neural ODEs represent the continuous dynamics of the hidden state through an ODE [9]. The hidden state $h(t)$ is defined as the solution of the initial value problem (IVP):

$$\frac{dh(t)}{dt} = f_\theta(h(t), t) \quad \text{with} \quad h(t_0) = h_0 \tag{33}$$

where the function f_θ defines the dynamic of the hidden state using a NN with parameters θ and h_0 is the initial condition, i.e., the value of the function $h(t)$ at the point t_0.

The hidden state $h(t)$ is continuously defined and can be computed at any given time point using a numerical ODE solver:

$$h_{t_1}, \ldots, h_{t_N} = ODESolve(f_\theta, h_0, \{t_0, t_1, \ldots, t_N\}) \tag{34}$$

where $\{t_0, t_1, \ldots, t_N\}$ represents the set with t_0 being the initial time point and t_N the final time point, and $h_{t_1}, \ldots, h_{t_N}$ represents the hidden states at time points $t_0, t_1, \ldots, t_N$.

However, neural ODEs may have difficulties in accurately modeling certain time-based dynamics, especially when dealing with complex, long-range dependencies. To overcome these challenges, it is necessary to incorporate a more advanced architecture that offers the advantages of continuous-time modeling and provides a framework capable of addressing complex and long temporal dependencies, an architecture

called ODE-RNN, more specifically, ODE-GRU, GRUs are an advanced variant that significantly boosts the RNNs' ability to address long-term dependencies efficiently.

The following algorithm outlines the training algorithm employed for the Neural ODE model.

Algorithm 3 Training of the Neural ODE

Input: Data points $\{(x_i, t_i)\}_{i=0,\ldots,N}$

Initialize:

    $\theta_0 \leftarrow$ Initialize parameters # initialize model parameters

    Optimizer $\leftarrow$ Adam(θ) # choose optimizer for minimising $L(\theta)$

$c_0 = x_0$ # define x_0 as the initial condition

for n = 1,..., epochs **do** # epochs specifies the number of training iterations

    $\{o_i\}_{i=1}^{N} = ODESolve(f_\theta, c_0, \{t_0, \ldots, t_N\})$ # f_θ is a function modeled by an NN

    Evaluate loss function $L(\theta_{n-1}; \{o_i\}_{i=1,\ldots,N}, \{x_i\}_{i=1,\ldots,N})$ # compute loss for all predictions

    $\nabla L \leftarrow$ Compute gradient $\nabla L(\theta_{n-1}; \{o_i\}_{i=1,\ldots,N}, \{x_i\}_{i=1,\ldots,N})$ #backpropagation

    $\theta_n \leftarrow$ Optimizer.Step$(\theta_{n-1}, \nabla L)$ # update model parameters

end for

return $\{o_i\}_{i=1}^{N}$

The following algorithm details the prediction process for the Neural ODE model, where the initial condition and initial time step used during training are provided, along with the target time steps for which predictions are required.

Algorithm 4 Neural ODE Forecasting

Input: Set of test points $\{(x_i, t_i)\}_{i=N+1,\ldots,M}$

$c_0 = x_0$ # define x_0 as the initial condition

$\{o_i\}_{i=N+1}^{M} = ODESolve(f_\theta, c_0, \{t_0, t_{N+1} \ldots, t_M\})$

return $\{o_i\}_{i=N+1}^{M}$

3.3 ODE-RNN and ODE-GRU

An ODE-RNN is a continuous-time model that defines the state between observations as the solution of the IVP: $h'_i = ODESolve(f_\theta, h_{i-1}, [t_{i-1}, t_i])$ and then at each observation, uses a RNNCell to update the hidden state: $h_i = RNNCell(h'_i, x_i)$ [34] .

Although the ODE-RNN updates the hidden state efficiently with an RNNCell, it can still struggle when dealing with long sequences, a common issue with standard RNNs. To mitigate this, the RNNCell is replaced by a GRUCell, making an ODE-GRU architecture.

In the same manner as the GRU network, to get output at time t the hidden state and input at time $t-1$ were used. This way, the update using the GRUCell is as follows $h_i = GRUCell(h'_i, x_{i-1})$. With this, the following algorithm outlines the training algorithm employed for the ODE-GRU model.

Algorithm 5 Training of the ODE-GRU

Input: Data points $\{(x_i, t_i)\}_{i=0,\dots,N}$
Initialize:
    $\theta_0 \leftarrow$ Initialize parameters # initialize model parameters
    Optimizer $\leftarrow$ Adam(θ) # choose optimizer for minimising $L(\theta)$
for n = 1,..., epochs **do** # epochs specifies the number of training iterations
    $h_0 = \mathbf{0}$ # initial hidden state, initialized to zero
    for i = 1,...,N **do**
        $h'_i = ODESolve(f_\theta, h_{i-1}, [t_{i-1}, t_i])$ # get solution at time t_i
        $h_i = GRUCell(h'_i, x_{i-1})$
        $o_i = LinearLayer(h_i)$
    end for
    Evaluate loss function $L(\theta_{n-1}; \{o_i\}_{i=1,\dots,N}, \{x_i\}_{i=1,\dots,N})$ # compute loss for all predictions
    $\nabla L \leftarrow$ Compute gradient $\nabla L(\theta_{n-1}; \{o_i\}_{i=1,\dots,N}, \{x_i\}_{i=1,\dots,N})$ #backpropagation
    $\theta_n \leftarrow$ Optimizer.Step($\theta_{n-1}, \nabla L$) # update model parameters
end for
return $\{o_i\}_{i=1,\dots,N}$; h_N

The prediction process for the ODE-GRU model is similar to that of the GRU. It requires the last hidden state and the last input from the training data, and the predictions are generated recursively for subsequent time steps. This can be seen in the following algorithm.

Algorithm 6 ODE-GRU Forecasting

Input: Set of test points $\{(x_i, t_i)\}_{i=N+1,\ldots,M}$

Last hidden state h_N and last data point from training set x_N

$h'_{N+1} = ODESolve(f_\theta, h_N, [t_N, t_{N+1}])$

$h_{N+1} = GRUCell(h'_{N+1}, x_N)$

$o_{N+1} = LinearLayer(h_{N+1})$

for i = N+1,...,M **do**

    $h'_i = ODESolve(f_\theta, h_{i-1}, [t_{i-1}, t_i])$ # get solution at time t_i

    $h_i = GRUCell(h'_i, x_{i-1})$

    $o_i = LinearLayer(h_i)$

end for

return $\{o_i\}_{i=N+1,\ldots,M}$

4 Methodology and Dataset

4.1 Programming Language

The models outlined in this dissertation were developed and executed using *Python*, chosen for its versatility, simplicity and wide range of specialized libraries. *Python* is extensively adopted in data science and machine learning because it has the ability to efficiently integrate various tools and frameworks, offering a solid environment for data analysis and manipulation.

For the analysis, visualization and pre-processing of the data the following libraries were used:

- **Pandas**: Data analysis and manipulation tool, that can be used for reading and writing data, reshaping, aggregating or transform data, etc [35];
- **NumPy**: Open source Python library used to deal with multidimensional array data structures [36];
- **Collections**: Module defined as a container that is used to store collections of data. It provides specialized containers like Counter which is used for counting occurences of elements in a dataset [37];
- **Math**: Module that provides acess to mathematical functions [38];
- **Matplotlib**: Comprehensive library for creating interactive visualizations of data in Python [39];
- **Sklearn.preprocessing**: Package that provides several utility functions used for pre-processing and transforme data [39].

For the construction of the models, the following tools were used:

- **Pytorch**: Open source machine learning framework based on the Python programming language and the torch library and it is used to create deep NNs [40];
- **Torchdiffeq**: Library that provides ODE solvers implemented in Pytorch [41].

4.2 Dataset

The dataset used for this project is a comprehensive financial dataset designed to examine different factors affecting cash flow.

The dataset comprises of 5099 rows of a total of 1475 different Group Id's, each group id representing a different company. Each row in the dataset represents an observation for a specific group id, and for each group id, along with the corresponding date, we have information about the cashflow (driven variable) and 6 additional features (drivers), which influence the cashflow. The features are as follows:

- **Cashflow**: is the time series we want to predict, indicates the net change, i.e, the difference between cash inflows and outflows, over a specific period of time, in this case, the period between the first day of each month, as represented by the feature **Timestamp**;
- **SV** (Sales Volume): refers to the sales value during a specific reporting period, in this case a month;
- **DSO** (Days Sales Outstanding): is the average number of days that it takes for a company to collect payment after a sale has been made;
- **CMVMC** (*Custo das Mercadorias Vendidas e das Matérias Consumidas*) (in english, cost of goods sold and materials consumed): represents the direct costs of producing goods sold by a company, including materials and labor directly involved in production;
- **FSE** (*Fornecimentos e Serviços Externos*) (in english, external supplies and services): is the total expenses associated with services such as power and fluids, materials, sucontracts, etc;
- **COP** (Cost of Personnel): is the expenses associated with employees;
- **DPO** (Days of Payable Outstanding): is the average of days that a company takes to pay its bills and invoices to its suppliers.

In the dataset, some features are closely related to each other. These connections reflect the DK, as they represent underlying dependencies that are inherent to the problem being studied. Incorporating this DK into the model may help improve its predictive accuracy by leveraging these meaningful relationships between the different features.

The relationship between SV and DSO is significant because DSO indicates the average time it takes for the cash flow to reflect the value of SV as an inflow. A higher DSO means that it takes too long for the company to collect payments, which can reduce the cashflow while a low DSO means that the company is collecting payments quickly, which can boost the cash flow. Similarly, FSE and DPO are related as DPO indicates the average time it takes for the value of FSE to be reflected in the cash flow as an outflow. A higher DPO means the company can retain cash longer, which can improve the cash flow while a low DPO means that the company is paying its bills too soon, which can reduce the cash flow.

Dataset analysis

To ensure the reliability of the forecasting model, it is essential to assess data availability for each group ID (company). By analyzing the number of observations per group ID, we can determine if there is sufficient data to train an NN architecture effectively.

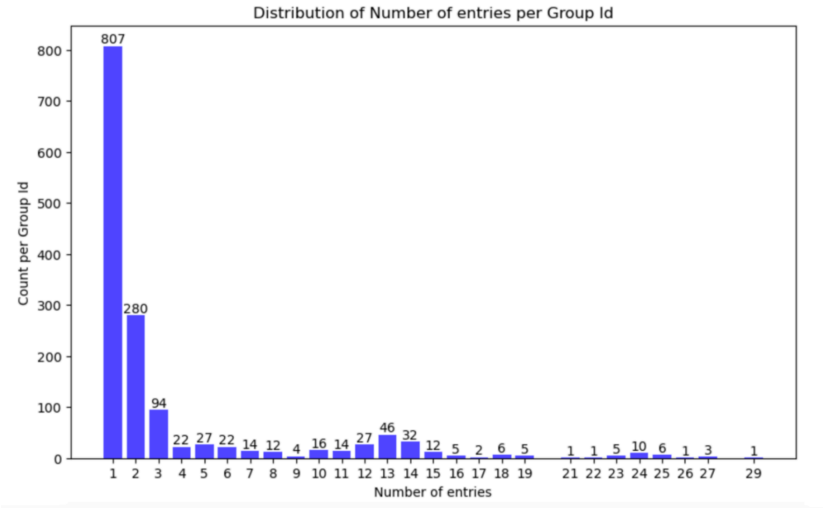


Figure 5: Distribution of number of observations per Group Id

As shown in Figure (5), out of a total of 1475 Group Ids, 807 have only a single observation, and more than half have less than 10 observations. This limited data availability poses a significant challenge for building a reliable predictive model, as NNs generally require a sufficient number of observations to accurately capture underlying patterns and relationships. The risk of overfitting increases when dealing

with very few observations, making it difficult for the model to generalize effectively. In this context, the integration of DK can be particularly beneficial.

Given that a large portion of the group IDs had very few observations, the focus was placed only on those with at least 10 observations. This selection consisted of 2865 observations and 193 different group IDs. By selecting group IDs with at least 10 records, it is ensured that there are a considerable number of data points per group ID to train a more reliable forecasting model.

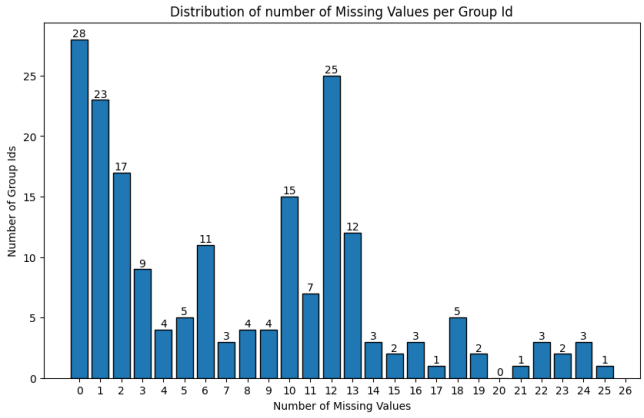
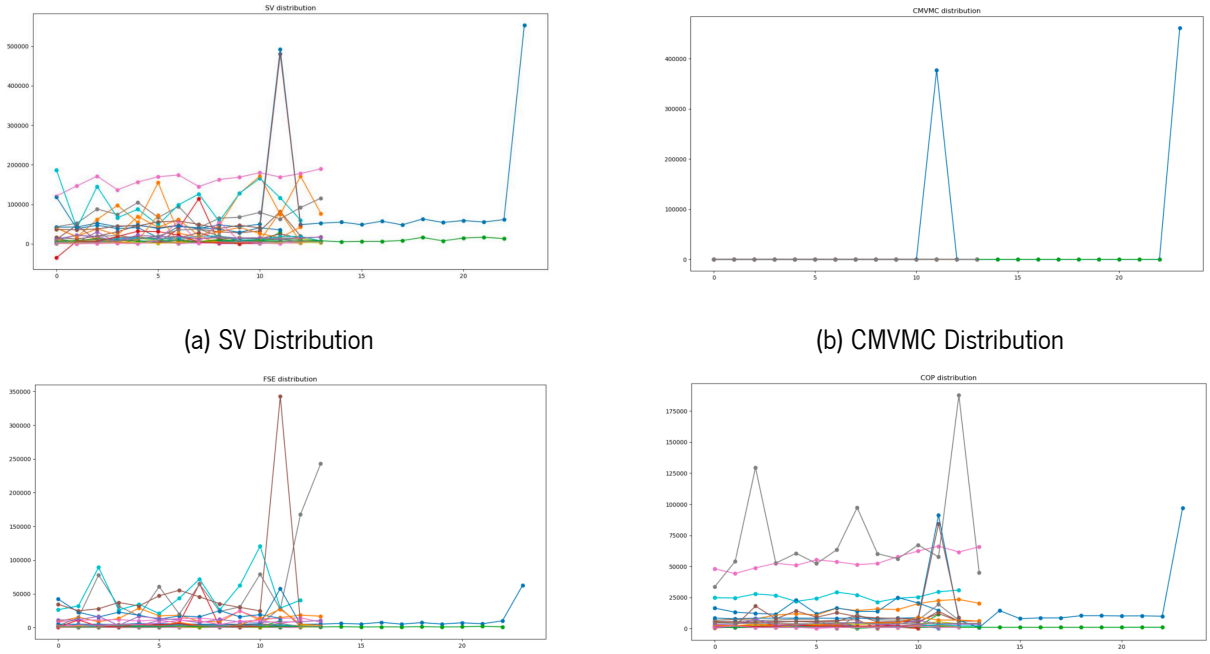


Figure 6: Distribution of missing values per Group Id

Following this, an analysis was performed to check for missing values and assess the completeness of the selected data. The analysis indicated that most of the group IDs contained missing values, only 28 of 193 group IDs contained no missing values, as shown in Figure (6).

Initially, to address the issue of limited observations per group ID, the idea was to train a model using all available group IDs. Following this approach, an analysis was conducted on the group IDs with at least 10 observations and no missing values to compare the dynamics of the features: SV, CMVMC, FSE and COP, that influence the cashflow.



(a) SV Distribution

(b) CMVMC Distribution

(c) FSE Distribution

(d) COP Distribution

Figure 7: Features distribution

The results of this analysis, as shown in Figure (7), revealed that the features dynamics varied significantly among the different group IDs, suggesting that it would not be beneficial to train a model using all group IDs together, as the varying dynamics could lead to a sub-optimal performance.

4.3 DK Integration

The features DSO and DPO indicate the average number of days a company takes to receive payments or make payments, respectively. Since these values represent averages, there is inherent uncertainty. It's not possible to state with certainty that a company will receive or pay in exactly x days, as the number reflects an average, not an exact number.

The DK carries a degree of uncertainty, posing a challege when converting it into explict and precise information that a forecasting model can use. While this knowledge is valuable, it is not easily translatable into concrete terms that a model can directly process. Therefore, instead of implementing any of the studied methods, a more manual approach for knowledge integration was taken. This involved designing a

new feature that encapsulates the DK.

The new feature will be referred to as Cashflow Adjusted. In its creation, two auxiliary features were first created: SV-DSO target date and FSE-DPO target date:

- SV-DSO target date: When the DSO value, at a certain timestamp i, is different than zero, we can conclude that the SV at timestamp i may not influence the cashflow at the same timestamp. Following this, using the value in DSO, a target date is calculated, representing the date that is expected for the SV value to impact the cashflow. For example, let's consider that at timestamp 01-01-2024, the DSO is 30, then the target date is 31-01-2024;

- FSE-DPO target date: The process used before was used to create this feature, but using FSE and DPO.

Since DS0 only indicates an average timeframe for when SV affects cashflow, there is uncertainty about whether the full SV amount will impact cashflow on the target date. To adress this, 90% of the SV value is assumed to be accounted for the target date, with the remaining 10% in the following month. This same logic was applied using FSE and DPO.

Since SV represents inflows and FSE represents outflows, taking their respective target dates into account, the SV value is removed and the FSE value is added to the cashflow creating the feature Cashflow_Adj (Cashflow Adjusted). This new feature allows us to retain the overall cashflow value while accounting for the events represented by SV and FSE. Therefore, for forecasting, we keep the features Cashflow, CMVMC, COP and Cashflow_Adj.

Furthermore, it is important to note that a cashflow forecast can be obtained from the Cashflow_Adj forecast by reversing the procedure just described, i.e., by adding the contributions of SV and subtracting those of FSE from the Cashflow_Adj.

4.4 Data Preprocessing

Given the large number of group IDs available, only two group IDs were selected for model training. To ensure a sufficient amount of data for training, the group IDs choosen had at least 10 observations. The group Id used will be refered to as G64 and G276; G64 has 15 observations while G276 has 25 observations.

The first step in the data preprocessing was identifying and estimating missing values. Given the limited number of observations, to address the missing values, a linear interpolation approach was used, as it provides a straightforward method to estimate values without introducing unnecessary complexity.

Secondly, any features that were deemed unnecessary for training were removed from the dataset. For the datasets without DK the remaning features were Cashflow, SV, CMVMC, FSE and COP. DSO and DPO represent days rather than actual cash transactions; hence, they do not have a direct influence on cash flow and were consequently excluded. For the datasets with DK the remaning features were Cashflow, CMVMC, COP and Cashflow_Adj. In same cases the variable CMVMC contained only zero values and since it does not provide any meaningful information this feature was also removed.

Subsequently, the data was scaled to the range $[0, 1]$ using Min-Max normalization to ensure that all input variables are on a similar scale.

A neural ODE requires a time variable to represent the evolution of time, as the model relies on continuous dynamics to predict how the systems evolves over time. In the dataset, since each observation corresponds to a specific month, a time variable was created to reflect these monthly progressions. For each subset, the first observation represents $t = 1$ and any other observation represents $t = 1 + l$ where l is the difference, in months, between the current observation and the previous observation.

Scaling the time variable is crucial when training a neural ODE because the *odeint* solver, utilized in subsequent experiments, works better with smaller step sizes, which helps improve the stability of the solution and reduce the propagation of errors during the integration process. By scaling, we enable the solver to function more efficiently, leading to better training results. With this, the time variable was also scaled in the range $[0, 1]$ using Min-Max normalization.

At last, every subset was split into two parts: one retained the last data point for testing, while the rest were used for training.

To analyze the impact of DK integration, all the following experiments were conducted on two different datasets, the dataset with DK, i.e, the dataset containing the features Cashflow, CMVMC, COP and Cashflow_Adj, and the dataset with no DK, i.e, the dataset containing the features Cashflow, SV, CMVMC (in the case where this feature was not removed), FSE and COP.

5 Experiments and Results Analysis

In this chapter, the experiments and analysis of the results are presented. Subsection 1 presents the data augmentation strategy implemented, and subsections 2, 3, and 4 present the experiments with the architectures studied: neural ODE, GRU, and ODE-GRU.

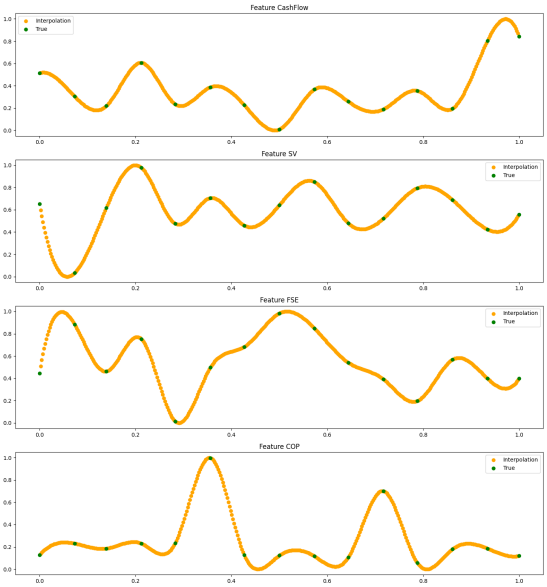
5.1 Transformation of monthly data to daily data

All experiments were initially conducted using the original dataset, which consisted of monthly data. However, due to the limited amount of data available, data augmentation was performed in an attempt to improve the performance of each model. The goal of this augmentation was to generate additional data points to enhance model generalization and potentially increase accuracy.

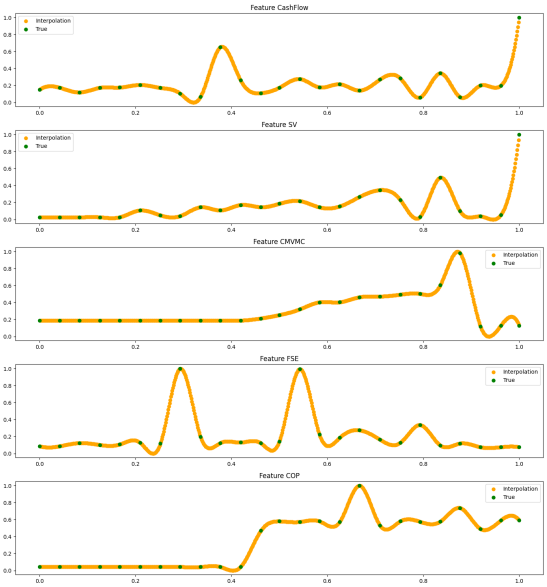
To generate additional data points, cubic spline interpolation was used on each feature in the dataset. Cubic spline [42] interpolation approximates values between established data points by fitting a piecewise cubic polynomial to the data. The polynomial is chosen such that it is twice continuously differentiable, meaning that its first and second derivatives are continuous across the data intervals. This approach guarantees that the interpolated values seamlessly align with the overall data pattern, thereby minimizing abrupt changes or oscillations.

The known values were taken from the original dataset, and the interpolation was applied to estimate the values at the missing daily time steps. This process generated a new set of daily data points, which were then used for model training and evaluation. This process was conducted on both datasets, with and without DK.

Figures (8) and (9) show the final results of transforming monthly data into daily data. Each graph represents the features in both datasets: without DK (8) and with DK (9), for the group IDs G64 and G276. The yellow points in each figure represent the data points created using cubic spline interpolation, based on the true points of the dataset, shown as green points in each figure.



(a) G64

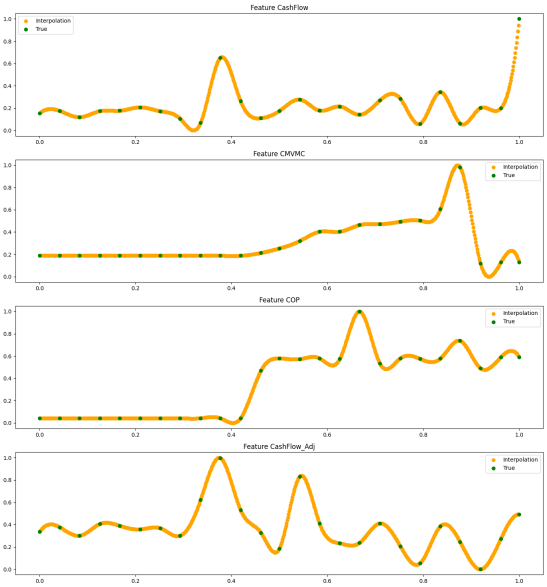


(b) G276

Figure 8: Visualization of the transformation from monthly to daily data with dataset without DK



(a) G64



(b) G276

Figure 9: Visualization of the transformation from monthly to daily data with dataset with DK

In the daily dataset, two experiments were conducted to evaluate the model's ability to predict future values. In the first experiment, the last 15 data points were reserved for testing, while in the second, only the last 7 data points were reserved. This adjustment was made after noticing that predicting 15 data points, i.e, predicting the next 15 days, was overly challenging for the model, given the limited amount of information available in the dataset.

5.2 Neural ODE Training and Results

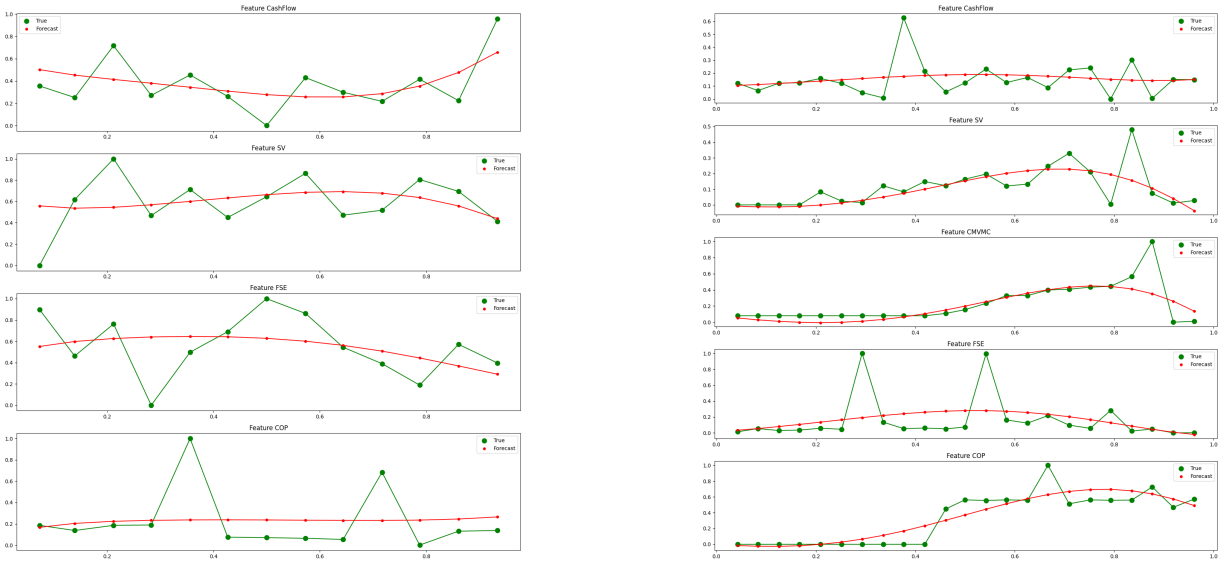
In the initial phase of the analysis, a simple neural ODE was implemented to model the system's dynamics. The neural ODE model used in these experiments is based on an existing implementation [43]. The architecture consisted of two fully connected layers; one hidden layer with 50 neurons and an output layer, which the number of neurons corresponds to the number of features used for prediction. As explained in the previous section, in cases where the feature CMVMC only had zero values, it was removed. In the dataset without DK, 4 features remained for prediction, Cashflow, SV, FSE and COP, this was the case of the group ID G64, when CMVMC was not removed, 5 features were used for prediction, this was the case of group ID G276. The same was applied to the dataset with DK, for group ID G64 3 features remained for prediction, Cashflow, COP and Cashflow_Adj, and for group ID G276 remained 4 features, since CMVMC was included.

Between the layers the activation function ELU [44] was applied to introduce non-linearity. While our primary focus is forecasting the cash flow, we also want to assess whether the model is capable of learning the dynamics of all the features. Therefore, the number of neurons in the output layer corresponds to the same number of features in each observation.

Each group Id was trained for 2000 epochs using the Adam optimizer, a learning rate of `1e-3` and the loss function was the Mean Squared Error(MSE), which quantifies the average squared difference between the predicted and actual values. The Neural ODE was solved using the odeint function from the torchdiffeq library. The solver uses by default the 4th-order Runge-Kutta method with an adaptive step size. This approach automatically adjusts the step size during integration.

Monthly Data

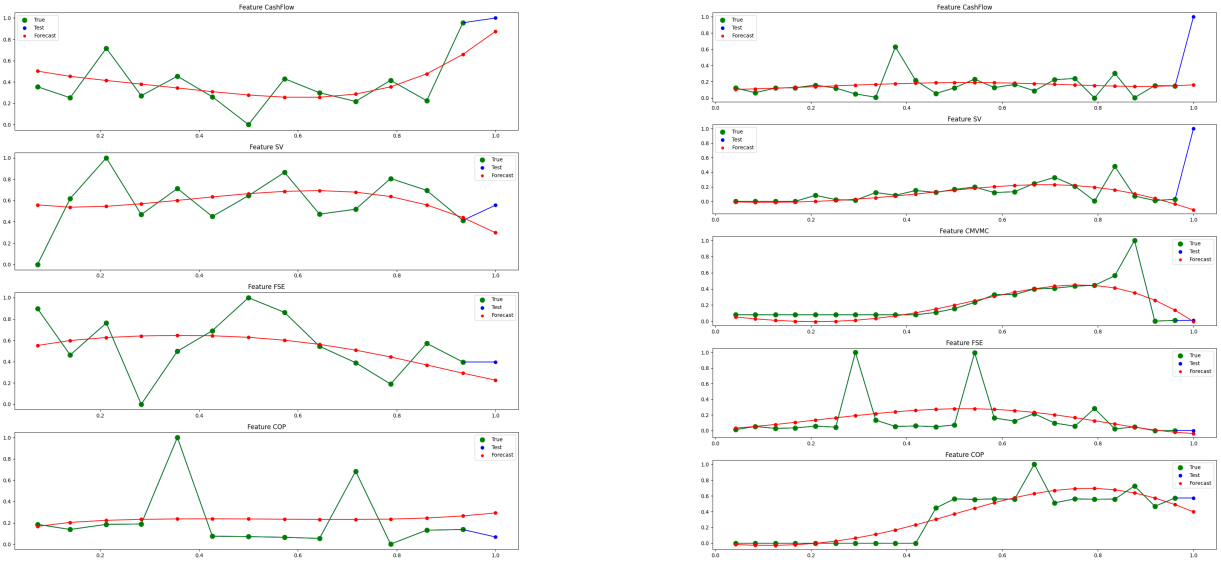
The results presented in Figure (10) evaluate the performance of the neural ODE model in capturing the dynamics of the variables during training,for both group Ids, when using the dataset without DK. With these results, we concluded that the model failed to represent the underlying dynamics adequately.



(a) G64 (b) G276

Figure 10: Neural ODE training results for group IDs G64 and G276 - no DK

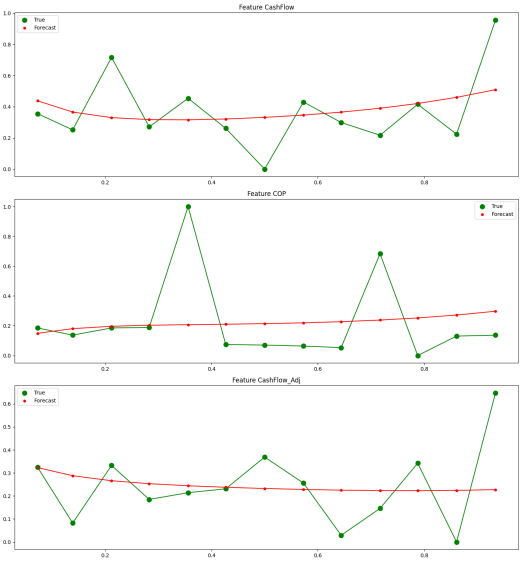
To further evaluate the model's performance, the Neural ODE was tested on unseen data. The results presented in (11) show the model's predictions. The findings reveal that the predictions did not align closely with the actual values.



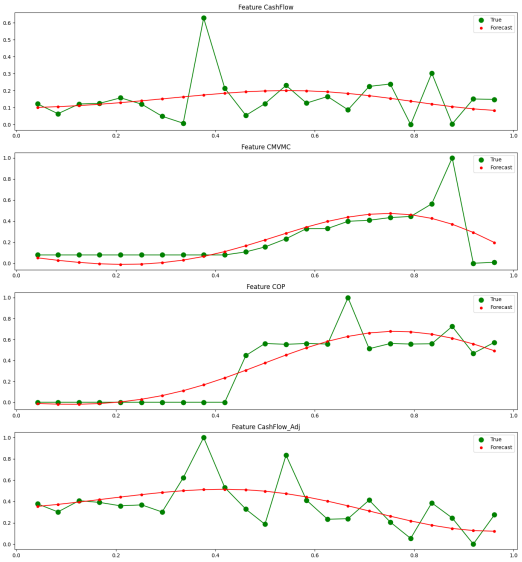
(a) G64 (b) G276

Figure 11: Neural ODE testing results for group IDs G64 and G276 - no DK

The Neural ODE was also unable to learn the data dynamics in the dataset with DK, as it can be seen in Figure (12), as well as it was not capable of generalizing to unseen data as shown in Figure (13). This suggests that the model may not be well-suited for the given data, as it struggled to capture the underlying patterns regardless of the inclusion of DK. With these results it is not possible to conclude whether or not the integration of DK is beneficial in this case. It is likely that a different model might be better suited to capture the dynamics of the data.

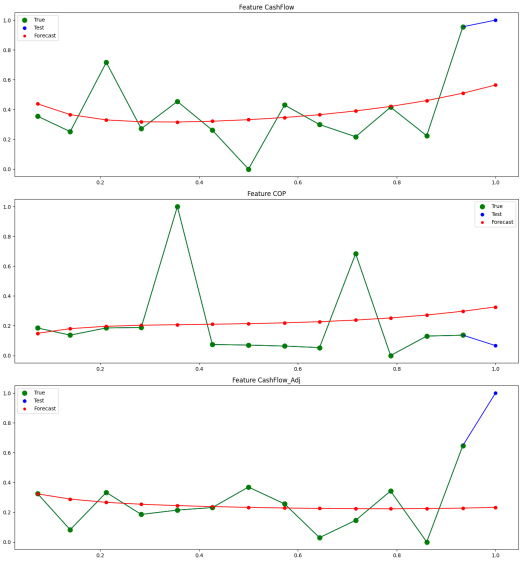


(a) G64

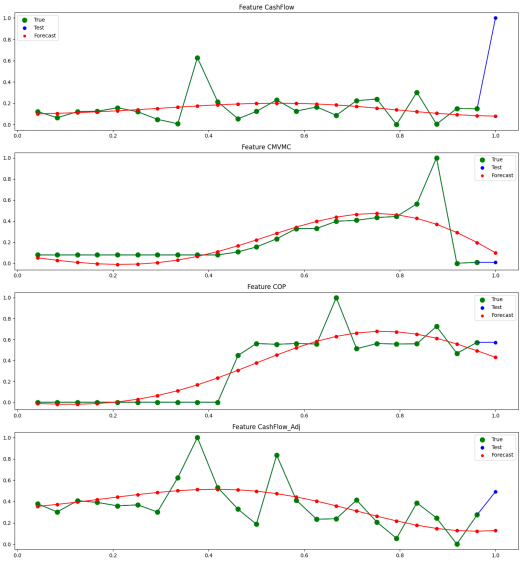


(b) G276

Figure 12: Neural ODE training results - with DK



(a) G64

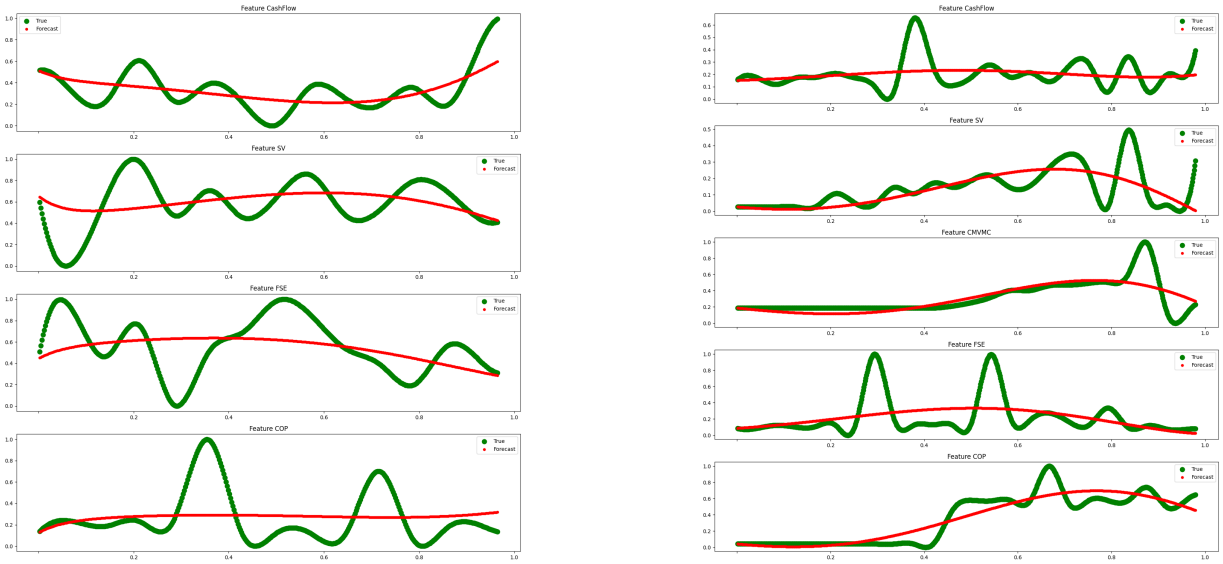


(b) G276

Figure 13: Neural ODE testing results - with DK

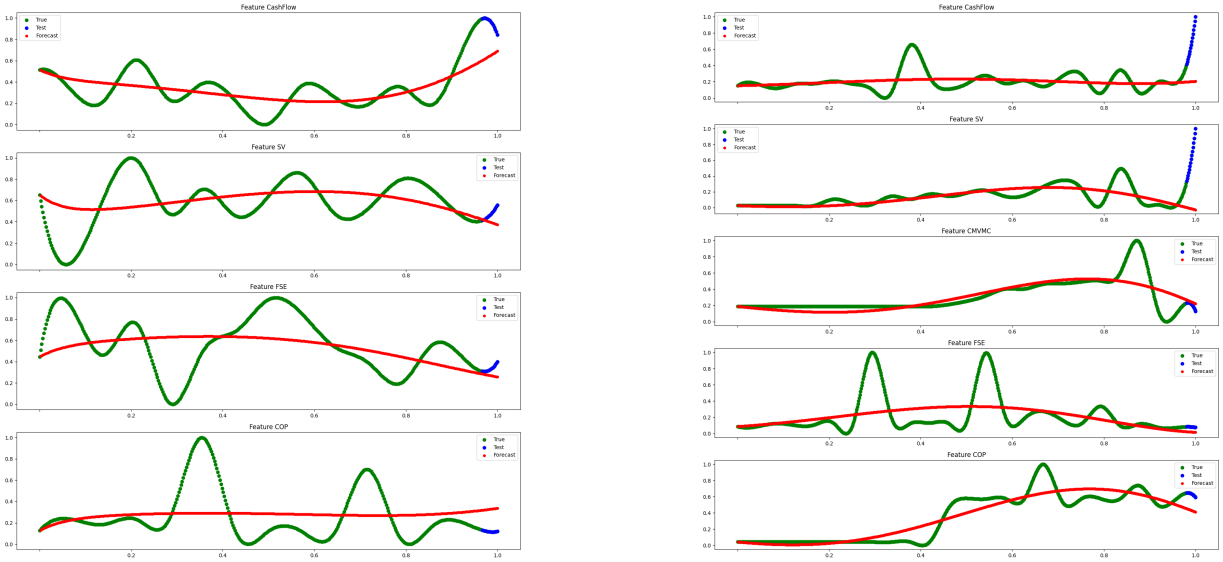
Daily Data - 15 days prediction

In the experiment using the daily dataset with 15 days allocated for testing, the Neural ODE model was unable to learn the dynamics of the data effectively, as in can be seen in Figures (14) and (16). Transitioning to a larger daily dataset, which provided more data points and higher granularity, did not result in significant improvement. The model continued to struggle in capturing the temporal patterns effectively. As expected, since the model could not learn well during training, it also struggled to make accurate predictions on unseen data, as seen in Figures (15) and (17).



(a) G64 (b) G276

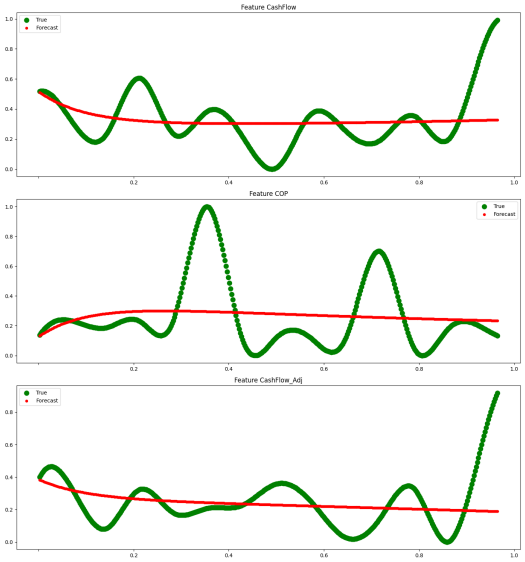
Figure 14: Neural ODE training results for group IDs G64 and G276 - no DK



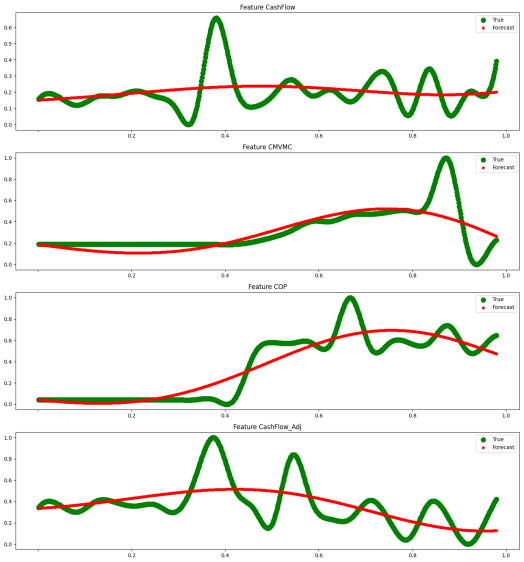
(a) G64 (b) G276

Figure 15: Neural ODE testing results for group IDs G64 and G276 - no DK

The model also failed to learn the data dynamics effectively, regardless of the presence of DK. Once again, while the integration of DK did not result in improved performance, reinforces the idea that a different modeling approach may be required for this particular dataset.

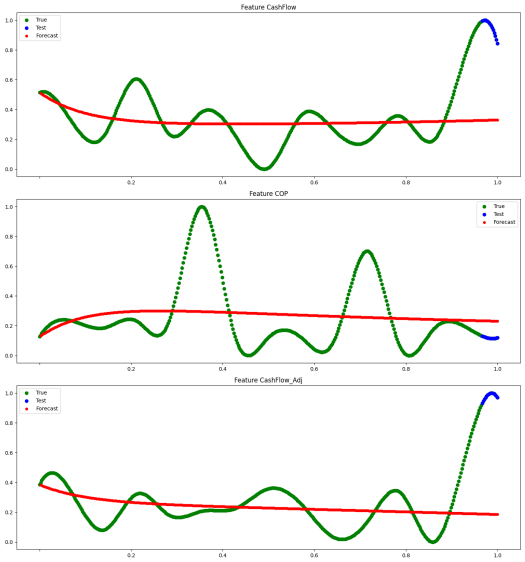


(a) G64

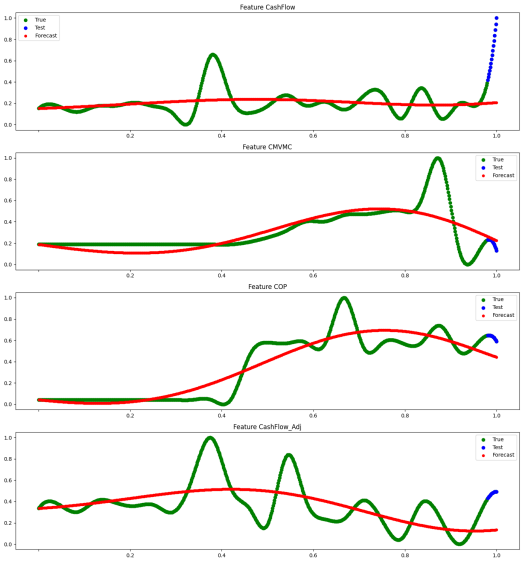


(b) G276

Figure 16: Neural ODE training results - with DK



(a) G64



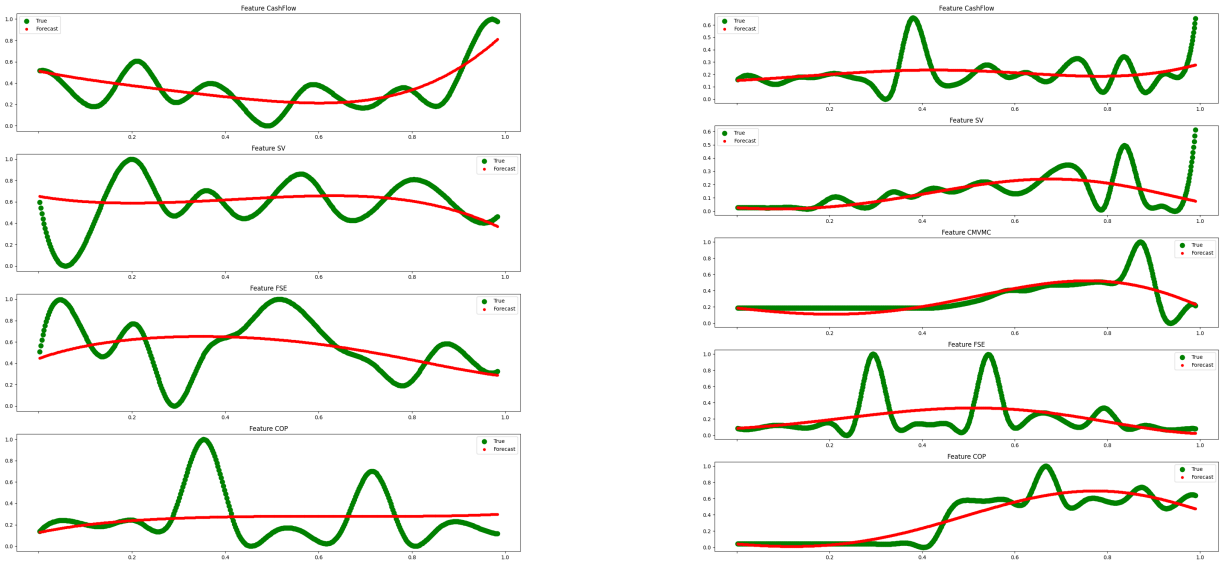
(b) G276

Figure 17: Neural ODE testing results - with DK

Daily Data - 7 days prediction

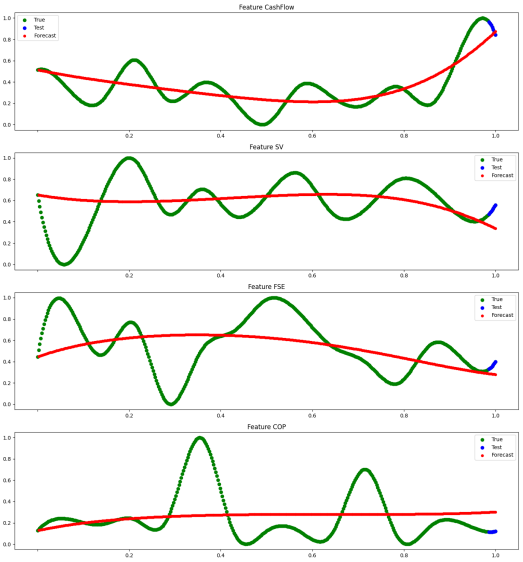
Similarly, in the experiment with the daily dataset and 7 days allocated for testing, the Neural ODE model still faced challenges in learning the data dynamics. It still couldn't capture the necessary patterns in the training data to make accurate predictions, as can be seen in Figures (18) and (20). As expected, this led to poor generalization on the unseen data, as seen in Figures (19) and (21).

The findings were aligned with those from past experience for both the datasets, with and without DK, as the training dataset remains largely unchanged.

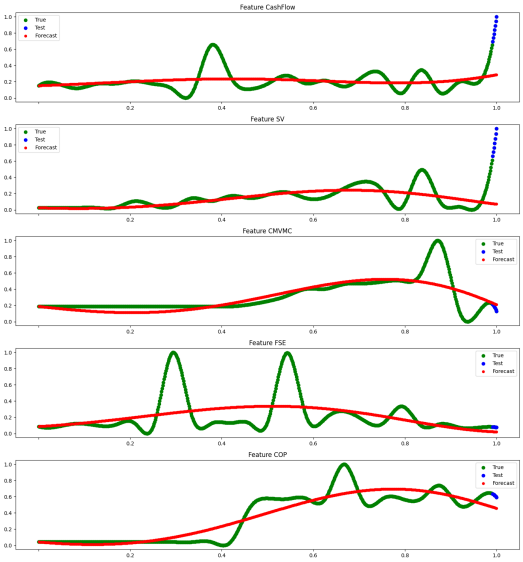


(a) G64 (b) G276

Figure 18: Neural ODE training results for group IDs G64 and G276 - no DK



(a) G64

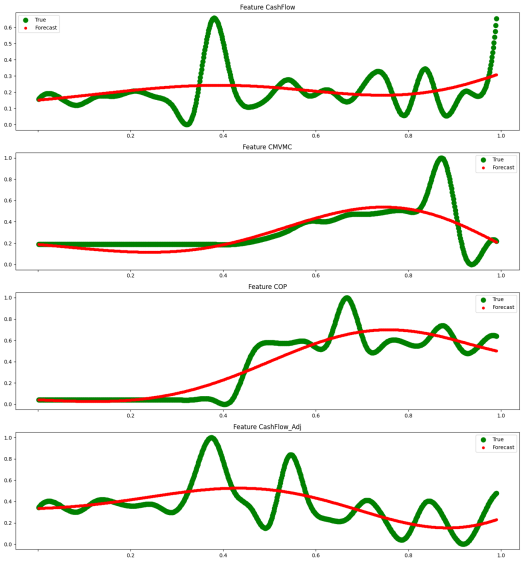


(b) G276

Figure 19: Neural ODE testing results for group IDs G64 and G276 - no DK

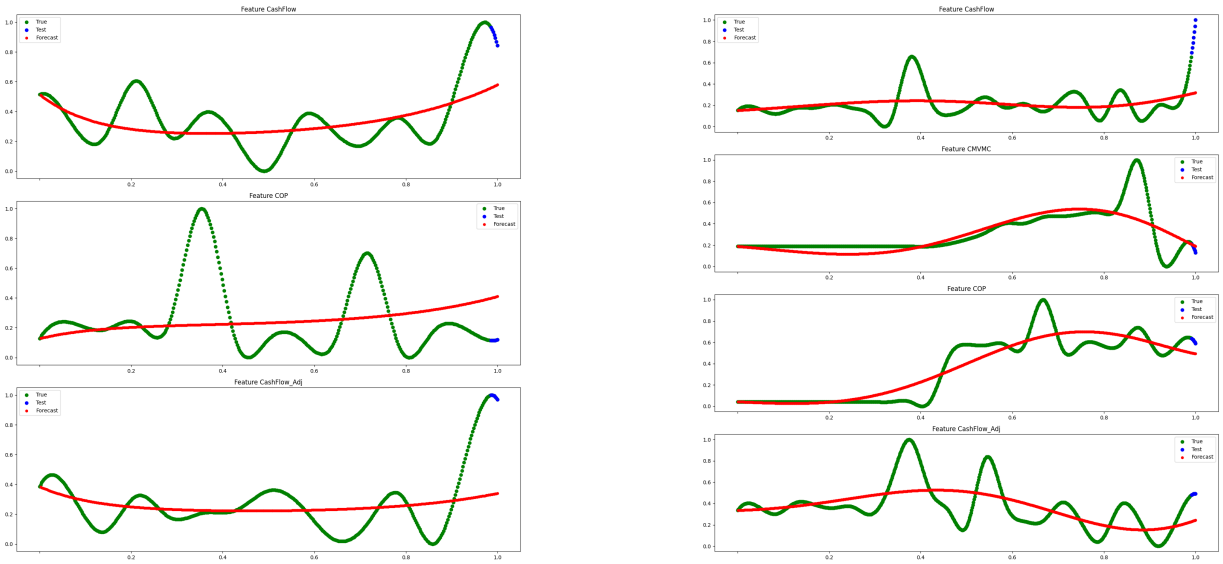


(a) G64



(b) G276

Figure 20: Neural ODE training results - with DK



(a) G64

(b) G276

Figure 21: Neural ODE testing results - with DK

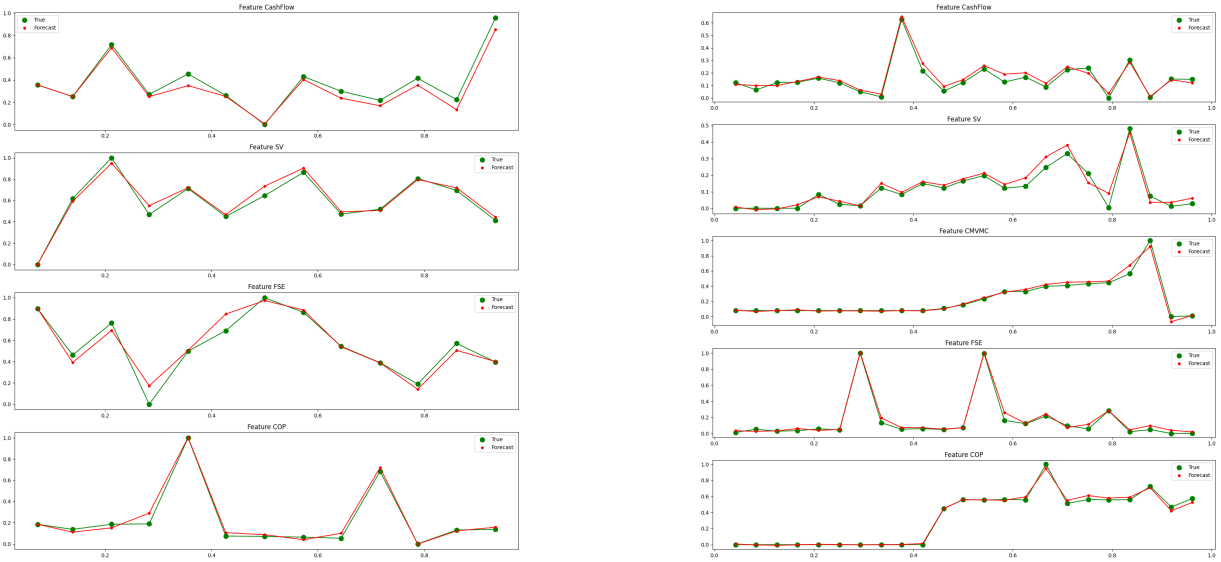
5.3 GRU Training and Results

The architecture consisted of two fully connected layers; one hidden layer with 50 neurons and an output layer, of which the number of neurons corresponds to the number of features used for prediction. Each group Id was trained for 2000 epochs using the Adam optimizer, a learning rate of `1e-3` and the loss function was the Mean Squared Error(MSE). All the following experiments were carried out under the same conditions as the previous experiments.

Monthly Data

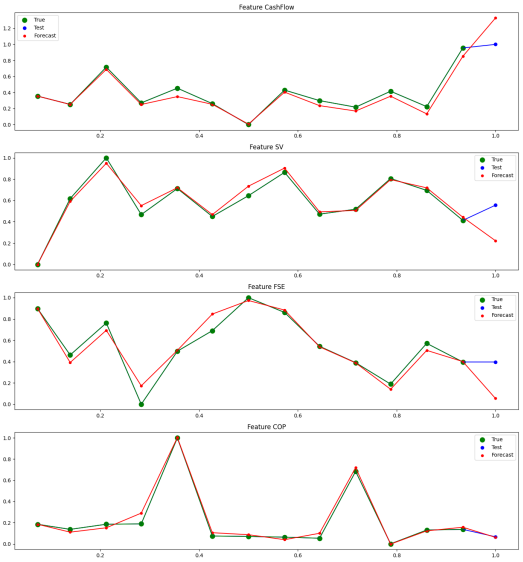
The GRU model, compared to the Neural ODE model, demonstrated to be better at learning the dynamics of the training dataset, achieving reasonably good predictions on the observed data. The results can be seen in Figure (22), for the dataset without DK, and in Figure (24), for the dataset with DK. However, when tested on unseen data, the performance significantly declined, highlighting the limitations of the model in generalizing beyond the training set, as it can be observed in Figures (23) and (25).The limited amount of

training data likely caused the GRU to overly rely on specific details in the training set instead of learning more robust general patterns, leading to overfitting. Indeed, when considering both group Ids, we possess data spanning a maximum of 2 years, with details only available for each month. In such scenarios, having a historical dataset extending over more years is essential for the models to discern meaningful patterns.

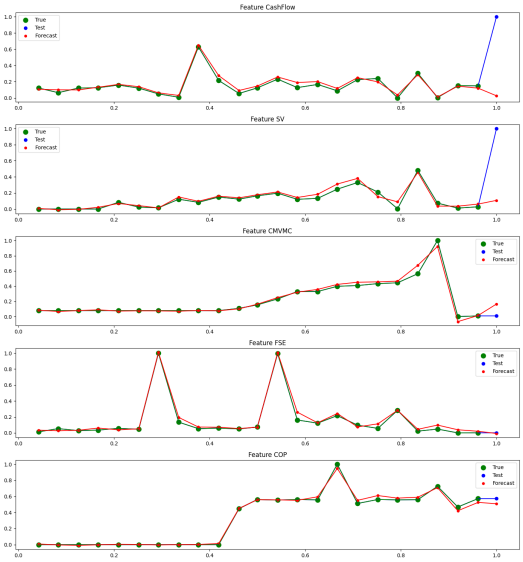


(a) G64 (b) G276

Figure 22: GRU training results for group IDs G64 and G276 - no DK



(a) G64

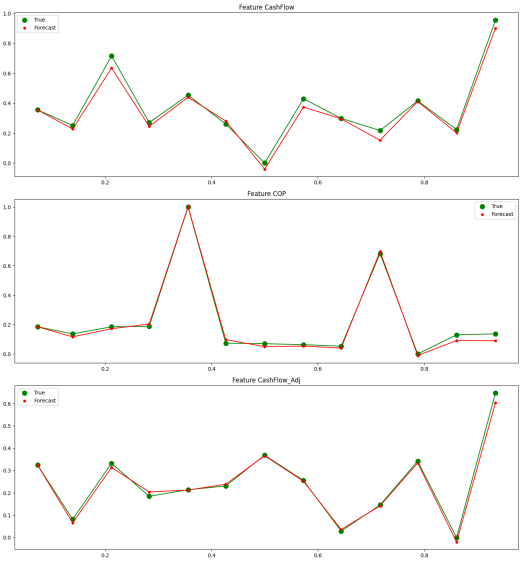


(b) G276

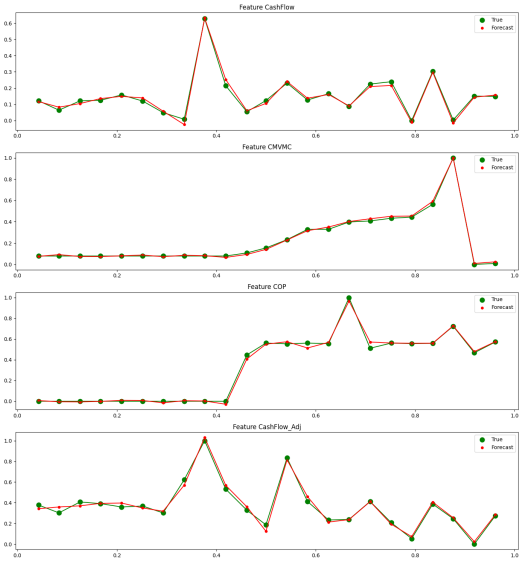
Figure 23: GRU testing results for group ids G64 and G276 - no DK

For both datasets with and without DK, the model was able to predict the training data. However, no significant improvements were observed when DK was integrated. Similarly, when attempting to generalize to unseen data, the model was unable to make accurate predictions, indicating that the integration of knowledge did not contribute to improved performance.

Also, for both group IDs, the predictions for Cashflow_Adj seem to yield better results compared to the predictions for cashflow without DK.

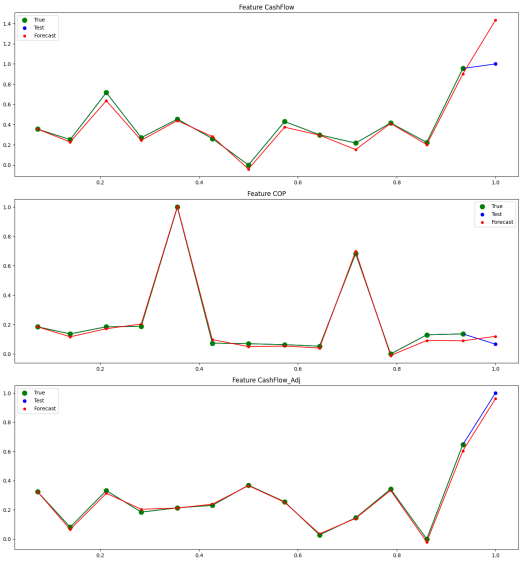


(a) G64

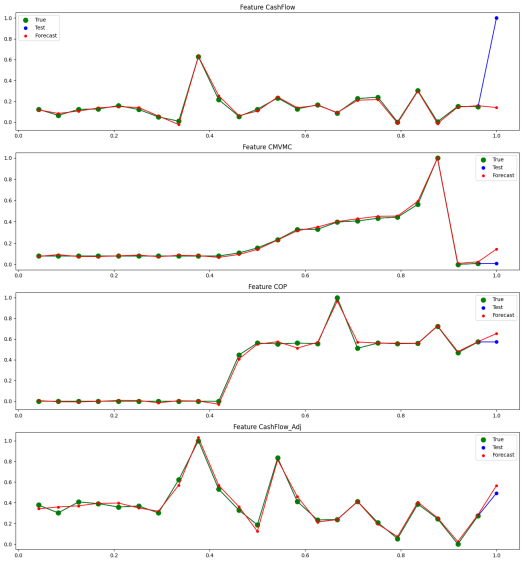


(b) G276

Figure 24: GRU training results - with DK



(a) G64



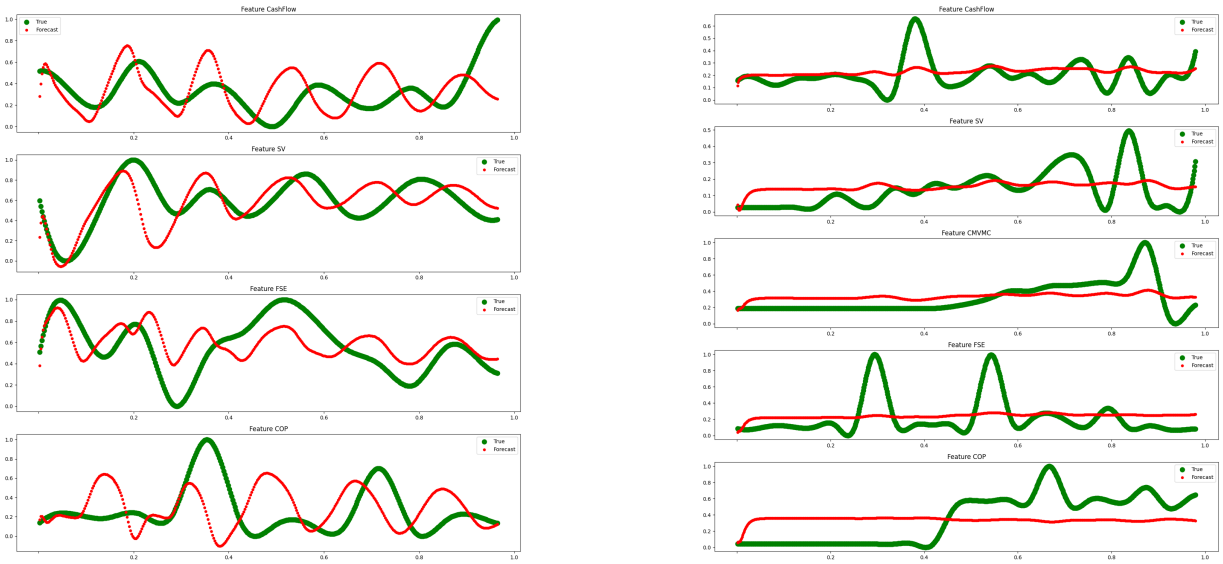
(b) G276

Figure 25: GRU testing results - with DK

Daily Data - 15 days prediction

When training the GRU on the daily dataset, which is significantly larger compared to the original monthly dataset, the model struggled to make accurate predictions even during training. This could be attributed to the increased error propagation over the extended sequence of daily data. With a larger number of time steps, small errors in earlier predictions may accumulate, making it challenging for the GRU to capture and learn the underlying dynamics of the data effectively. Consequently, the model failed to achieve convergence or learn meaningful patterns from the the dataset as can be see in Figures (26) and (28).

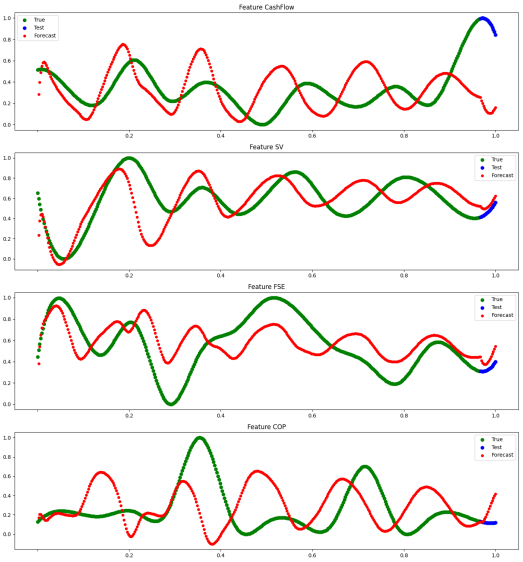
As expected, given the difficulty of the model in learning the dynamics of the training set, it also failed to produce accurate predictions on the test set, highlighting its inability to generalize to new data, as can be seen in Figures (27) and (29).



(a) G64

(b) G276

Figure 26: GRU training results for group IDs G64 and G276 - no DK

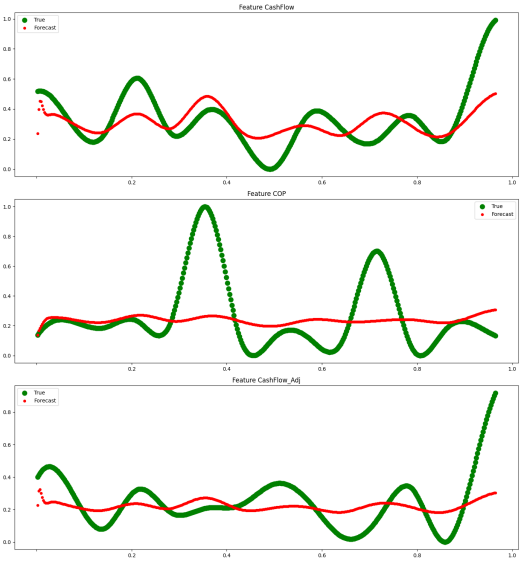


(a) G64



(b) G276

Figure 27: GRU testing results for group IDs G64 and G276 - no DK

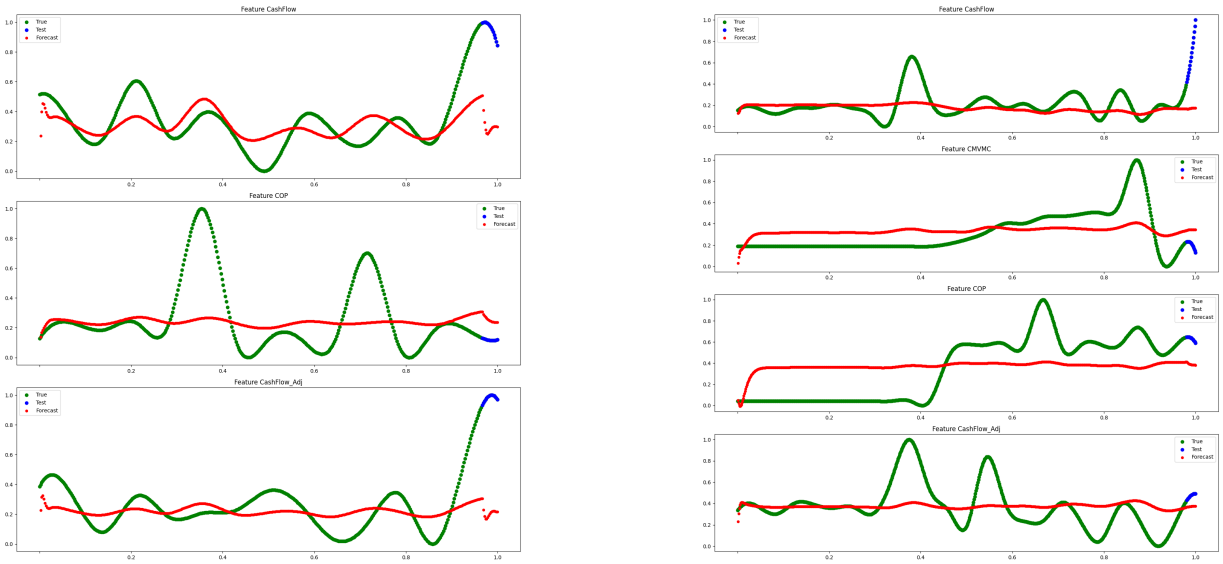


(a) G64



(b) G276

Figure 28: GRU training results - with DK



(a) G64

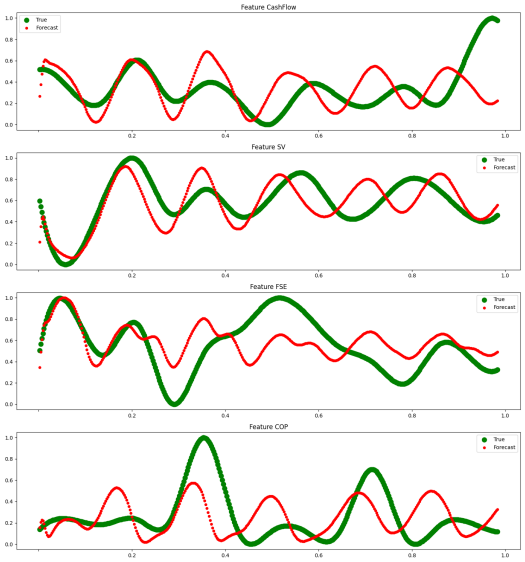
(b) G276

Figure 29: GRU testing results - with DK

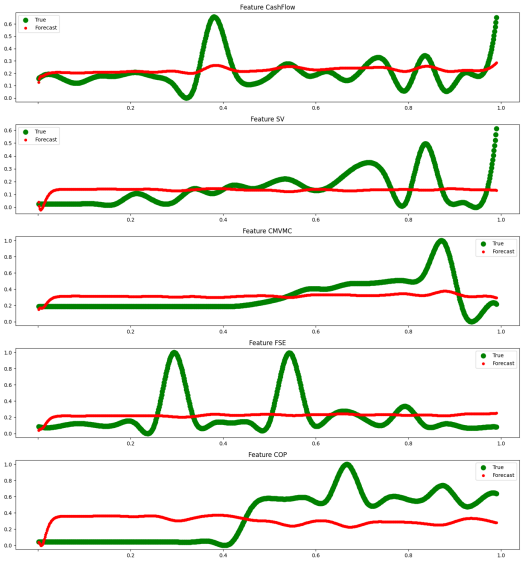
Daily Data - 7 days prediction

This time only 7 days were held out for testing, the results were consistent with the previous experiment. The model struggled to learn the underlying patterns and as expected failed to generalize well to the unseen data. This outcome is not surprising, as the model had already shown difficulty in effectively learning from the training data when only 15 days were left for testing.

The training results can be seen in Figures (30) and (32), and the test results in Figures (31) and (33).

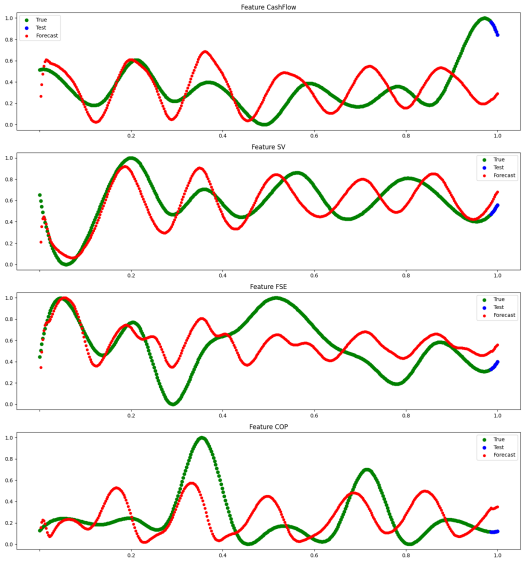


(a) G64



(b) G276

Figure 30: GRU training results for group IDs G64 and G276 - no DK



(a) G64

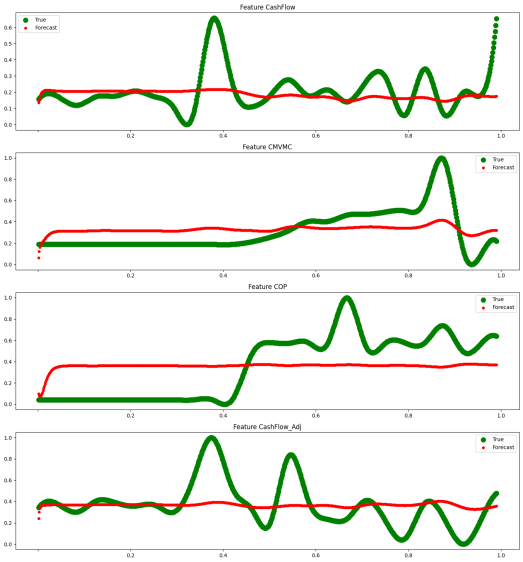


(b) G276

Figure 31: GRU testing results for group IDs G64 and G276 - no DK

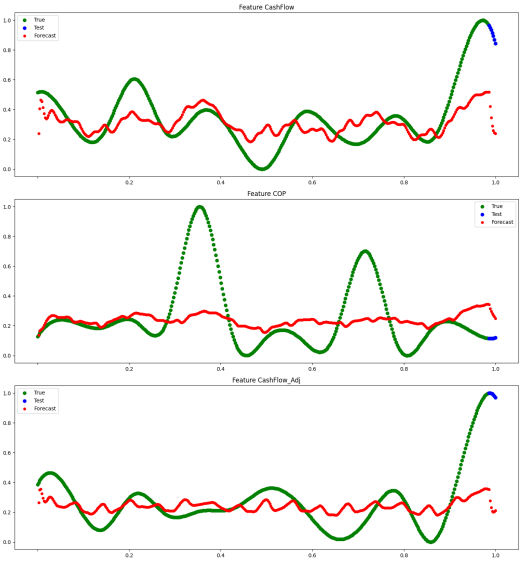


(a) G64

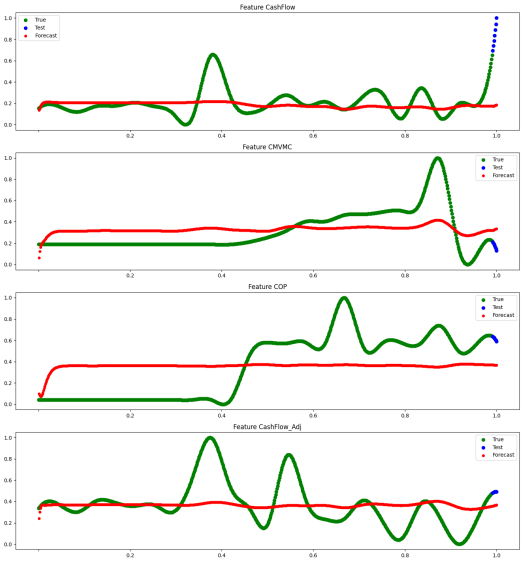


(b) G276

Figure 32: GRU training results - with DK



(a) G64



(b) G276

Figure 33: GRU testing results - with DK

5.4 ODE-GRU Training and Results

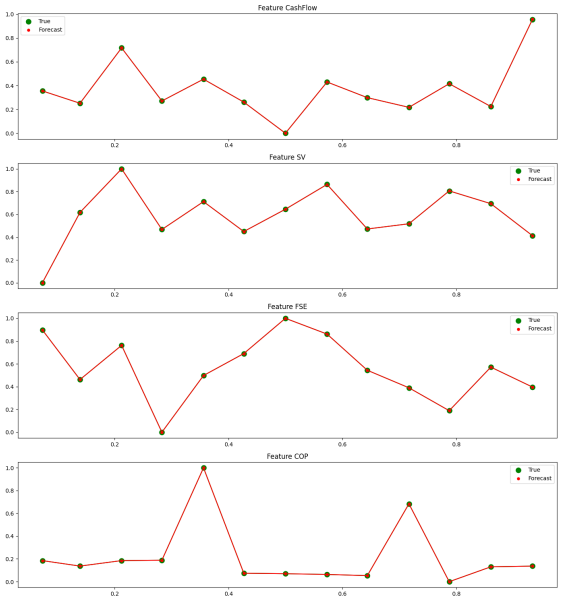
Due to the results obtained, it was necessary to transition to a different architecture capable of better capturing the dynamics of the variables, the ODE-GRU.

The architecture consisted of 4 fully connected layers; the first 3 hidden layers with 50 neurons and an output layer, of which the number of neurons corresponds to the number of features used for prediction. The first two hidden layers correspond to the Neural ODE component of the ODE-GRU, while the third hidden layer belongs to the GRU part. The output layer maps the final hidden state to the predicted values, matching the number of features in each observation. Each group Id was trained for 2000 epochs using the Adam optimizer, a learning rate of `1e-3` and the loss function was the Mean Squared Error(MSE).

Monthly Data

In this experiment, the ODE-GRU model was able to learn the underlying data dynamics effectively. However, the ODE-GRU demonstrated even greater precision compared to the GRU model. The predictions were remarkably close to the actual values, indicating that the combination of the Neural ODE with the GRU provided better performance compared to the standalone Neural ODE and GRU models. The results can be seen in Figures (34) and (36).

While the ODE-GRU showed promising results by accurately capturing the dynamics of the training data, it also faced difficulties when predicting unseen data. The model likely suffered from overfitting, as it was highly tuned to the training data, and the patterns it learned did not generalize well to the test set. This indicates that, despite its higher precision in training, the model was unable to effectively extrapolate to new data, as can be seen in Figures (35) and (37).

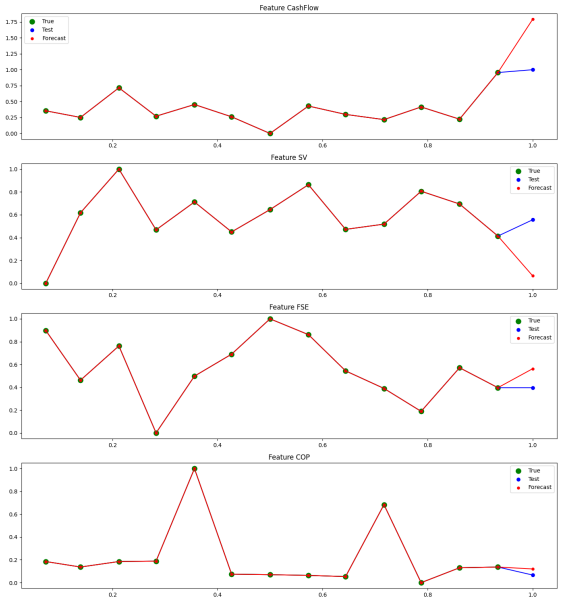


(a) G64



(b) G276

Figure 34: ODE-GRU training results for group IDs G64 and G276 - no DK



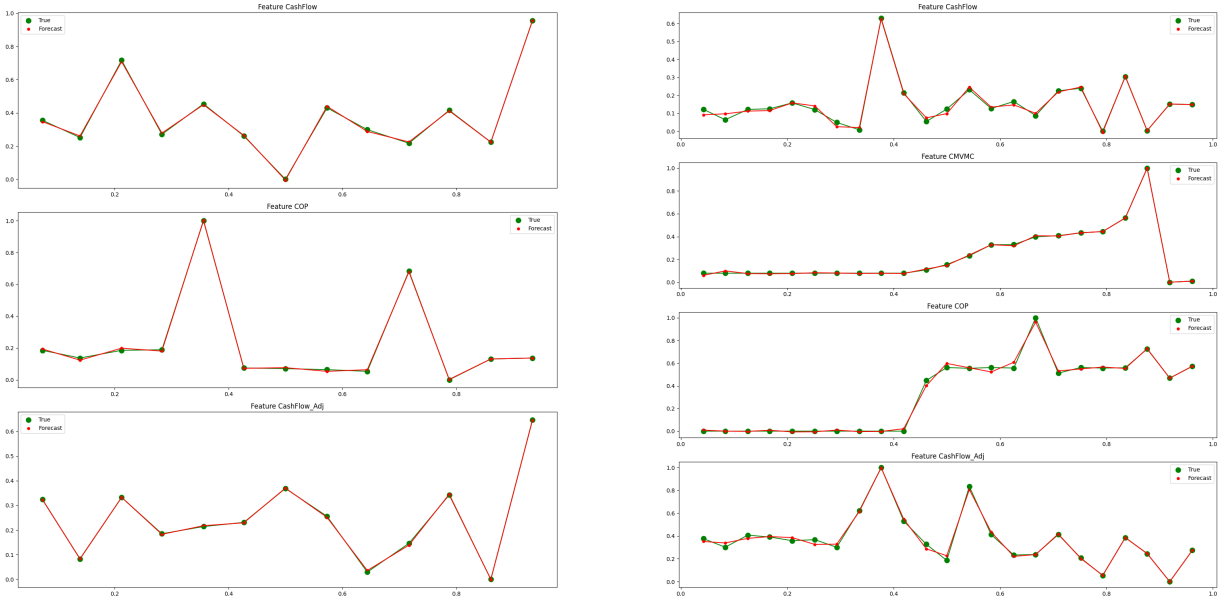
(a) G64



(b) G276

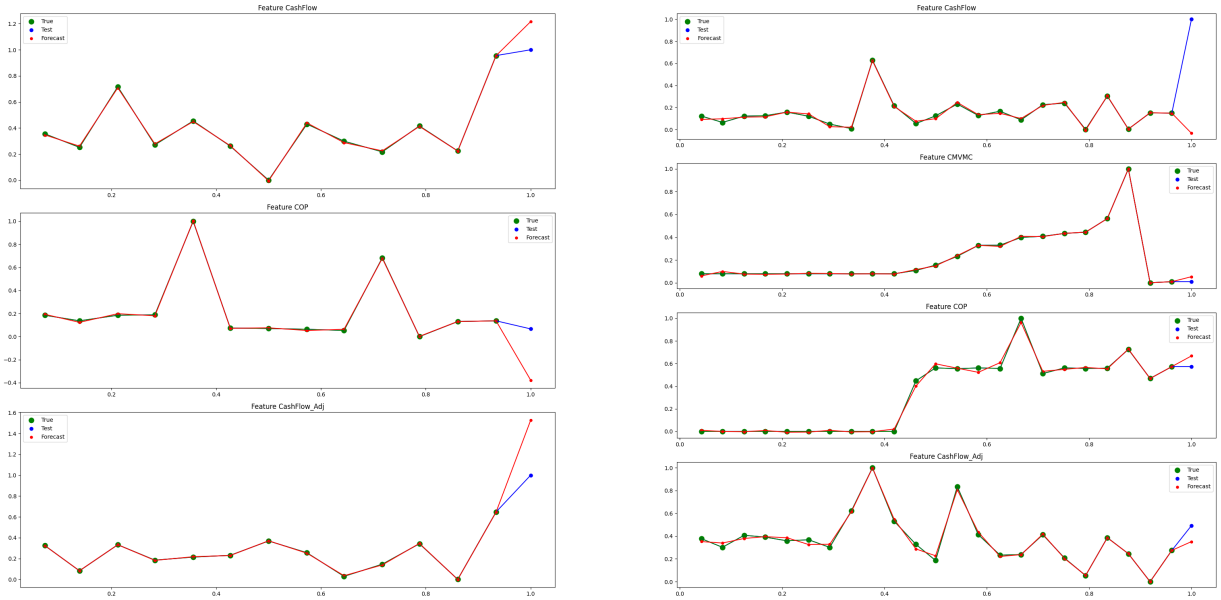
Figure 35: ODE-GRU testing results for group IDs G64 and G276 - no DK

For the ODE-GRU model, similar to the GRU, the model was able to predict the data both with and without DK. However, no notable improvements were observed when integrating DK. Furthermore, when attempting to generalize to unseen data, the model struggled to make accurate predictions, suggesting that the inclusion of DK did not lead to enhance performance. Additionally, when examining the test predictions for group ID G276, the predictions for Cashflow_Adj appear to yield better results compared to the predictions for cashflow without DK.



(a) G64 (b) G276

Figure 36: ODE-GRU training results - with DK



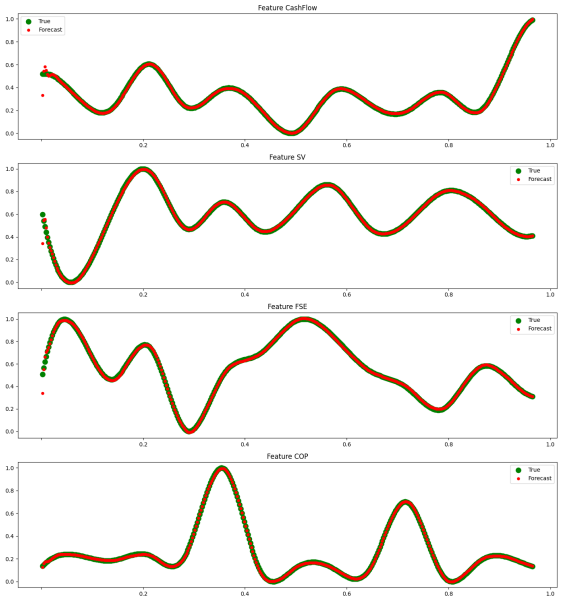
(a) G64

(b) G276

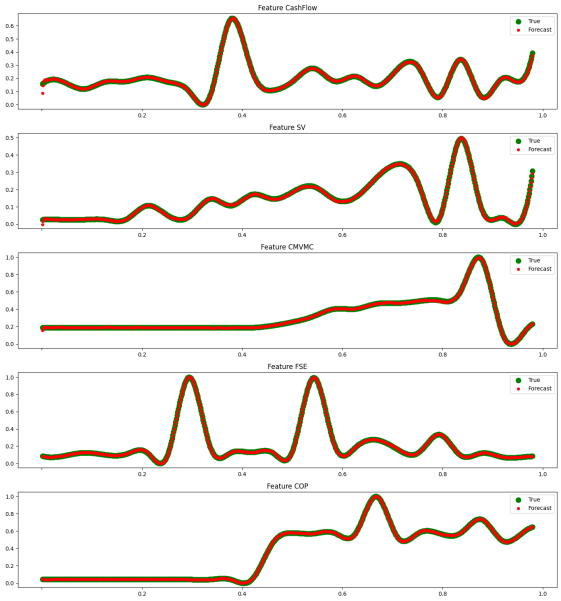
Figure 37: ODE-GRU testing results - with DK

Daily Data - 15 days prediction

In this experiment, where only 15 days were left for test, the model was able to learn the dynamics of the training data effectively, as it can be seen in Figures (38) and (40). However, it still struggled to make accurate predictions for the test set, as it shows in Figures (39) and (41). This suggests that forecasting 15 days into the future may be too challenging for the model given the limited information of the data. The results are similiar for both datasets, with and without DK, which reinforces that the inclusion of DK did not lead to enhanced performance.

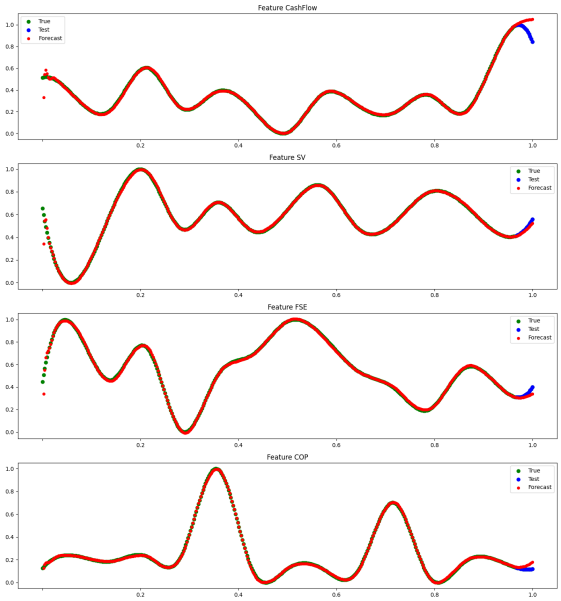


(a) G64

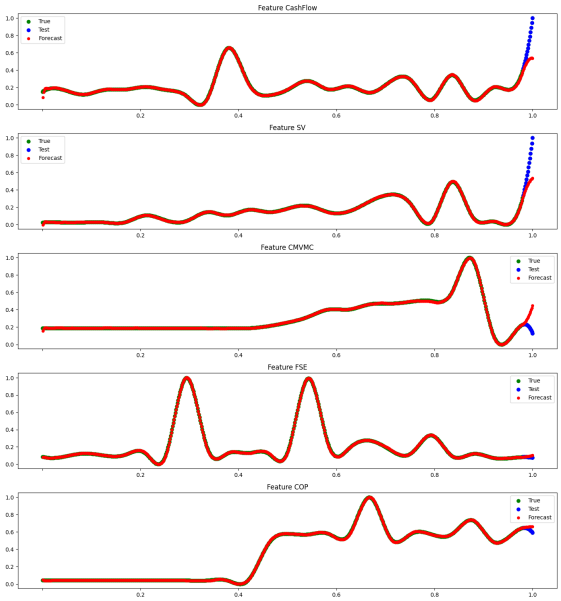


(b) G276

Figure 38: ODE-GRU training results for group ids G64 and G276 - no DK

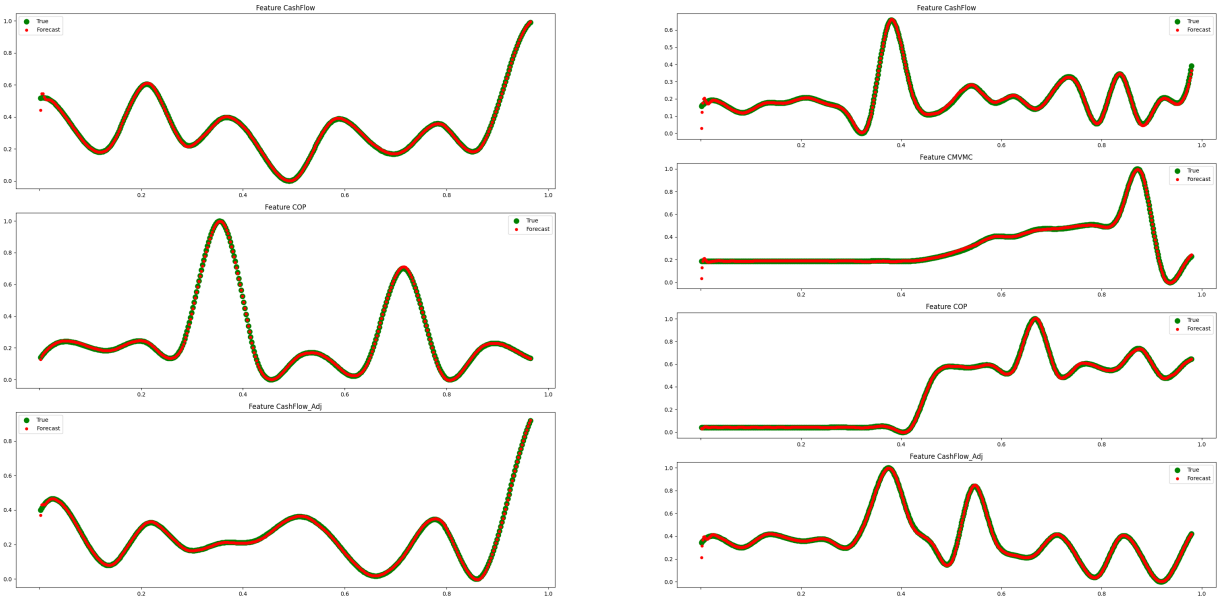


(a) G64



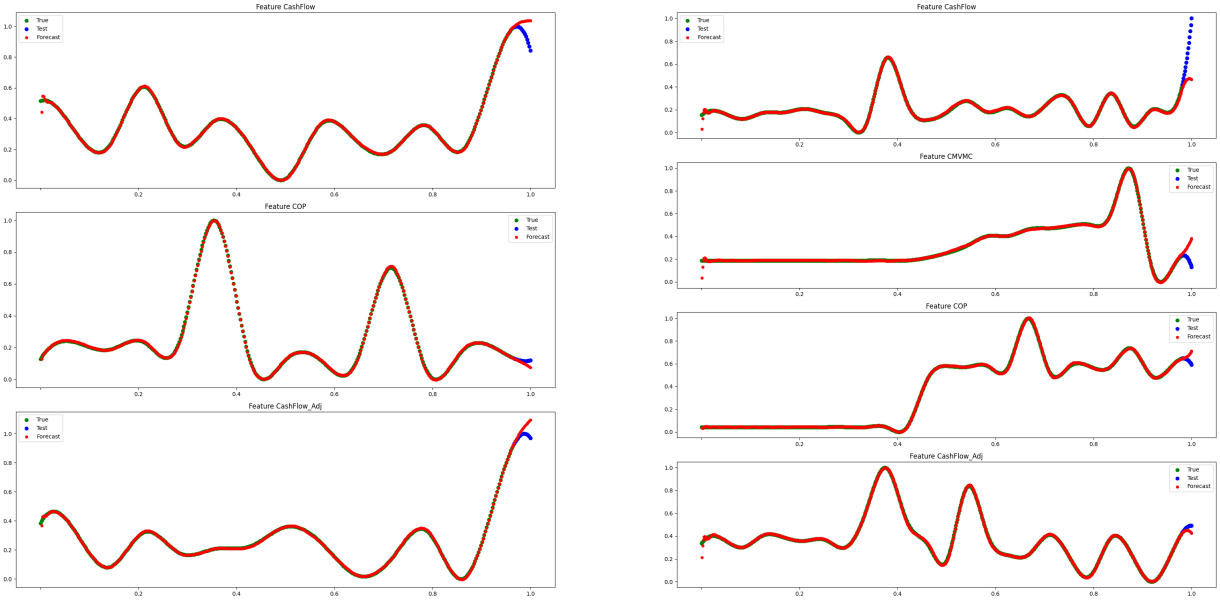
(b) G276

Figure 39: ODE-GRU testing results for group ids G64 and G276 - no DK



(a) G64 (b) G276

Figure 40: ODE-GRU training results - with DK

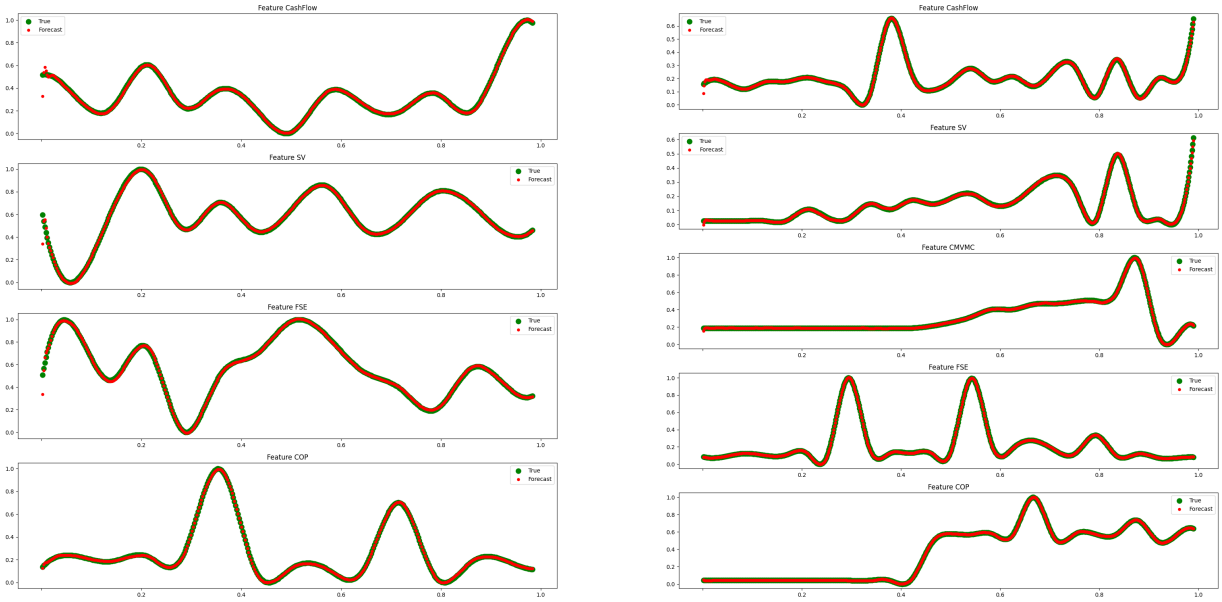


(a) G64 (b) G276

Figure 41: ODE-GRU testing results - with DK

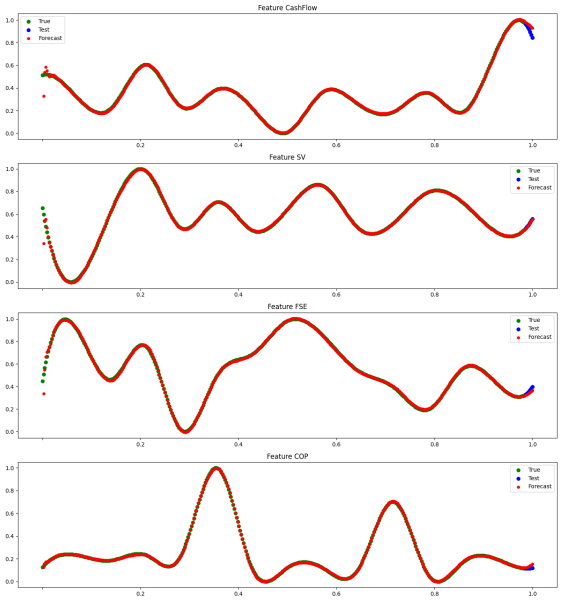
Daily Data - 7 days prediction

In the experiment where only 7 days were left for testing, the model was still able to learn the dynamics of the training data well, as seen in Figures (42) and (44). For the test set, while the model didn't predict perfectly, the forecasts were closer to the true values compared to the previous experiment with 15 days of testing, as it shows in Figures (43) and (45). This suggests that the model performs better when predicting a shorter time horizon. The ability to forecast with higher accuracy over a small number of future steps might indicate that the model struggles with maintaining prediction accuracy as the forecast period extends. Similarly to the previous experiments, DK integration did not enhanced the performance of the model.

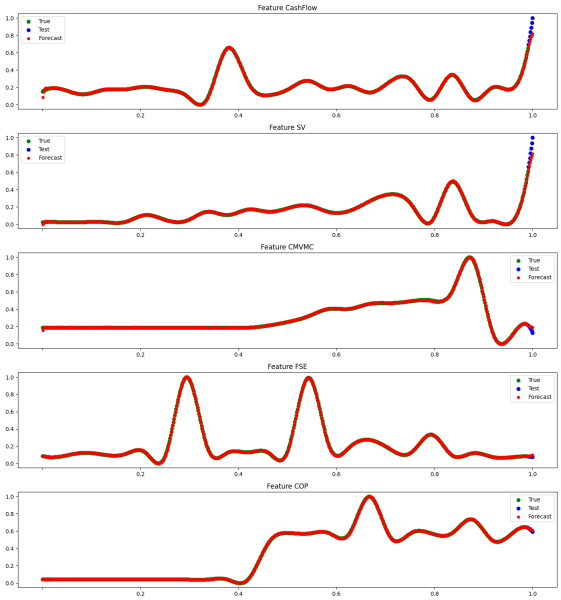


(a) G64 (b) G276

Figure 42: ODE-GRU training results for group ids G64 and G276 - no DK

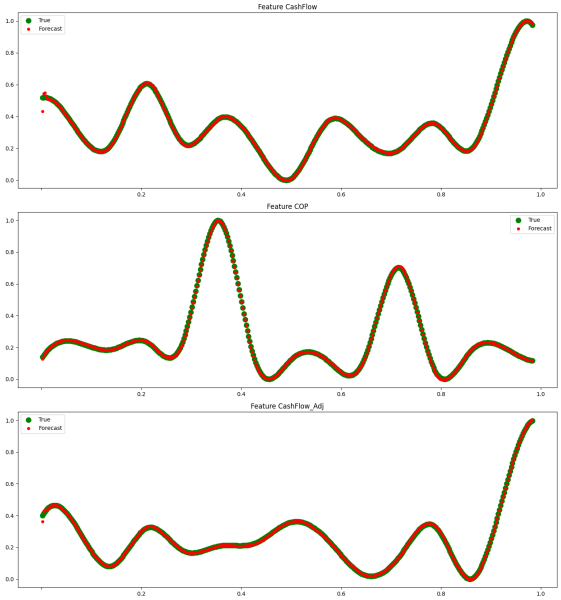


(a) G64

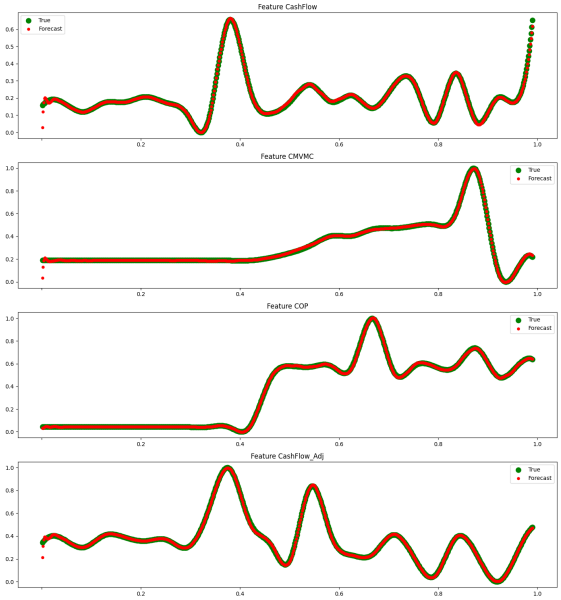


(b) G276

Figure 43: ODE-GRU testing results for group ids G64 and G276 - no DK

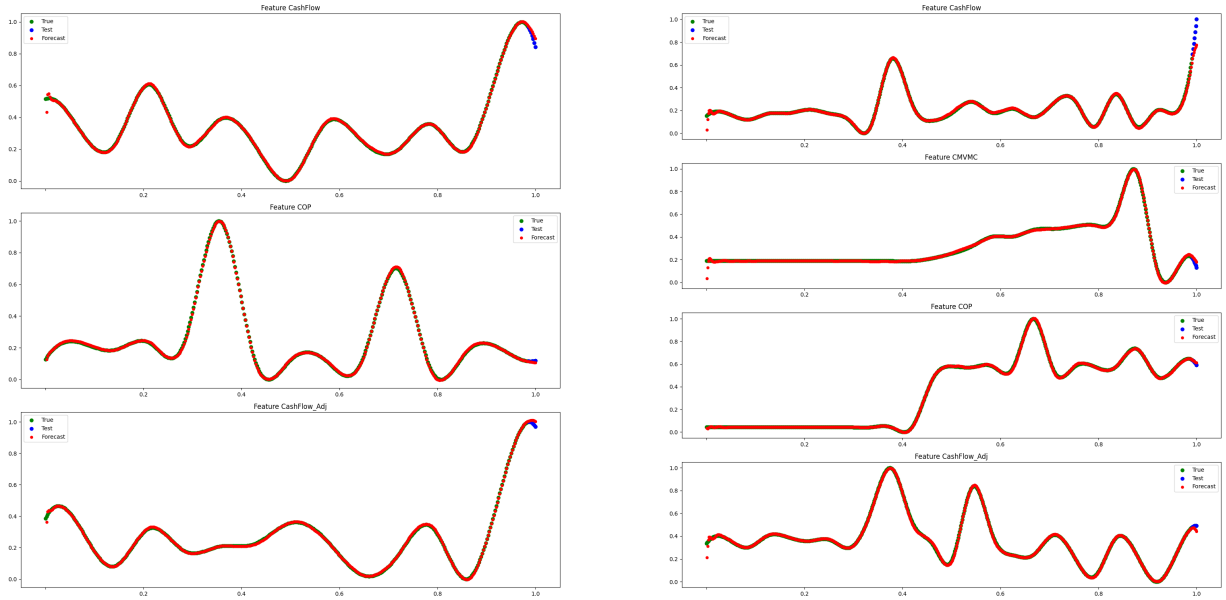


(a) G64



(b) G276

Figure 44: ODE-GRU training results - with DK



(a) G64

(b) G276

Figure 45: ODE-GRU testing results - with DK

5.5 Results Analysis

In the tables, (2) and (3) are presented the MSE values obtained after training and testing the models on group IDs G64 and G276, respectively. It is important to note that the MSE values represent the average calculated across the predictions for all variables, rather than just for the cashflow. The results include the MSE values for all three models: Neural ODE, GRU, and ODE-GRU, across different experimental setups. These setups involve both the original monthly dataset and the daily dataset, with and without DK. For the monthly dataset, for prediction 1 month was used while for the daily dataset 7 and 15 days were considered.

The MSE values in the tables (2) and (3) confirm the observations discussed in the previous subsections. Starting with the results obtained after training with the monthly dataset, for both group IDs, it can be observed that the ODE-GRU model achieved the best results among the three, while the neural ODE had the worst performance. When comparing the datasets with and without DK, a slight decrease in MSE values can be noticed, although not significant enough to show an improvement in the predictions, as shown in the previously presented figures. The test results do not align with the performance observed during training,

making it difficult to draw definitive conclusions regarding the best-performing model. For group Id G64, for the dataset without DK, the MSE values indicated that the best model was the neural ODE while the worst was the ODE-GRU, and for the dataset with DK, the best model was the GRU while the worst was the neural ODE. The dataset with DK only had better performance using the models GRU and ODE-GRU. For group ID G276, for the dataset without DK, the MSE values indicated that the best model was the ODE-GRU while the worst was the neural ODE and for the dataset with DK, the best model was the GRU while the worst was the neural ODE. In this case, the dataset with DK had better performance compared to the dataset without DK with all models.

For the daily dataset, starting with the training of the neural ODE model, the MSE values decreased compared to those obtained with the monthly dataset. The MSE values for the dataset with 7 days for prediction and 15 days for prediction were relatively close, with a slight decrease when using the dataset with DK, though not significant. The test results for group Id G64 indicate better performance when using the dataset with a 7 day prediction window compared to the 15 day prediction window. Additionally, in both cases, lower MSE values were obtained with the dataset without DK. However, for group ID G276, the lower MSE value was obtained with the dataset with DK, particularly with the dataset with 7 days for prediction. And when comparing the dataset without DK, the 15 days dataset had a smaller MSE value.

In the case of the GRU, the training results were worse, as evidenced by the significantly higher MSE values compared to the monthly dataset. The GRU model was unable to learn the dynamics of the daily dataset, which may have been caused by the accumulation of errors throughout the training process. When looking at the test results, for group ID G64 the smallest MSE value was obtained with the dataset without DK and with 7 days for prediction, while for group ID G276 the smallest MSE value was obtained with the dataset with DK and with 15 days for prediction.

The ODE-GRU model once again achieved the lowest MSE values, during training, compared to the other two models. For group ID G64, the inclusion of DK led to improved results after training. However, for group ID G276, this was not the case, as the MSE value increased compared to the dataset without DK. When analyzing the results for datasets with 7-day and 15-day prediction windows, the test results showed that the ODE-GRU model achieved lower MSE values when forecasting 7 days compared to 15 days, as previously observed in the figures.

Additionally, it was observed that, in some cases, the predictions for Cashflow_Adj were better compared

to the ones for cashflow, particularly for group ID G276. While a direct comparison would require reversing the adjustments made to Cashflow_Adj, by adding the contributions of SV and subtracting those of FSE, this observation highlights the potential benefit of incorporating DK into the model.

Train Results						
	No DK (1 month)	**With DK** (1 month)	**No DK** (7 days)	**With DK** (7 days)	**No DK** (15 days)	**With DK** (15 days)
Neural ODE	5.56e-02	4.74e-02	4.69e-02	3.96e-02	4.61e-02	3.65e-02
GRU	3.00e-03	8.00e-04	4.90e-02	3.41e-02	6.41e-02	3.17e-02
ODE-GRU	1.25e-07	3.99e-05	1.00e-04	2.14e-05	1.00e-04	1.73e-05
Test Results						
	No DK (1 month)	**With DK** (1 month)	**No DK** (7 days)	**With DK** (7 days)	**No DK** (15 days)	**With DK** (15 days)
Neural ODE	4.05e-02	2.82e-01	1.89e-02	2.10e-01	3.96e-02	3.45e-01
GRU	8.41e-02	6.41e-02	1.29e-01	3.11e-01	1.73e-01	3.45e-01
ODE-GRU	2.24e-01	1.74e-01	9.00e-04	5.00e-04	3.10e-03	4.20e-03

Table 2: MSE for Different Models using Group ID G64

Train Results						
	No DK (1 month)	**With DK** (1 month)	**No DK** (7 days)	**With DK** (7 days)	**No DK** (15 days)	**With DK** (15 days)
Neural ODE	2.48e-02	2.19e-02	1.93e-02	1.77e-02	1.88e-02	1.77e-02
GRU	1.00e-03	5.00e-04	4.39e-02	4.11e-02	3.84 e-02	3.96e-02
ODE-GRU	5.00e-04	3.00e-04	1.23e-05	4.70e-03	1.19e-05	4.17e-05
Test Results						
	No DK (1 month)	**With DK** (1 month)	**No DK** (7 days)	**With DK** (7 days)	**No DK** (15 days)	**With DK** (15 days)
Neural ODE	3.96e-01	2.52e-01	1.86e-01	9.16e-02	1.53e-01	1.00e-01
GRU	3.55e-01	1.93e-01	1.99e-01	1.38e-01	1.24e-01	9.36e-02
ODE-GRU	2.95e-01	2.75e-01	4.70e-03	5.00e-03	2.64e-02	2.40e-02

Table 3: MSE for Different Models using Group ID G276

6 Conclusions and Future Work

The aim of this study was to evaluate the performance of predictive models with the integration of DK in financial time series forecasting, focusing specifically on cashflow forecasting. In order to achieve this, various methods for integrating DK were initially explored. However, due to the nature of the DK available regarding the dataset, it was not possible to apply the methods studied. Consequently, a more hands-on approach was adopted. This approach involved the creation of a new feature, CashFlow_Adj, which incorporated the domain-specific knowledge about the original features. To evaluate the performance of the DK integration, experiments were conducted using both the original dataset and the modified version with integrated DK. The first experiments were made only with these two datasets, but due to the poor results, the datasets were further refined by transforming the monthly observations into daily ones, increasing the size of both datasets with the intention of improving the model performance. Two experiments were run with the daily dataset, one with 7 days for testing and the other with 15 days. This allowed an evaluation of how well the models performed with different prediction windows.

The initial experiments were carried out using a neural ODE because of its ability to model continuous temporal dynamics. However, the model struggled to learn from the training dataset, even when a larger dataset was used, indicating that it was not the most appropriate choice for this problem. The GRU model was then selected for its effectiveness in capturing temporal patterns in sequential data. While it performed well on the monthly dataset, it failed to generalise to unseen data, suggesting that more years of data are needed to improve predictions. Finally, the ODE-GRU was used for its ability to combine the strengths of Neural ODE and GRU for modelling complex temporal relationships. Although the ODE-GRU outperformed the other models during training, it also struggled to make accurate predictions for unseen data in the monthly dataset. When tested on the daily dataset, the ODE-GRU showed improved performance, particularly in predicting the 7-day dataset compared to the 15-day dataset, suggesting that the model is better at predicting shorter time horizons.

Despite DK integration, no significant improvement in model performance was observed, indicating that further refinement in how DK is integrated might be needed. The transition from monthly to daily data led to slight improvements, but these were limited, as the daily data was still based on the limited original dataset. Improvements were only achievable when forecasting over a short time window of 7 days.

Future work should focus on collecting more data over a longer period of time to help the models learn more substantial patterns. Another possible line of investigation involves obtaining cashflow forecasts by applying the inversion procedure to the Cashflow_Adj predictions, allowing for a more comprehensive comparison between the predictions with and without DK integration. Additionally, further investigation is required into alternative methods of incorporating DK, as the methods studied did not allow for the integration of the existing knowledge due to its uncertain nature. Investigating new methods for integrating DK, together with collecting more data, could lead to more effective use of existing knowledge and potentially improve prediction performance.

Referências

[1] TreviPay. Cash flow forecasting: A comprehensive how-to guide for businesses. (Online resource - https://www.trevipay.com/resource-center/blog/cash-flow-forecasting-a-comprehensive-how-to-guide-for-businesses/- last accessed on 25/10/2024), n.d.

[2] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[3] Jason Brownlee. *Deep learning for time series forecasting: predict the future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 2018.

[4] Robert H. Shumway and David S. Stoffer. *Time series analysis and its applications*, volume 3. Springer, 2000.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[6] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

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

[8] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[9] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[10] CFI Team. Domain knowledge (data science). (Online resource - https://corporatefinanceinstitute.com/resources/data-science/domain-knowledge-data-science/), 2024.

[11] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[12] Himel Das Gupta and Victor S Sheng. A roadmap to domain knowledge integration in machine learning. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*, pages 145–151. IEEE, 2020.

[13] Tirtharaj Dash, Sharad Chitlangia, Aditya Ahuja, and Ashwin Srinivasan. A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Scientific Reports*, 12(1):1040, 2022.

[14] Sandro Radovanović, Boris Delibašić, Miloš Jovanović, Milan Vukićević, and Milija Suknović. Framework for integration of domain knowledge into logistic regression. In *Proceedings of the 8th international conference on web intelligence, mining and semantics*, pages 1–8, 2018.

[15] Ian H.Witten and Eibe Frank. *Data mining*. ELSEVIER, 2nd edition, 2005.

[16] Mohammed J. Zaki and Wagner Meira Jr. *Data mining and machine learning: Fundamental concepts and algorithms*. Cambridge University Press, 2020.

[17] Eyal Krupka and Naftali Tishby. Incorporating prior knowledge on features into learning. In *Artificial Intelligence and Statistics*, pages 227–234. PMLR, 2007.

[18] Nikhil Muralidhar, Mohammad Raihanul Islam, Manish Marwah, Anuj Karpatne, and Naren Ramakrishnan. Incorporating prior domain knowledge into deep neural networks. In *2018 IEEE international conference on big data (big data)*, pages 36–45. IEEE, 2018.

[19] Yaser Abu-Mostafa. A method for learning from hints. *Advances in Neural Information Processing Systems*, 5, 1992.

[20] Arka Daw, Anuj Karpatne, William D Watkins, Jordan S Read, and Vipin Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling. In *Knowledge Guided Machine Learning*, pages 353–372. Chapman and Hall/CRC, 2022.

[21] Zhiting Hu, Zichao Yang, Russ R Salakhutdinov, LIANHUI Qin, Xiaodan Liang, Haoye Dong, and Eric P Xing. Deep generative models with learnable knowledge constraints. *Advances in Neural Information Processing Systems*, 31, 2018.

[22] Kuzman Ganchev, Joao Graça, Jennifer Gillenwater, and Ben Taskar. Posterior regularization for structured latent variable models. *The Journal of Machine Learning Research*, 11:2001–2049, 2010.

[23] Michelangelo Diligenti, Marco Gori, and Claudio Sacca. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244:143–165, 2017.

[24] Michelangelo Diligenti, Soumali Roychowdhury, and Marco Gori. Integrating prior knowledge into deep learning. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 920–923. IEEE, 2017.

[25] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, pages 5502–5511. PMLR, 2018.

[26] Russell Stewart and Stefano Ermon. Label-free supervision of neural networks with physics and domain knowledge. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[27] Charu C Aggarwal et al. *Neural networks and deep learning: A Textbook*, volume 10. Springer, 2018.

[28] Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2018. `http://neuralnetworksanddeeplearning.com/`.

[29] Gábor Petneházi. Recurrent neural networks for time series forecasting. pages 1–3, 2019.

[30] Yaroslav Zhyliuk. What is recurrent neural networks (rnn) and how does it works? (Online resource - `https://medium.com/@y.zhyliuk/what-is-recurrent-neural-networks-rnn-and-how-does-it-works-f3f9ad6ed471`- last accessed on 22/10/2024), 2024.

[31] Anishnama. Understanding gated recurrent unit (gru) in deep learning. (Online resource - https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2- last accessed on 24/10/2024), 2023.

[32] Jason Brownlee. What is teacher forcing for recurrent neural networks?, 2021. (Online resource - https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/-last accessed on 23/01/2025.

[33] Henry J Ricardo. *A modern introduction to differential equations*. Academic Press, 2020.

[34] Yulia Rubanova, Ricky TQ Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. *Advances in neural information processing systems*, 32, 2019.

[35] Pydata. Pandas documentation. (Online resource - https://pandas.pydata.org/- last accessed on 22/10/2024),2024.

[36] NumPy documentation. Numpy. (Online resource - https://numpy.org/doc/stable/- last accessed on 22/10/2024), 2024.

[37] The Python Standard Library. collections — container datatypes. (Online resource - https://docs.python.org/3/library/collections.html#counter-objects- last accessed on 22/10/2024), 2024.

[38] Python Software Foundation. math — mathematical functions. (Online resource - https://docs.python.org/3/library/math.html- last accessed on 21/10/2024), n.d.

[39] Matplotlib Documentation. Matplotlib. (Online resource - https://matplotlib.org/- last accessed on 22/10/2024), n.d.

[40] Pytorch. Pytorch documentation. (Online resource - https://pytorch.org/docs/stable/index.html- last accessed on 24/10/2024), 2023.

[41] Ricky T. Q. Chen. torchdiffeq. (Online resource - https://github.com/rtqichen/torchdiffeq- last accessed on 24/10/2024), 2018.

[42] SciPy. Cubicspline. (Online resource - https://docs.scipy.org/doc/scipy-1.15.0/reference/generated/scipy.interpolate.CubicSpline.html-last accessed on 20/01/2025.

[43] Cecília Coelho. Latentodelstm. (Online resource - https://github.com/CeciliaCoelho/LatentODELSTM/blob/main/nnModels.py-last accessed on 14/01/2025),2023.

[44] PyTorch. Elu. (Online resource - https://pytorch.org/docs/stable/generated/torch.nn.ELU.html-last accessed on 14/01/2025.