Stefan Artur Sobczyszyn Borg

Neural Ordinary Differential Equations for Forecasting in the Energy Sector

Master Project, Spring 2019

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Abstract

Neural Ordinary Differential Equations (NODE) [Chen et al., 2018] is a novel family of neural networks which explicitly model the hidden state as a dynamic system, and solves this system using a numerical ordinary differential equation (ODE) solver as a network component. This thesis performs an initial study on the applicability of NODE to various time series forecasting problems in the energy sector. To the best of my knowledge this has not been done before. Electricity is a challenging commodity due to its short lifetime once produced. Energy providers continuously need to keep track of the demand and supply in the market in order to be profitable. The small margins and large trade quantities means a small improvement in efficiency yields a big improvement in returns. My hypothesis is that NODE have a special advantage on forecasting time series due to their ability to learn the underlying dynamics of the predicted system. Through empirical experimenting, five different NODE models are tested on three energy forecasting problems. These problems have a forecasting horizon of 1 to 24 hours. In addition to forecasting accuracy, claims on advantageous properties are tested for verification, and specific elements of the NODE training process are examined. The results indicate NODE models are able to outperform existing models on a 24 hour horizon, and perform comparable on a 1 hour horizon. As the findings are promising, the conclusion is that further experimentation on NODE could lead to valuable results.

Sammendrag

Nevrale Ordinære Differentiallikninger (NODE) [Chen et al., 2018] er en ny familie av nevrale nettverk som eksplisitt modellerer nettverkets skjulte tilstand som et dynamisk system, og løser dette systemet ved å bruke en numerisk differensialligningløser som en komponent i nettverket. Denne oppgaven gjennfører en innledende undersøkelse av NODE til prognosering av ulike tidsserieproblemer i energisektoren. Så vidt jeg vet, har dette ikke blitt gjort før. Elektrisitet er en utfordrende vare på grunn av sin korte levetid når den først er blitt produsert. Energileverandører må kontinuerlig følge med på etterspørselen og tilbudet i markedet for å være lønnsomme. De små marginene og store handlede kvanta betyr at en liten forbedring av effektivitet gir en stor forbedring i avkastning. Min hypotese er at NODE har en spesiell fordel til prognosering av tidsserier på grunn av deres evne til å lære den underliggende dynamikken til det konrekte systemet. Gjennom empirisk eksperimentering testes fem forskjellige NODE-modeller på tre energiprognoseproblemer. Disse problemene har en prognosehorisont på 1 til 24 timer. I tillegg til å undersøke nøyaktighet, blir påstander om fordelaktige egenskaper testet for verifisering, og spesifikke elementer i NODE-treningsprosessen blir undersøkt. Resultatene indikerer at NODE-modeller er i stand til å utkonkurrere eksisterende modeller på en 24-timers horisont, og er sammenlignbare på en 1-timers horisont. Med disse lovende funnene, er konklusjonen at ytterligere eksperimentering med NODE kan føre til verdifulle resultater.

Preface

This thesis is the last milestone in concluding the five year Computer Science Master's program at the Norwegian University of Science and Technology (NTNU) in Trondheim. It has been supervised by Odd Erik Gundersen and is a collaboration with TrønderEnergi.

I would like to thank Odd Erik and Gleb at TrønderEnergi for their time and effort in discussing and giving valuable input on my work.

Most importantly, I want to thank my family for all their support. Especially you Anna and Jacek. Thank you for your unconditional love through all the ups and downs during these years. For that I am grateful.

Note on Collaboration

This thesis is a collaboration with TrønderEnergi AS. As part of this collaboration TrønderEnergi has shared resources for running the experiments and proprietary company software. Because of the partial dependency on this software, the software code developed for experimentation has not been published.

Note on Notation

Throughout this paper there are some reappearing variables. Effort has been made to keep these variables consistent. To avoid the burden of additional information, the type of these variables will be explained here, and not elaborated on in the text. Unless otherwise specified or a different context is apparent the following definitions apply:

x, y, z - Are reserved for vectors and matrices. Where the variables are indexed with a sequence s.a. $t : T$ the variables are a vector of vectors - i.e. a matrix.

t, S, T - Are scalars reserved for indices and other forms related to time.

N, n, i, d - Are scalars reserved for counting.

Contents

List of Figures

List of Tables

1 Introduction

This chapter gives a high level introduction of the thesis. The chapter begins with an introduction. This is followed by the motivation for the work. In section 1.3 the research goal, research questions and hypothesis is presented. Section 1.4 explains the research approach, summarized in three main phases. Section 1.5 shortly states the contributions of the thesis, before the chapter is rounded off in section 1.6, laying out the thesis structure.

1.1 Introduction

In recent years neural networks have been reintroduced as an excellent tool for function approximation, reaching state-of-the-art results in domains such as computer vision (object detection, image recognition and more) [Krizhevsky et al., 2012], and natural language processing [Graves et al., 2013]. Two main reasons can be accredited this growth:

1. **Increased amounts of available data** - In our digital time and age, data is everywhere. Thousands and thousands of images and texts are available on the internet. The internet of things allows sensors and other devices to stream data real time over the internet. Over the years, this data has been stored and continues to increase in size. At the same time, storing and retrieving this information has been made easier by new hardware and software.

2. **Increased computing power** - Development of hardware components has increased the efficiency of training neural networks. Networks which previously would take days to train, can now be trained within hours, or even minutes. This means neural networks can be trained faster and with more data.

Neural Ordinary Differential Equations (NODE) [Chen et al., 2018] is a novel family of neural networks which takes advantage of these factors. The main component of NODE is an ODE Block consisting of a numerical ordinary differential equation (ODE) solver and a neural network defining the dynamics of the ODE. Inspiration for treating a neural network as a dynamic system stems from observations on the ResNet [He et al., 2015]. Particularly, it has been shown the residual connections can be thought of as solving one step of Euler's formula [Liao and Poggio, 2016; Weinan, 2017]. NODE models have shown interesting results on time series data, showing the models are able to learn the dynamics of some simple equations [Chen et al., 2018]. However, these experiments were run on synthetic data with well known dynamics. So far, little is known about NODE performance applied to real world problems. This thesis seeks to examine just that, by applying NODE on several time series forecasting problems from the energy sector.

Three experiments are conducted, each considering a different forecasting problem. The NODE component is integrated in five different models, which are compared to each other and to existing machine learning models. Two of the NODE models from [Chen et al., 2018] are used. In addition to this, three new architectures are proposed and tested. Other than evaluating accuracy, claims made by Chen et al. [2018] are tested for verification. Specifically, experiments on synthetic data show NODE are able to approximate functions using fewer parameters and less training data. Also selected properties of the ODE Block, related to approximation accuracy, are investigated.

The results show that NODE have the highest accuracy on a 24 hours forcast horizon, and perform comparably on problems considering a one hour forecast horizon. It is shown that some of this performance is likely due to the NODE specialized component. However, the experiments were not able to verify the claims on additional perks related to parameters and training data.

The conclusion is that NODE show promising results, and that further investigation can yield improvements over what is seen. To this extent, further research on the models is required. Especially, it would be interesting to extend the time horizon for all problems.

1.2 Motivation

One of the challenges with electricity as a commodity is that it, once produced, can not be easily stored. At the same time there must always be an exact balance between production and consumption. If there is an overweight of consumption there is not enough power for everybody. An overweight of production leads to wasted energy. A tool to balance the supply and demand of energy is the open power market.

Since 1991 the Norwegian power market has been an open market where companies and other entities can buy and sell electricity following supply-demand principles. The market is divided into wholesale and end-user markets. The end-user market is a market between power suppliers, end-users (i.e. households), and smaller businesses and industry. The wholesale market is a much bigger market were large volumes of power are bought and sold by power producers, brokers, and large industrial entities. This market consists of the day-ahead market, the continuous intraday market, and the balancing markets.

The primary market of the wholesale market is the day-ahead market. In this market the prices are determined for each separate hour of the following 24-hour period. Participants participate in an auction, making bids and offers for delivery contracts. Power prices are set based on all the buy and sell orders received, and the capacity for each *bidding area*. The Nordic power areas are divided into *regions* as seen in figure 2.2. All the orders are fulfilled at the end of the 24-hour period, called *the hour of operation*.

Because of this discrepancy between time of purchase and time of delivery, unforeseen events can cause an auction winner not being able to fulfill their contracts. For instance, a power supplier can realize it will not be able to produce the said amount of power, or realize it will actually be able to produce more than the said amount. This is where the intraday market comes in to play. In the intraday market contracts can be traded between participants to ensure the balance between supply and demand is met. This market is open up to one hour before the hour of operation. Trading at this point is called *replanning*.

Even though the day-ahead and intraday markets create a good market balance, it is unavoidable that an imbalance occurs at a specific hour of operation. At the final hour before delivery the *transmission system operator* (TSO) [1] ensures, through the balancing market, that there is a final balance. The TSO ensures sufficient balancing capacity through separate reserves and contracts with electricity providers. The wholesale market is illustrated in figure 1.1.

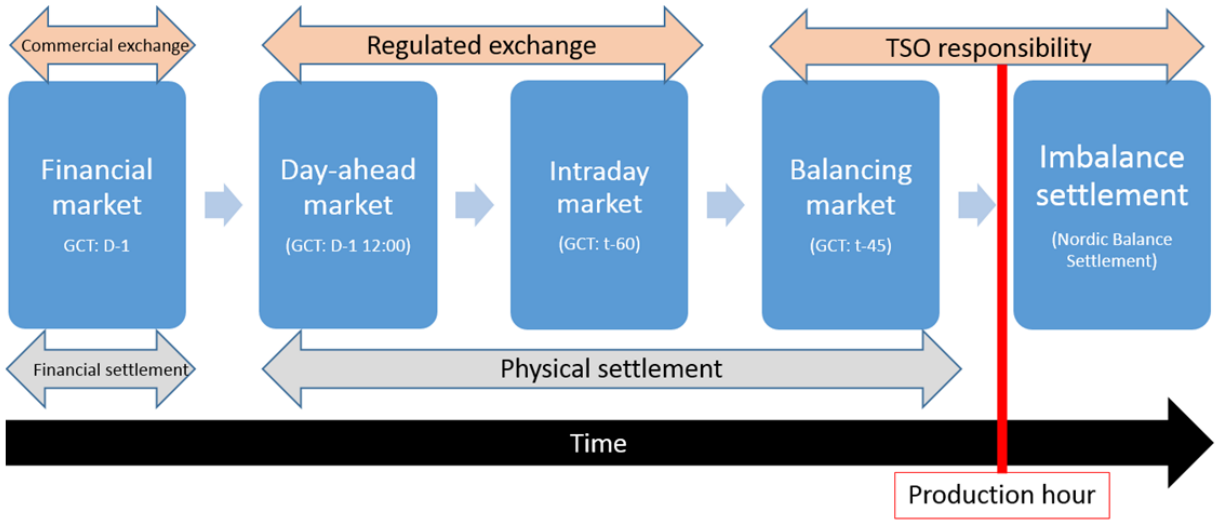


Figure 1.1: Structure of the open power market. This thesis considers the physical settlement markets. Adopted from NVE [2019].

[1] A TSO is an entity responsible for transporting energy. For practical reasons the TSO usually has a monopoly in the segment. This entity is Statnett in Norway.

Being an energy company, TrønderEnergi interacts with both the wholesale market and end-user market continuously, both on the producer and consumer side. By accurately forecasting internal and external processes related to these markets, TrønderEnergi can leverage better information to increase profits of its operation. In section 2.1.1 the specific forecasting scenarios studied in this thesis are introduced.

1.3 Goal, Research Questions, and Hypothesis

Goal: *Perform an initial feasibility study of NODE for time series problems from the energy sector.*

The main objective of this thesis is to conduct an initial feasibility study investigating whether NODE can be successfully applied to time series forecasting in the energy sector. The main interest is evaluating how NODE fare against existing forecasting methods at TrønderEnergi in terms of accuracy, with other factors as a secondary interest. Work suggest NODE are specifically suited for time series with (un)known underlying patterns. I hypothesize that NODE can predict these patterns better than other methods by modelling the patterns as dynamic systems, learned from data, and solving them with methods from calculus.

Research question 1: *Can NODE models consistently beat existing methods in forecasting accuracy?*

This is the most important question, and our main interest. In energy forecasting small increases in accuracy can have huge benefits in profit. If NODE models can consistently outperform existing models, this gives a clear incentive to integrate the methodology in production.

Research question 2: *What NODE model is the most suiting?*

The NODE ODE Block component can be integrated into different models in different ways. It is interesting to find out what model has the best performance in terms of forecasting accuracy. Even if NODE does not answer RQ1 positively, an answer to this question could lay ground for what NODE model(s) to pursue for possible future research.

Research question 3: *Does the claim on NODE models needing less training data hold?*

One of the claimed perks of NODE models is achieving the same accuracy as other models, using less training data. Knowing whether this claim holds for problems in energy is of use, as there exist problems were access to data is scarce. Depending on answers to RQ1 and RQ2 there can be situations were NODE are preferable based on the amount of available data.

Research question 4: *Does the claim on NODE models needing fewer parameters hold?*

One of the claimed perks of NODE models is achieving the same accuracy as other models, using less trainable model parameters. In the machine learning community there is a general preference to models with fewer parameters. Models with fewer parameters are generally easier to understand and train. Another factor where the size of a model is relevant is in storage and transportation of the model to and from memory. Also, when running a model, it requires the entire model to be fit into the working memory. This in terms limits the batch size of training data and the entire training process.

Research question 5: *What part of the latent representation in the Latent ODE Regressor v3 model is the most important for accurate forecasts?*

During experimentation with the Latent ODE Regressor v3 model (one of the NODE models introduced in section 3.1), some unexpected behaviour was observed. This raised the question of how to optimally configure the NODE model for accurate predictions. Answering this question will give future experimentation a more informative starting point.

1.4 Research Approach

The work in this thesis was created over the course of five months, and can be discretized into three phases. The phases have been followed in parts chronologically, and in parts overlapping and iteratively. As work moves on change in plans is highly likely due to new information and insights. Backtracking to earlier phases and adjusting for new findings can thus occur. In addition, with time new research on relevant topics is published. Staying updated in the field necessarily requires continuous study. This is especially true for a new methodology, as NODE are. The three phases were:

Phase I: Background Study. In this phase the main objective was to get familiar with the research context. This entailed reading a lot of scientific papers and other relevant content. Getting a good understanding of the NODE principles and methodology, as well as getting familiar with related theory and models NODE build upon, was crucial before continuing.

Phase II: Preparation of Experimental Environment. In this phase the main objective was to prepare an environment in which experiments could be conducted cleanly. Among tasks performed during this phase was integrating models with relevant data, verifying validity of the data input to models, proper handling of saving and loading of models, and consistent visualization across models. This phase also conducted initial informal tests. These acted as a filtering for deciding whether to take problems and models to the formal experimenting in phase III. As the main research interest is an initial study of forecasting accuracy, testing more models on more problems was prioritized, leaving less time for formal experimenting in phase III. This prioritization allowed experimentation of an additional forecasting scenario and an additional NODE model. Summed up, this phase was a precursor to avoid errors and wasted time during formal experiments.

Phase III: Experimentation. In this phase the main objective was to plan, conduct, and document experiments. The experiments had to be planned so to answer the research questions. Conducting the experiments had to be done meticulously to ensure they were consistent with the plan, and to ensure the results were sound and valid. Documenting meant t save the right plots, and numerical results for tables. Also, while documenting, an initial verification of the results was conducted.

1.5 Contributions

This thesis has the following contributions:

1. Five different NODE models are tested on three different energy forecasting problems, and compared against existing machine learning models. Results indicate the NODE models can outperform existing models on 24 hour ahead forecasts, and perform comparably on 1 hour ahead forecasts.

1.6 Thesis Structure

The thesis is structured in the following six chapters:

Chapter 1 Introduction: Introduces the thesis, motivation, research goals, questions, hypothesis, and contributions.

Chapter 2 Background: Gives an overview of underlying theory used in later chapters. The chapter starts by introducing three forecasting scenarios, before relating these to a machine learning approach, and introducing existing machine learning models at TrønderEnergi. Next, two machine learning models related to NODE are introduced, before NODE themselves are explained. Finally, the chapter ends with an exploration of work related to the thesis.

Chapter 3 Experiment Design: This chapter plans and describes the conducted experiments. First, the NODE models used are described. Then, the experimental plan for the three experiments is explained. Next a section on the experimental setup, listing all model parameters and information needed to recreate the experiments. Finally, a summary of model properties in relation to each experiment is presented.

Chapter 4 Results: Presents the results from the conducted experiments. Results from each experiment is chronologically presented and commented. The chapter ends with a summary of the main take-aways from all experiments.

Chapter 5 Evaluation and Discussion: This chapter starts with an evaluation of the experiments. Next, the results are discussed in light of the research questions, before the chapters ends with a discussion on some apparent limitations of the work in the thesis.

Chapter 6 Conclusion and Future Work: The last chapter of the thesis presents the conclusion and states some interesting topics for future work.

2 Background

This chapter presents the background in detail, and states relevant previous work related to the thesis. First, a section on forecasting is presented in section 2.1. Three specific forecasting scenarios are explained. Then the forecasting problem is stated as a machine learning problem, and a few currently methods used at TrønderEnergi are presented. The forecasting section ends with common time series terminology. Section 2.2 introduces NODE. First, two relevant neural network models are explained, before the NODE component, NODE models, and training algorithm is presented. The second part of the background looks at related work, with a main weight on work related to NODE.

Scope. Throughout this chapter (with the exception of section 2.1) a general understanding of machine learning and neural networks is assumed. This includes common models such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), common activation functions such as ReLU, and parameter update algorithms such as stochastic gradient descent and backpropagation. Instead, this chapter will focus on immediately relevant and more advanced subtopics of deep learning and machine learning. For a general introduction to the fields, the reader is referred to Goodfellow et al. [2016]; Mitchell [1997]; Russell and Norvig [2014] and references therein, as well as references within this chapter. For a detailed tutorial on variational auto-encoders (introduced in section 2.2.2) see Doersch [2016].

2.1 Forecasting

2.1.1 Energy Forecasting Scenarios

TrønderEnergi has a wide range of interactions with the power market. As an energy producer they offer and bid for contracts in the day-ahead market. As a company with office buildings TrønderEnergi is an end-user in the end-user market. This section studies the specific forecasting scenarios related to the thesis.

One Hour Ahead Forecasting of Windmill Power Production

The one hour production forecast scenario is related to trading contracts in the balancing market. The bidding in the balancing market closes 45 minutes before the hour of operation. To trade on this market, TrønderEnergi needs to know how much energy they are able to produce. One of the production sources are windmill parks. With accurate one hour ahead production forecasts TrønderEnergi is better positioned to make informed trading decisions.

To create the forecasts previous production and weather data is combined, and a forecasting model predicts the production at the next hour. The source of the production data comes internally, as each windmill logs its production. As the wind parks take some time to process the production data, the earliest available data comes at a lag of one hour. This means the model does not have the data from the previous hour available. The weather data is extracted externally through agreements with forecast providers. For windmills it is especially the wind speed and wind direction which are important factors for the final production.

Short Term Forecasting of Office Building Consumption

Electricity for TrønderEnergi's headquarter office building is bought in the end-user market. A common price model for industry clients is to discretize a period into ticks, and the company pays a price equal to the highest tick for each tick in the period. From the supplier side, the peak demands can be challenging to produce. Therefore, the rationale behind this model is to encourage a stable consumption of energy.

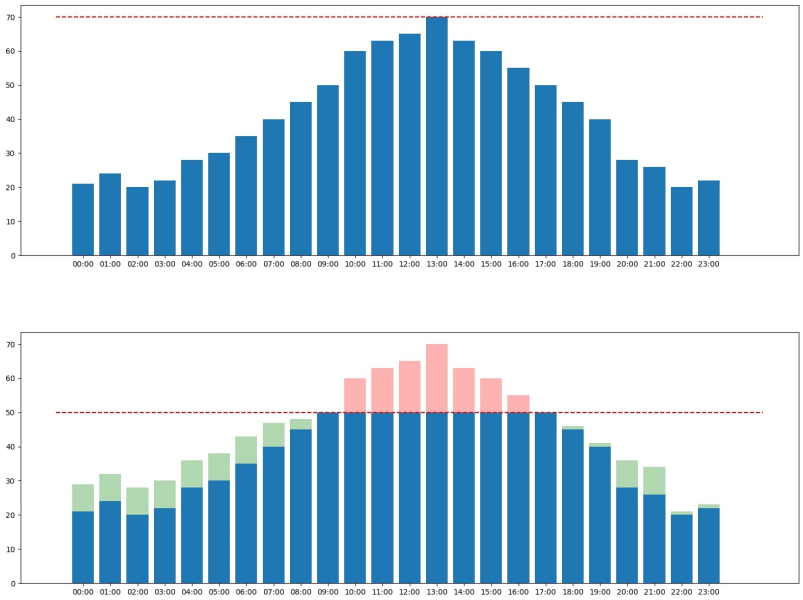


Figure 2.1: Theoretical illustration of electricity price before (top) and after (bottom) rescheduling building consumption. The red dotted line indicates the electricity price for each tick in the period. After rescheduling the price drops from 70 to 50 units.

Office consumption has certain rigid processes, needing to be scheduled at specific times, and processes that are flexible. If the peaks in consumption are known, the flexible processes at the peaks can be rescheduled to a later or earlier time. This way the tops are cut without affecting the productivity of the company as exemplified in figure 2.1. Note that in this problem we are especially interested in accurate predictions at the peaks.

The scenario considers a one hour ahead forecast. To create the forecasts previous consumption and temperature data is combined, and a forecasting model predicts the consumption for the next hour. The source of the consumption data comes from sensor installed locally. This data is available continuously, which means the model has the consumption data from the previous hour available. The temperature data is extracted through external services.

Day-Ahead Forecasting of NO3 Region Consumption

The contract prices in the day-ahead market depend on both the consumption demand and the production from consumers and suppliers respectively. This forecasting scenario targets the consumption demand. The power market is divided into several regions as shown in figure 2.2. TrønderEnergi operates mainly in the NO3 region. Knowing the demand in this region will help TrønderEnergi adjust the prices of their production. If the consumption forecast is lower than what the market predicts, TrønderEnergi needs to assure to sell its production and get the day-ahead contracts. In the opposite case TrønderEnergi can wait, and sell its production in the intraday market at a presumably higher price.

The scenario considers a 24 hour ahead forecast. To create the forecasts previous region consumption and temperature data is combined, and a forecasting model predicts the consumption for each of the next 24 hours. The consumption data is available continuously, which means the model has the consumption data from the previous hour available. The temperature data is extracted through external services.

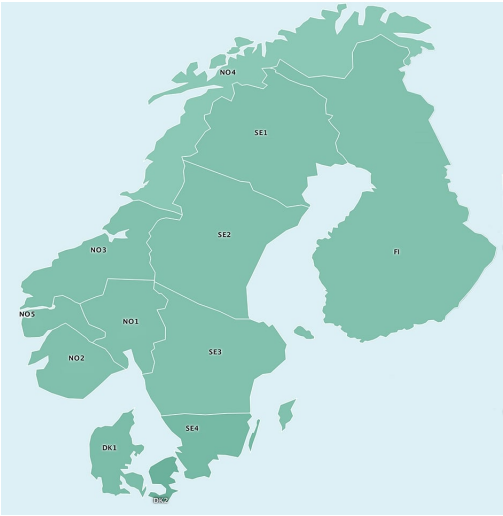


Figure 2.2: The Nordic countries are divided into several power regions. Each region has its own electricity supply and demand, setting the prices for that region. TrønderEnergi is interested in forecasting the total consumption of energy in the NO3 region.

2.1.2 Time Series Forecasting as a Machine Learning Problem

The forecasting problem is posed as a classic supervised machine learning regression problem. We are provided a training dataset $D_{train} : \{\mathcal{X}_{train}, \mathcal{Y}_{train}\}$, sampled from an unknown distribution $P(\mathcal{X}, \mathcal{Y})$ consisting of tuples (x, y) of observed features $x \in \mathcal{X}_{train}$ and target features $y \in \mathcal{Y}_{train}$. Our interest is finding the true distribution $P(\mathcal{X}, \mathcal{Y})$. However, computing the exact distribution is often intractable because of the large state space, or due to complex functions describing the relation between $\mathcal{X}$ and $\mathcal{Y}$. Instead, the distribution can be approximated with machine learning and the given training data.

The supervised learning goal is to use the available training data to learn a function f, parameterized by parameters θ, which maps observed features to the target features, according to the unknown distribution.

$$y = f_\theta(x) \tag{2.1}$$

The ultimate goal for the approximating function f is to *generalize* for unobserved tuples, not available in the training set $(x, y) \notin D_{train}$. In practice, as we do not have access to the true distribution, the model can not be evaluated on the true distribution. Instead the generalization is approximated using a test set $D_{test} : \{\mathcal{X}_{test}, \mathcal{Y}_{test}\}$ which is withheld during training of the model. It is important that tuples in the test set are not present in the training set. Otherwise we can not know whether the model generalizes to unseen samples.

In time series forecasting each tuple (x, y) is associated with a time stamp, t. This adds an additional constraint to the supervised problem that each unobserved tuple should be at a time t_T where $t_T > t$ for all $(x, y)_t \in D_{train}$.

In this thesis, the function in equation 2.1 is approximated using neural networks with, possibly, many hidden layers. The parameters are updated through an iterative gradient descent optimization. The gradients are retrieved based on a loss function applied to the network predictions and the targets in the training set. This means our training goal is to find a θ which optimizes

$$L = \mathcal{L}(y, f_\theta(x)) \tag{2.2}$$

for a chosen loss function $\mathcal{L}$ and tuples $(x, y)_t$ from the test set. A common loss function, and one that will also be used in this thesis, is the *mean absolute error* of all samples N

$$\mathcal{L} = MAE = \frac{1}{N}\sum^{N}|y - f_\theta(x)| \tag{2.3}$$

Based on the loss residuals θ are updated with the help of the backpropagation algorithm [Rumelhart et al., 1986]. By applying the chain rule, the backpropagation algorithm iterates backwards through the net, and computes how much each specific parameter of the model is contributing to the total loss. Stochastic gradient descent then updates each parameter accordingly

$$\theta_{t+1} = \theta_t + \gamma\frac{dL}{d\theta_t} \tag{2.4}$$

for a learning rate γ which is usually a decimal in range $(0, 0.1]$.

This iterative training ends after a preset number of iterations through the training set, or when the loss of the model stops improving. Note that although the parameters are updated using the training set, the final model chosen is the one that shows the best loss on the *test set*. This is the model that best coincides with the ultimate goal of *generalizing to unseen examples*.

2.1.3 Machine Learning Models at TrønderEnergi

Experiments in section 3.2 compare NODE models to existing models at TrønderEnergi. This section summarizes these models. Two of them are versions of neural networks while the last one is an ensemble method based on decision trees [Quinlan, 1986]. This section does not go into technical details on the models, but rather intends to give an intuitive understanding of each method. For details the reader is referred to the cited work and references therein.

Benchmark Model 1: Dense Neural Networks

A dense neural network (DNN) is the simplest form of deep neural nets[1]. These nets are directed graphs consisting of several layers of interconnected perceptrons [Rosenblatt, 1958]. The perceptron in each layer is connected to all perceptron in the previous layer, and outputs a weighted sum of the inputs through a non-linear activation function. Each connection has an associated weight which are the trainable parameters of the model. As in section 2.1.2 the training is treated as an optimization problem in parameter space.

Benchmark Model 2: Long-Short Term Memory Neural Networks

The Long-Short Term Memory neural net (LSTM) [Hochreiter and Schmidhuber, 1997; Gers et al., 2000] integrate advanced units specialized at processing sequence data such as time series. The LSTM unit resembles a recurrent unit in that it has an inner state changing over time based on previous activations, but, unlike a regular recurrent unit their memory is selective. An LSTM unit has two types of memory: (1) The *cell state* which can be thought of as a long-term memory state, and (2) the *hidden state* - working memory. These states are propagated through each LSTM unit. What information is passed on and what information is discarded is decided by a number of *gates* with trainable parameters. The process of each LSTM unit is as follows: First, the hidden state is sent through a *forget gate* and updates the cell state. This gate dictates which parts of the cell state information to discard based on the new observed data and the forget gate parameters. Then, the unchanged hidden state is again propagated through the *input gate*, and another update to the cell state is made. This gate dictates which part of the cell state information to save based on

[1] The coined term "deep" in deep neural net refers to a network with multiple hidden layers.

the observed data and the input gate parameters. Finally, the hidden state is updated according to the *output gate*, and multiplied with the new cell state to get the new hidden state. The cell state and the hidden state are then propagated to the next level of the network. LSTMs can be *unfolded* and undergo the same optimization algorithm as mentioned in section 2.1.2, optimizing the trainable parameters of the different gates.

Benchmark Model 3: Decision Tree Gradient Boosting

Decision tree gradient boosting [Friedman, 2002] is an unsupervised machine learning algorithm based on an ensemble of decision trees. The algorithm resembles the popular Random Forest algorithm [Breiman, 2001]. Summarized it does the following:

1. The first step of the algorithm is to create a single prediction to be the average of the target feature in the training data. The prediction model now consists of a single average leaf node.
2. The second step is to create a new decision tree and place the residuals of each sample in the leafs. The residual is the sum between the current model prediction and the target of the sample. The new model prediction is now the average from the first step, plus a fraction[2] of the prediction from the tree created in the second step. Our model now consists of the first leaf node, plus the first decision tree.
3. The third step is to calculate new residuals for each sample in the training set, based on the latest model, and return to step 2 using these residuals in the leafs. This recursion between step 2 and step 3 continues until a stopping condition is reached, which is normally (1) a preset amount of trees or (2) when adding additional trees does not improve the new predictions.

Prediction on new samples are approximated by running them through the created model. Figure 2.3 illustrates this.

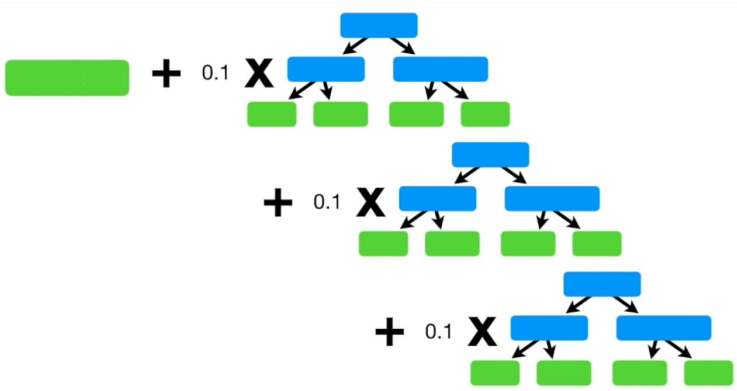


Figure 2.3: An example of the decision tree gradient boosting algorithm. Starting with a single averaging leaf the prediction is improved by adding fractions of predictions from additional decision trees. In this example the learning rate is set to 0.1 and there are three decision trees in the ensemble. Adopted from Starmer [2019].

2.1.4 Common Time Series Terms

When analyzing time series there are *patterns* that are commonly referred to. In addition, preprocessing time series with several *transformations* is also frequently done. This section explains the patterns and transformations used later in this work.

[2]This fraction is called the learning rate.

Patterns

Trend. A trend exists when there is a long-term increase or decrease in the data. A trend does not have to be linear. When a trend goes from increasing to decreasing, or vice-versa, we say the trend "changes direction".

Random Walk. A time serie exhibits a random walk when no apparent pattern can be seen. It looks as the next value is more or less randomly chosen.

Seasonality. Seasonal time series are affected by factors that occur at regular time intervals. This can for instance be connected to a specific time of the year, week, day or hour. For instance, the shopping industry usually has a spike in activity during Christmas times every year. Seasonal patterns are always of a fixed and known frequency.

Transformations

In this section common transformations to time series are defined. For illustration purposes the mock time series $X = [0, 3, 2, 5, 8]$ will be used, where each entry indicate a measurement made at the index time step.

Lag. The value at a time step shifted corresponding to the lag distance. The lag(-2) at the 5th index entry of X is 2.

Rolling Mean. The average of the previous n values. The rolling mean at the 5th index of X with a *window* of $n = 3$ is $\frac{2+5+8}{3} = 5$.

Min-Max Scaling. Scales the entire time serie to lie between a set range $[min, max]$. Each entry is adjusted with the formula $z_i = \frac{x_i - min}{max - min}$. The most common min-max scale is to normalize entries to the range $[0, 1]$ and use $[min, max] = [min(X), max(X)]$. In the case of X this is $[0, 8]$.

2.2 Neural Ordinary Differential Equations

2.2.1 ResNet

The ResNet [He et al., 2015] introduces a new way to propagate the forward signal in neural networks. Up to this point, each layer of a network only has connections to layers directly preceding or superseding itself

$$x_{n+1} = \mathcal{F}(x_n) \tag{2.5}$$

for a forward signal x_{n+1} from layer n with activation function $\mathcal{F}$.

The ResNet however, uses residual connections (hence its name), where layers now incorporate skip connections. This allows the current layer to explicitly use information from several previous layers

$$x_{n+1} = \mathcal{F}(x_n) + x_n \tag{2.6}$$

In addition to activating through the next layer, the input to the previous layer is included directly to the output of that layer as is shown in figure 2.4. Intuitively, shortcut connections explicitly state that the activation of the layer is just an adjustment to x_n. The idea is that it is easier to take a small step in the right direction instead of taking a huge leap across the activation space. Results solidifying this show that ResNets are most efficient when the layer activations are close to the identity function [He et al., 2016].

The new propagation definition and the residual block makes it possible to build even deeper networks, because the vanishing gradient problem [Hochreiter, 1998; Hochreiter et al., 2001] is not as severe as previously. In the vanishing gradient problem the parameter gradients grow too small or too large as the network gets deeper. Before the gradient signal reaches deep levels of the network the magnitude of the loss has decreased/exploded in size. As a consequence the network parameters converge too early or completely explode and diverge. Either way, this causes problems when

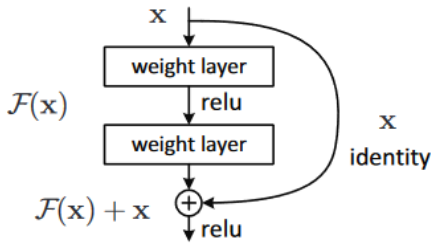


Figure 2.4: A residual block. Input can skip one or several layers. The input to the next layer is the activation from the previous layer and the skip connection. Adopted from He et al. [2015].

training the network, and results in a network with poor accuracy. This phenomenon can be compared to the task of predicting the origin source of the water in an ocean. At the deep end, the source is tiny raindrops falling from the sky up in the mountains. This water gathers in rivers and ends up in the ocean. Knowing the exact origin of each drop in the ocean from the very beginning is thus extremely hard. If the origin predictions are updated based on the error of actual origin, the updates at each step will be very small.

2.2.2 Variational Auto-Encoder

The variational auto-encoder (VAE) [Kingma and Welling, 2013] is a probabilistic machine learning model which approximates the target variables via an unobserved latent state z. It assumes each entry in the observed data X is generated by some random process depending on the latent state. First, a state z is generated from an unknown distribution $p_{\theta^*}(z)$. Then, the observed data sample x is drawn from a distribution conditioned on the latent state $p_{\theta^*}(x|z)$. The problem is neither of the prior distributions, or the joint conditional distributions, are known. The VAE tries to learn these from the provided data. As will be seen, once these are learned the VAE model is able to generate synthetic data. This is possible because the model learns the joint distribution of the observed variable and target variable[3].

The VAE model is characterized by two components; An encoder model, and a decoder model. The encoder Q with parameters ϕ takes an observed input x and outputs a latent state encoding z. This gives rise to the encoder function, also called the recognition function

$$z = Q_\phi(z|x) \tag{2.7}$$

Normally the latent state has a lower dimension than the original input. The intuition for this is that the recognition function should be able to compress the information from the input, extract the important features, and thus learn a more efficient representation. The other component in a VAE is a decoder. The decoder, P with parameters θ takes the latent state z and outputs a decoded state $\tilde{x}$. This leads to the decoder function, also called the generative function

$$\tilde{x} = P_\theta(\tilde{x}|z) \tag{2.8}$$

The primary goal is to create a decoder which is able to recreate the original x encoded by z, where $\tilde{x} \approx x$. A high level representation of the VAE model is shown in figure 2.5.

[3]Machine learning models can be separated into discriminative and generative models [Ng and Jordan, 2002]. Since generative models learn the joint probability distribution $P(X, Y)$ between two variables it is able to *generate* new samples. A discriminative model only learns the conditional probability $P(Y|X = x)$. Thus it omits learning the prior $p(x)$, and is not able to generate likely samples of x.

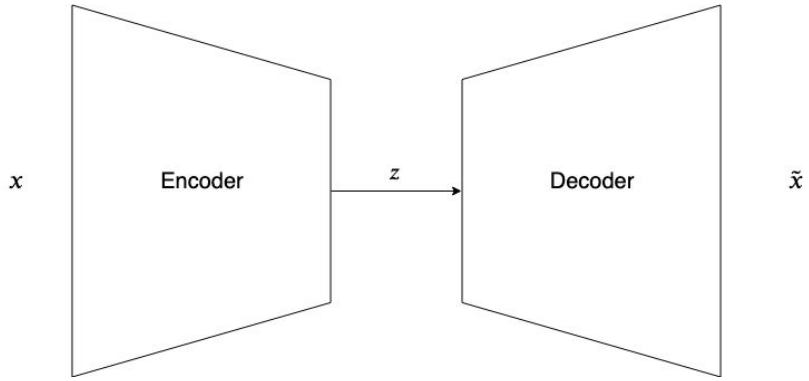


Figure 2.5: High-level structure of a VAE. The input x is encoded to a state z and decoded back to an output $\tilde{x}$. Synthetic samples similar to x can be generated by adding a small amount of noise to z before decoding.

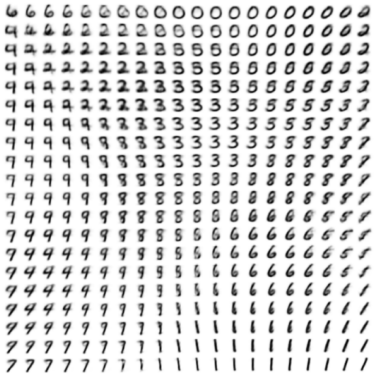


Figure 2.6: An example of VAE results on the MNIST dataset [LeCun et al., 1998] showing the corresponding decoded values of latent state space distribution. Note the gradual change in number values as the latent state changes.

More on Latent State and Generating Samples

In a VAE the latent state is modelled as a set of normal distributions, where each dimension of the latent state is an independent normal. The encoder outputs the mean μ and the standard deviation σ for each dimension. Synthetic samples are generated by sampling from the entire latent state distribution. Since the latent state is explicitly modelled as normal distributions, similar samples are generated when sampling adjacent latent states. An example of generated samples is shown in figure 2.6. This stochastic sampling was invented by two research groups simultaneously [Kingma and Welling, 2013; Rezende et al., 2014].

VAE Loss Function

As previously, the model is trained by optimizing its parameters on a loss function, with greedy gradient based updates. However, the loss now includes a *regularization* term. A regularization term is included to restrict the model properties in a certain way. In the VAE the restriction we want to impose is that the latent state should have a normal distribution, so that latent states can be sampled. Thus, the loss function has one term optimizing predictions of the target, and one term to restrict the latent state distribution. The VAE uses the probability distribution function between the target values and the model predictions as the first term, and the Kullback-Leibler divergence between the latent state distributions and a normal distribution $p(z) = \mathcal{N}(0, 1)$ as the regularization term.

Probability Distribution Function. The first term of the VAE loss function is the probability distribution function, also called the *reconstruction loss*. The probability of observing the *reconstructed* x given a latent state z sampled

from the encoder is maximized

$$P(x) = E_{z\sim Q_\phi}[P_\theta(x|z)] \tag{2.9}$$

Kullback-Leibler Divergence. The Kullback Leibler divergence measures the divergence[4] between two distributions P and Q

$$D_{KL}(P||Q) = -\sum_{x\in\mathcal{X}} P(x)log\left(\frac{Q(x)}{P(x)}\right) \tag{2.10}$$

The final loss function is the sum of the reconstruction loss and the KL-divergence over the training samples N

$$L(\theta,\phi) = \sum_{i}^{N} E_{z\sim Q_\theta(z|x_i)}[P_\phi(x_i|z)] + D_{KL}(Q_\theta(z|x_i)||p(z)) \tag{2.11}$$

2.2.3 Neural Ordinary Differential Equations

In its most general context, Neural Ordinary Differential Equations [Chen et al., 2018] can be though of as a continuous version of the ResNet. The key observation is that equation 2.6 is Euler's formula for solving a differential equation numerically. That is

$$x_{n+1} = \mathcal{F}(x_n) + x_n \tag{2.12}$$

can be rewritten to

$$x_{t+1} = x_t + zf(x_t) \tag{2.13}$$

with a small step, z. In ResNets this z is represented as a layer in the network. NODE take this one step further and make an infinitesimal step $\frac{dz(t)}{dt}$ at each layer. That is, the network models the continuous dynamics of an ODE, where discrete states $z(t)$ are the output of each layer in the net. The function defining the ODE dynamics can be approximated with a function approximator, in our case a neural network, f

$$\frac{dz(t)}{dt} = f(z(t), t, \theta) \tag{2.14}$$

With equation 2.14 and the initial value x_{t_0} we have everything we need to solve the initial value problem

$$\begin{cases} z(0) = x_{t_0} \\ \frac{dz}{dt} = f(z(t), t, \theta) \end{cases} \tag{2.15}$$

[4]Note it is not a distance measure as it is not symmetric.

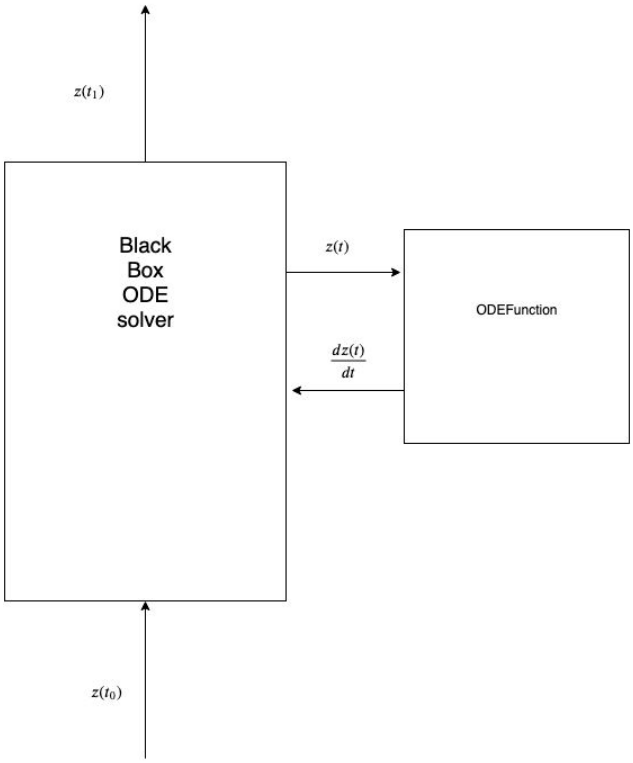


Figure 2.7: The ODE Block consists of two components: A black box ODE Solver, and an ODE Function with trainable parameters defining the dynamics of the ODE solved by the solver.

ODE Block Component

At the core of the NODE model lies the ODE Block component. This is used to solve the initial value problem in equation 2.15. It consists of a black-box ODE Solver, solving for states $z(t)$, and an ODE Function f with trainable parameters θ, specifying the dynamics of equation 2.14. The relation of these two components is shown in figure 2.7. In this thesis, the ODE Function is a neural network. Using an ODE solver leverages mathematical numerical methods leaning on over 120 years of development [Runge, 1895; Kutta, 1901; Hairer et al., 1993], and there exists a number of out-of-the-box software packages implementing these methods. Most ODE solver software can adapt the computation time through a time-error trade off. They are time agnostic, meaning they can solve the equation in the forward as well as the backward time direction. This will especially come in handy when calculating the gradients during training, as we will see in section 2.2.3. In addition, ODE solvers also have the ability to output solutions for several intermediate time steps.

It is worth highlighting that the ODE Solver does not have any trainable parameters itself. Accurate solving of the equation relies on the ODE dynamics given by the ODE Function. During training of the model it is the dynamics of the ODE Function which are learned, and the parameters of this function are adjusted through the chosen optimization algorithm.

ODE Block Instead of a ResNet Component

To get a better understanding of what the ODE Block component does, it will be helpful to compare a NODE model with a simple ResNet. As already stated, the NODE model can be seen as a continuous equivalent to the ResNet, where each layer in the ResNet is one iteration through the ODE Solver in the ODE Block. In contrast to ResNet, the number of iterations is not explicitly specified in the ODE Block. Rather, it is decided by the time-error trade off, and

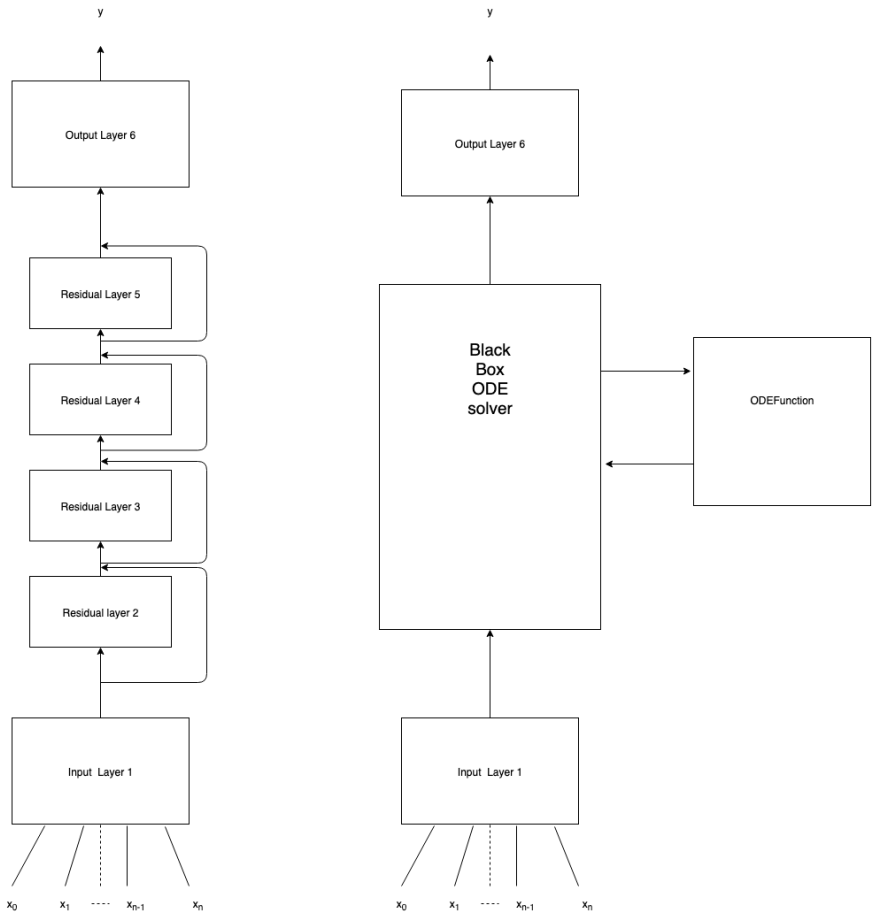


Figure 2.8: All the residual blocks can be substituted with one ODE Block. The depth of the ODE Block is not as explicit as the residual network depth, but an indicator of depth is number of function evaluations through the ODE Function.

the complexity of the unknown dynamics the NODE tries to model. Therefore, as proposed by Chen et al. [2018] a measure of network depth in NODE models is the number of iterations through the ODE Solver. This means that each residual layer in a ResNet is conceptually equivalent to one iteration through the ODE Solver. The logic of this thought process suggests we can substitute multiple residual layers with one ODE Block as shown in figure 2.8.

Latent NODE for Time Series as a VAE

Another variant of NODE, inspired by VAEs, is the Latent ODE. This model is particularly designed to handle time series. As with normal VAEs it has generative properties through sampling the latent state. In addition to the encoder and decoder components of a normal VAE, the Latent ODE has as a third component; an ODE Block. The ODE Block is located in the middle, between the encoder and the decoder as shown in figure 2.9. The ODE Block introduces a dimension of time to the latent state. Think of the latent state as a continuous function that changes over time, and whose dynamics are a function of its current state. With this definition of the latent state the encoder produces the input starting state $z(t_0)$ and the ODE Block outputs the latent state at the desired output step $z(T)$ to the decoder. Remember that the ODE Solver also can output values at intermediate steps. This means the Latent ODE can predict values for all arbitrary times, both forward and backwards from t_0. The decoder decodes the transformed latent states, and outputs the target value at the specified time step. The high level structure of the Latent ODE is illustrated in figure 2.9. The Latent ODE model is trained the same way as a VAE, explained in section 2.2.2.

Backpropagation Through an ODE Solver

Using ODE Blocks in machine learning requires knowledge of how to propagate the gradients through the ODE Solver when optimizing the model. Since the exact number of function evaluations needed by the ODE Solver is not known, the normal approach of storing all the operations of the forward pass through the network can incur a high cost on memory as well as introduce additional numerical error. Therefore, to do backpropagation in the optimization process, NODE models use the adjoint sensitivity method [Pontryagin et al., 1962]. This method computes the gradients by solving an augmented ODE backwards in time through the ODE Solver. It scales linearly with the problem size, and has constant memory cost.

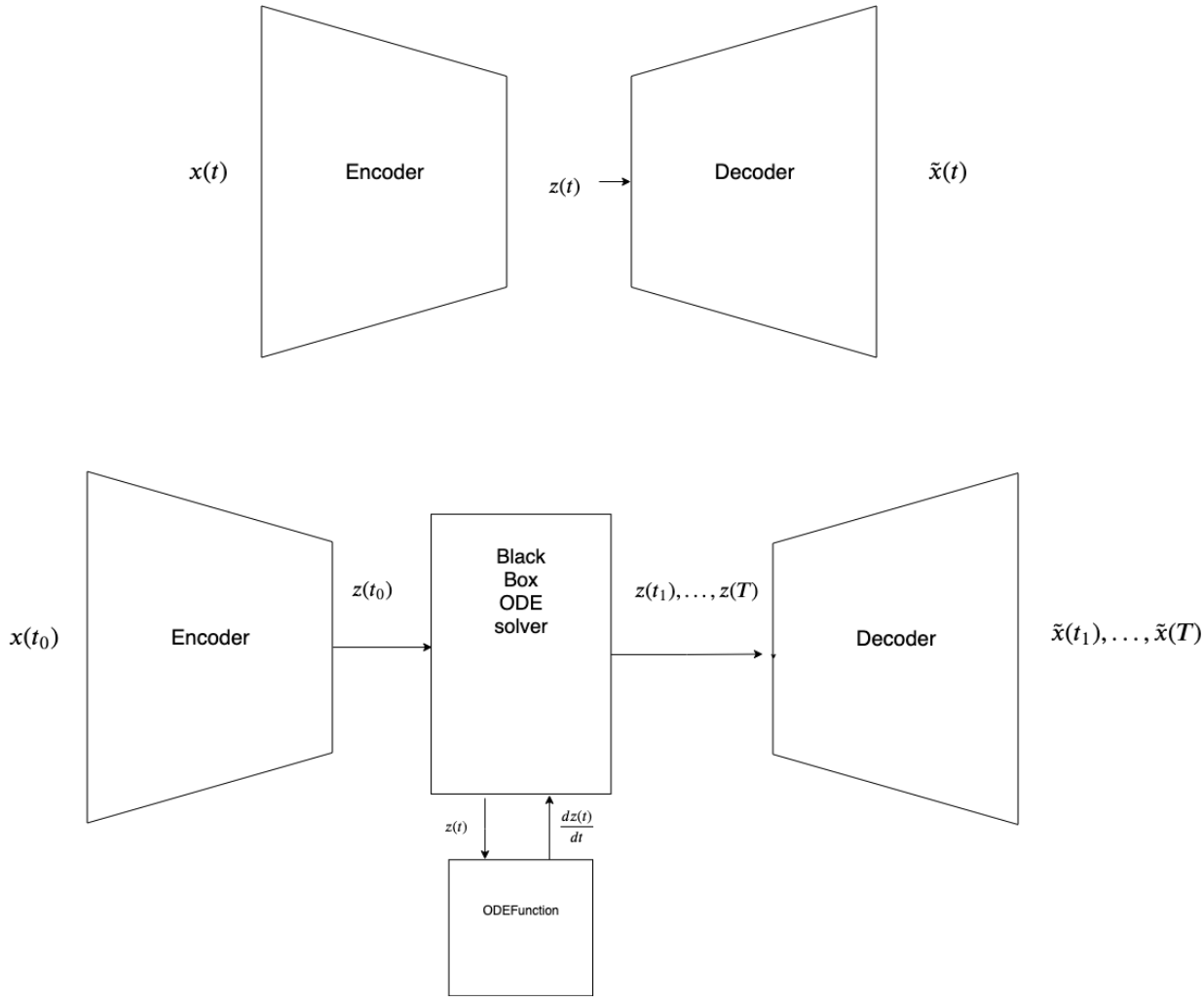


Figure 2.9: Latent ODE (bottom) compared to a vanilla VAE (top). Latent ODE integrates an ODE Block between the encoder and decoder, adding a dimension of time. The VAE does not explicitly keep a time dimension, whereas the ODE Block in the Latent ODE solves the latent state through time. The latent states at different time steps are then decoded.

To explain how the backpropagate through the ODE Solver works, consider optimizing a loss function L whose input is the resulting latent state of the Latent ODE

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t)), t, \theta)dt\right) = L(ODESolver(z(t_0), f, t_0, t_1, \theta)) \tag{2.16}$$

To optimize this function we need to keep track of three different variables. The first two variables are the gradients needed to train the model: (1) The gradients w.r.t. the parameters θ of our dynamics function, which are needed to train the dynamics function. (2) The gradients w.r.t. the input state $z(t_0)$, which are needed to train the encoder. The third variable are the intermediate latent states $z(t)$. Although the intermediate states do not update any parameters directly, they are required by the ODE Solver as auxiliary variables to solve for the gradients. All the needed variables are calculated by solving the latent state backward in time using an ODE solver, starting from the final state $z(t_1)$. At each discrete time step the gradients of the parameters are calculated. Remember, the ODE Solver itself has no trainable parameters.

The first step in backpropagation is finding how the gradients depend on the latent state $z(t)$ at each intermediate time step through the ODE Solver. This is called the adjoint state

$$a(t) = \frac{\delta L}{\delta z(t)} \tag{2.17}$$

How the adjoint state changes over time (its dynamics) is given by another ODE which can be thought of as applying the chain rule to $a(t)$ at a small step dt

$$\frac{da(t)}{dt} = -a(t)^T \frac{\delta f(z(t), t, \theta)}{\delta z} \tag{2.18}$$

$L(z(t_0))$ can now be inferred by solving the initial value problem starting at $L(z(t_1))$ and dynamics given by equation 2.18 backwards in time. Since the dynamics of the adjoint uses the current latent state in its computation, we also need to know this value. However as already mentioned, the ODE Solver is agnostic to time direction, allowing us to compute the value of the latent state backwards, starting from its final value $z(t_1)$.

Finally, to find the gradients with respect to the parameters θ we need to evaluate a third integral. This integral depends on both $z(t)$ and $a(t)$ which we know how to obtain

$$\frac{dL}{d\theta} = \int_{t_1}^{t_0} a(t)^T \frac{\delta f(z(t), t, \theta)}{\delta \theta} dt \tag{2.19}$$

The three values can all be solved with one call to an ODE Solver. A summary of the backpropgation algorithm through an ODE Block can be seen in figure 2.10. For a modern proof of the equations used the reader is referred to the appendices in Chen et al. [2018].

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$
$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state
def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state
return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products
$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Figure 2.10: Algorithm for deriving the gradients through an ODE Block. Adopted from Chen et al. [2018].

2.3 Related Work

There are two avenues of related work. The first one is that of forecasting. As in many other domains, the publicly available[5] state-of-the-art contains a hint of machine learning and neural networks. In the M4 competition [M4-Competition, 2019], a prestigious forecasting competition attracting the largest companies and universities in the world, all the top submissions contained some form of machine learning, with the winner utilizing a hybrid approach between statistical methods and machine learning [Smyl et al., 2019]. In fact, the overall impression was that ensemble methods, using a combination of classical statistical models and machine learning methods, performed better than any of the two alone [Makridakis et al., 2018]. Given the complexity of the best forecasting methods, dissecting and delving into these methods is out of the scope of this thesis. The interested reader is referred to Makridakis et al. [2018]. Instead, we will focus on the second avenue; namely work in and around the vicinity of NODE.

2.3.1 Neural Ordinary Differential Equations

The first paper on NODE [Chen et al., 2018] present a proof-of-concept on the ODE Block. After introducing the theoretical motivation and concepts, the authors demonstrate NODE on several different tasks. First, the ODE Block together with convolutional neural components is tested in a supervised fashion on the MNIST digit classification dataset [LeCun et al., 1998]. Specifically, each of the six standard residual blocks is replaced with one ODE Block. The "ODE-Net" is compared to the ResNet and a second Runge-Kutta network (RK-Net), where the gradients are backpropagated through a Runge-Kutta integrator. The main take-away, shown in table 2.1, is that the ODE-Net has similar performance to the ResNet in terms of test error, while needing about a third of the parameters. This is credited the shared weights in the ODE Function in each ODE Block. Another notable take-away is the constant memory use of the algorithm. This is because the adjoint method can find the gradients backwards in time by only storing the current state of the loss, as opposed to storing all the activations in the forward pass.

	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\mathcal{O}(1)$	$\mathcal{O}(\tilde{L})$

Table 2.1: Performance comparison on MNIST. L denotes the number of layers in the ResNet, $\tilde{L}$ is the number of ODE Function evaluations in a forward pass through the net. Adopted from Chen et al. [2018].

Second, NODE are applied to normalizing flows [Jimenez Rezende and Mohamed, 2015]. In short, normalizing flows are used in machine learning to transform simple distributions to more complex distribution. Why would one want a more complex distribution? Machine learning models in general try to capture the distribution of the training dataset by maximizing the probability of the data. Oftentimes the data does not fit the assumptions of simple distribution (i.e. the assumption of a normal distribution in variational auto-encoders in section 2.2.2). Normalizing flows tackle this by transforming these simple distributions through an invertible bijective mapping. The biggest downside of normalizing flows is that the determinants of the Jacobians (gradient matrix) need to be easy to compute to be of practical use. Else-wise the transformations between the two distributions is too costly. NODE' main contribution to normalizing flows addresses this problem. When moving normalizing flows from the discrete to the continuous space the computation of the transforms are affected, and gives rise to what is coined continuous normalizing flows (CNF). Specifically, a determinant operation is substituted with a trace operation. Where the determinant operation has a cubic cost, the trace operation is linear. In what may be seen as a follow-up paper, Grathwohl et al. [2019] investigate the usability of continuous normalizing flows. Summed up this paper builds on the idea of NODE and CNF to create a continuous-time reversible generative model named Free-Form Jacobian of Reversible Dynamics (FFJORD). They demonstrate this model on a variety of density estimation tasks on both toy and real world data, and for variational inference approximation in VAEs.

Finally, and most importantly for forecasting, is the application of NODE on time series data [Chen et al., 2018]. In a creative way, the ODE Block is used to represent the dynamics of the latent space of a VAE (Latent ODE model

[5]Given the great implications of accurate forecasting, it is not unreasonable to think there exists even better methods not publicly available.

introduced in section 2.2.3). That is, the latent space representation at time t_0 given by the encoder is propagated through time and latent representation at times $t_{1:T}$ can be obtained. These latent representations are then decoded and the observed variables $x_{t:T}$ are obtained. This system is trained the same way as a VAE and has the same generative capabilities. The authors motivation for forecasting comes from prediction of irregularly sampled data such as medical records. A challenge with this is how to handle missing data. Other methods tackle this by putting observations in discrete bins, impute the missing values (i.e. interpolation, smoothing, EM algorithm) [Che et al., 2018], or incorporate the time stamp of the data point as an input to the network [Choi et al., 2016]. The neat thing about moving to continuous space is that the Latent ODE model can make predictions at arbitrary time instances, and thus omits the problem. The Latent ODE model is demonstrated on synthetic spiral data and benchmarked up against an RNN with trajectories shown in figure 2.11. The results in table 2.2 indicate that the Latent ODE model is able to extrapolate smoother and more accurate trajectories than the RNN model.

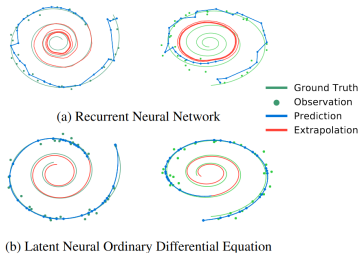


Figure 2.11: Plotted results on sampled clock-wise and counter clock-wise spirals. Adopted from Chen et al. [2018].

# Observations	30/100	50/100	100/100
RNN	0.3937	0.3202	0.1813
Latent ODE	**0.1642**	**0.1502**	**0.1346**

Table 2.2: Root mean square error from the spiral example in figure 2.11 on the test set. Adopted from Chen et al. [2018].

Follow-Up NODE Papers

Another recent paper [Dupont et al., 2019] which builds on Chen et al. [2018] shows that the continuous nature of NODE makes them unable to model certain, simple, functions. Their contribution is the Augmented NODE (AugNODE). AugNODE augment the latent space with extra dimensions. This allows the model to learn the underlying dynamic function using simpler dynamic flows through the latent space. Experimenting on simple regression tasks the authors claim AugNODE models are more expressive, reducing the computational cost of the forward and backward passes, show better generalization to unseen samples, and achieve lower losses with fewer parameters. In a second experiment on image datasets CIFAR-10 and MNIST, AugNODE again reach lower losses than NODE. The second experiment also shows more stable training with fewer passes through the ODE Solver, indicating the dynamics are in fact easier to learn.

A second series of two recent paper, also building on the original NODE paper, discusses properties of NODE [Gholami et al., 2019; Zhang et al., 2019]. The first of the two papers discusses several problems connected to the backpropagation algorithm for ODE Blocks. First, that the approach may be numerically unstable when using ReLu/non-ReLU activations and general convolution operators. The authors argument this instability comes from incorrect assumptions on the constraints when computing a solution backwards through the ODE Solver. Chen et al. [2018] claim that the ODE can be solved backwards in time, as long as the activation functions are Lipschitz continuous[6]. Gholami et al. [2019] show that reversing an ODE is problematic since a reverse ODE is not the same as a forward ODE after a certain time step. This time step is changing relative to the ODE initial conditions, thus the reversibility is not guaranteed for

[6] A Lipschitz continuous function means that the function can not grow faster than a certain threshold. For a function f there exist a real constant $K \leq 0$ so that $|f(x_2) - f(x_1)| \leq K|x_2 - x_1|$.

all time steps. This means that simply running the ODE Block backwards might lead to small compounding errors with each iteration. The conclusion is that Lipschitz continuous functions *could* be reversible in theory, but that in practice there are complications due to instabilities and numerical discretization errors.

Second, the authors discuss problems with the proposed scheme for finding the gradients numerically. The problem arises from the fact that the latent state gradients are derived by the adjoint state in terms of an integral equation[7], instead of computing the gradients directly using the chain rule. This problem is explained as incorrectly replacing input of the neural network with its output. The inconsistency leads to quite large errors for small time steps. The authors call this the Optimize-Then-Discretize (OTD) approach. Further, to address this problem they propose deriving discretized optimality conditions, called the Discretize-Then-Optimize approach (DTO). This approach comes at increased memory requirements.

Motivated by the two problems discussed earlier Gholami et al. [2019] propose the Adjoint NODE (AdNODE). AdNODE uses the DTO approach, has the same computational complexity as the method in Chen et al. [2018], but trades off memory usage. The method uses a checkpointing routine which has a memory footprint of $\mathcal{O}(L) + \mathcal{O}(N_t)$ for a network with L ODE Blocks and N_t total number of runs through the ODE Solvers. The method uses several ODE Blocks and computes the gradients one ODE Block at a time. For each block the input activations are stored during the forward pass. This amounts to the $\mathcal{O}(L)$ memory cost. Then, the backpropagation is performed in multiple stages. Each ODE Block is solved with the saved input activation, and the intermediate values are stored in memory. This amounts to the $\mathcal{O}(N_t)$ memory cost. The stored values are used to solve the adjoint backwards in time using the DTO method. Once computation is finished for an ODE Block, the memory is released for the computation of the next ODE Block. The AugNODE model is tested on the CIFAR-10 and CIFAR-100 dataset, where ODE Blocks replace each residual block in a SqueezeNet [Gholami et al., 2018] (Variant on ResNet). This model is compared with NODE and show more stable and accurate training. Gholami et al. [2019] also report divergent training of NODE when using the more complex RK2 (trapezoid rule) instead of the Euler discretization.

In the second paper in the series, Zhang et al. [2019] present AugNODEv2, a framework to train the parameters and activations of another model. The network extends AugNODE in the direction of neural evolution by introducing a system of coupled ODEs. This coupling allows evolution of both the model parameters and its activations. It is motivated by findings in Gholami et al. [2019], where using more time steps or different discretization schemes do not affect NODE generalization to unseen examples, calling for alternative optimization techniques. This approach is similar to recent Hypernetworks [Ha et al., 2016], where an auxiliary neural network provides the main network with model parameters. Two configurations of AugNODEv2 are used to find parameters and activations of three different convolutional networks. These are tested on the CIFAR-10 dataset. The three networks are compared to a NODE evolution model and baseline models. Results show both AugNODEv2 configurations are able to outperform the other models by $\sim 1\%$ on average.

Stability of ODE Solutions

Reported in all the follow-up papers on NODE is the instability during training. Dupont et al. [2019] address this problem by introducing additional dimensions to the latent state, while Gholami et al. [2019] address problems with the underlying training formulations. There has also been significant work on the stability of ResNets, which we know to be discrete equivalents of NODE. In a recent paper Haber and Ruthotto [2017] discuss stable forward propagation of deep residual networks relating the exploding and vanishing gradients problem to stability of the discrete ODE. Other recent work on stability of residual networks is contained in Ciccone et al. [2018] and Chang et al. [2017]. The general question of stability is related to Lipschitz continuity, where the results indicate the growth between residual layers should be controlled by a threshold constant. Ruthotto and Haber [2018] discuss deep neural networks motivated by Partial Differential Equations (PDEs). The authors show PDEs have properties rendering them robust to perturbations of the initial conditions, which yield certain stability results.

Generally, the stability of discrete ODEs (ResNets) and continuous ODEs (NODE, AugNODE, AdNODE) is not

[7] Recall the adjoint dynamics in equation 2.18, and its definition in equation 2.17. To retrieve the loss gradients w.r.t. the latent state the adjoint is integrated.

equivalent in practice. The work presented by Gholami et al. [2019]; Dupont et al. [2019] indicate that practical application of NODE require stricter constraints to the dynamics of the underlying ODE, than in the discrete case.

3 Experiment Design

This chapter describes the design of the experiments conducted in this thesis. Each experiment is designed to answer one or more of the research questions in section 1.3. It should be noted that the experiments were initially conducted chronologically. Information gained from the first experiment affected the setup of experiment 2 and 3, and the results of experiment 1 and 2 raise new questions. As a consequence new research questions occurred. This entailed new specifications to the environment setup phase explained in section 1.4, which again meant the experiments had to be backtracked for a sound comparison. In this sense, the thesis work also follows an iterative process and not a strict chronological one, as is shown in figure 3.1.

This chapter is built up as follows. In section 3.1 the high-level structure of the NODE models used in the experiments is presented. Section 3.2 explains the experimental plan for the three experiments. This includes how each experiment is designed to answer specific research questions, and which subset of models were used in each experiment. Section 3.3 describes the environmental setup in which the experiments were conducted. It lists the model parameters, inputs and target features in detail. Finally, section 3.4 summarizes the model capabilities in relation to each experiment.

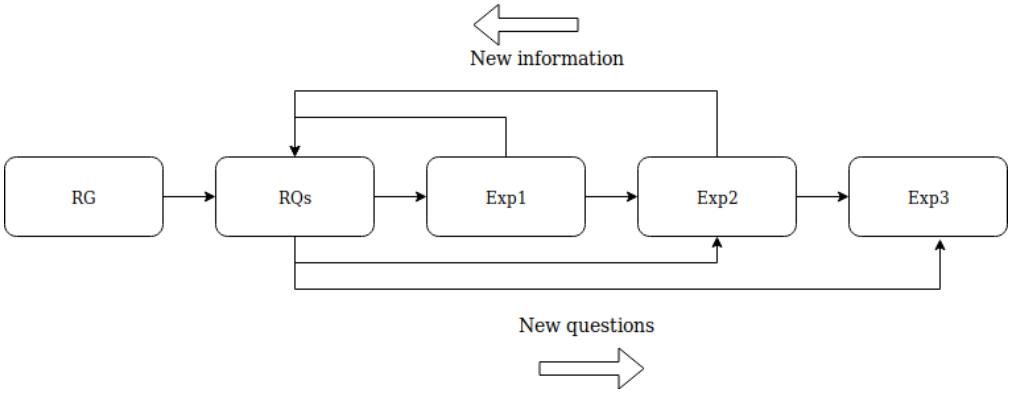


Figure 3.1: Starting from the research goal (RG) the research questions (RQs) have been affected by the chronological execution of the partly independent experiments. The results and information gained after completing an experiment, have had an effect on the final research questions and the following experiments.

3.1 Models

Over the course of the experiments five different NODE models and three different benchmark models are used. In this section the high-level architecture of each NODE model is defined. For specifics on hyperparameters and layers the reader is referred to section 3.2 and appendix A. Recall from section 2.1.2 that all of the models are trained supervised with input/target tuples $(x, y)_t$. However, each model builds these tuples in a slightly different manner. Throughout this section we consider a general time series dataset with available features $\mathcal{F}_t$ at each time step t. Depending on model, this set is decomposed into various input/target tuples. This section operates with consistent time indices S and T where $S \leq 0$ and $T > 0$.

Latent ODE

The Latent ODE in figure 3.2 is the same NODE model as described in section 2.2.3. This model specifies the input features as all the data from $\mathcal{F}$ at times $t = [S, 0]$ and uses the same features at times $t = [1, T]$ as the target features.

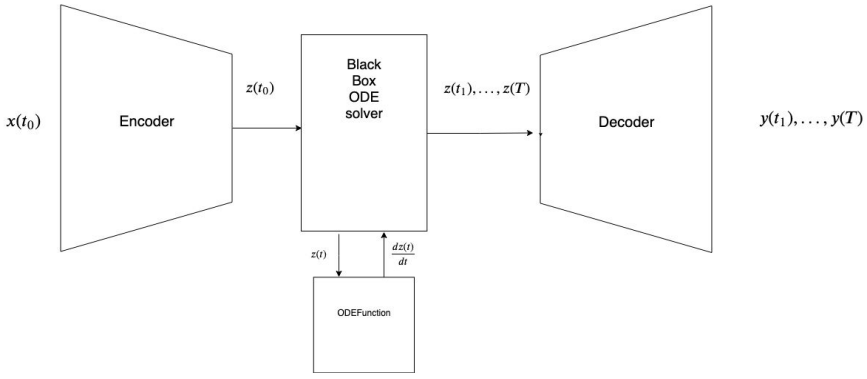


Figure 3.2: The Latent ODE as presented in section 2.2.

The input/target tuples are thus $(\mathcal{F}_{S:t}, \mathcal{F}_{t+1:T})_t$. In the experiments the model has recurrent units in the encoder and fully connected layers in the ODE Function and decoder.

ODE Regressor

The ODE Regressor in figure 3.3 is the same NODE model as described in section 2.2.3. This model divides the features in $\mathcal{F}_t$ into two mutually exclusive subsets $\mathcal{X}_t$ and $\mathcal{Y}_t$ and generates the input/target tuples $(\mathcal{X}_t, \mathcal{Y}_t)_t$. In the experiments the model uses fully connected layers in the ODE Function, input, and output layers.

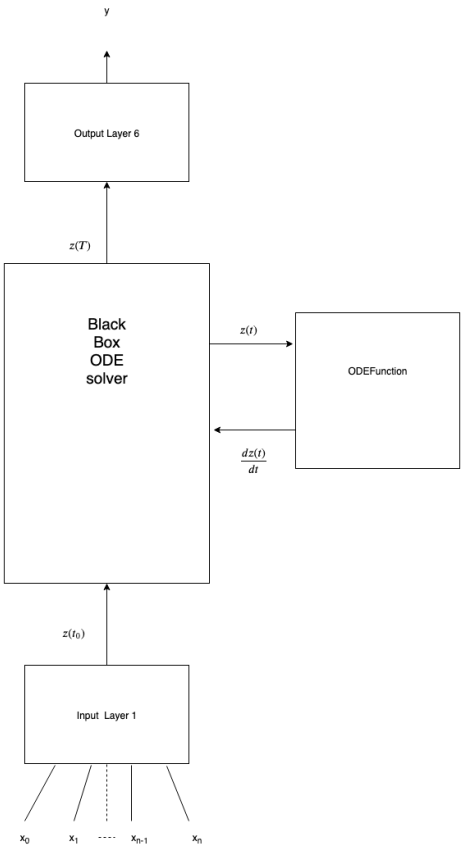


Figure 3.3: The ODE Regressor as presented in section 2.2.

Latent ODE Regressor Version 1

The Latent ODE Regressor v1 shown in figure 3.4 is a slight alteration to the Latent ODE. This model specifies the input features as all the data from $\mathcal{F}$ at times $t = [S, 0]$, but as opposed to the Latent ODE uses only a subset $\mathcal{Y} \subset \mathcal{F}$ of the features at times $t = [1, T]$ as the target features. The input/target tuples are thus $(\mathcal{F}_{S:t}, \mathcal{Y}_{t+1:T})_t$. In the experiments the model uses recurrent units in the encoder and fully connected layers in the ODE Function and decoder.

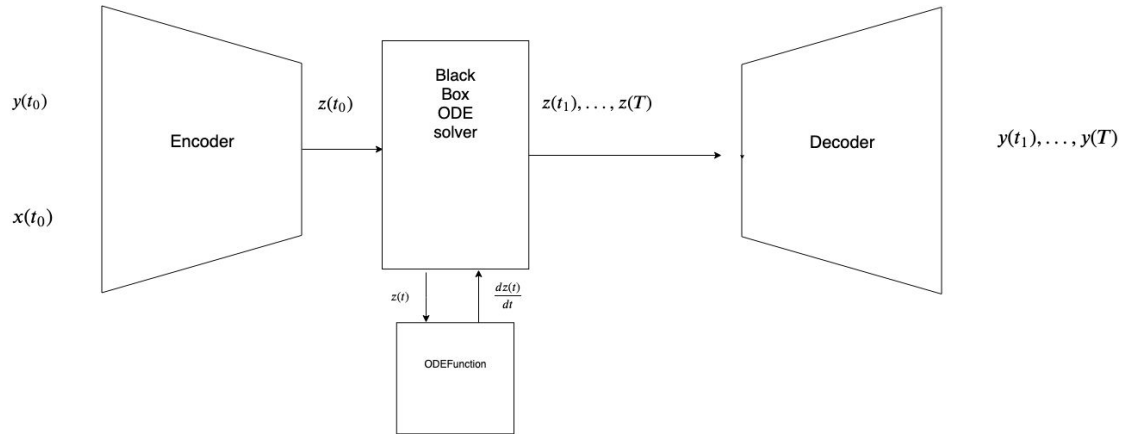


Figure 3.4: The Latent ODE Regressor v1 is an alteration of the Latent ODE. This model only outputs the specified target feature.

Latent ODE Regressor Version 2

The Latent ODE Regressor v2 in figure 3.5 is a slight alteration to the Latent ODE Regressor v1. In this model the decoder is expanded with an additional data input layer, as opposed to giving this data as input to the encoder. Now there is a data input layer to the encoder and a second one to the decoder. The model splits $\mathcal{F}_t$ into two mutually exclusive subsets $\mathcal{X}_t$ and $\mathcal{Y}_t$, and specifies the input features to the encoder as the data from $\mathcal{Y}$ at times $t = [S, 0]$. The data input to the decoder are the features from $\mathcal{X}_t$ at times $t = [1, T]$. The target are the features from $\mathcal{Y}_t$ at times $t = [1, T]$. The input/target tuples are thus $((\mathcal{Y}_{S:t}, \mathcal{X}_{t+1:T}), \mathcal{Y}_{t+1:T})_t$. The experiments use recurrent units in the encoder and fully connected layers in the ODE Function and decoder.

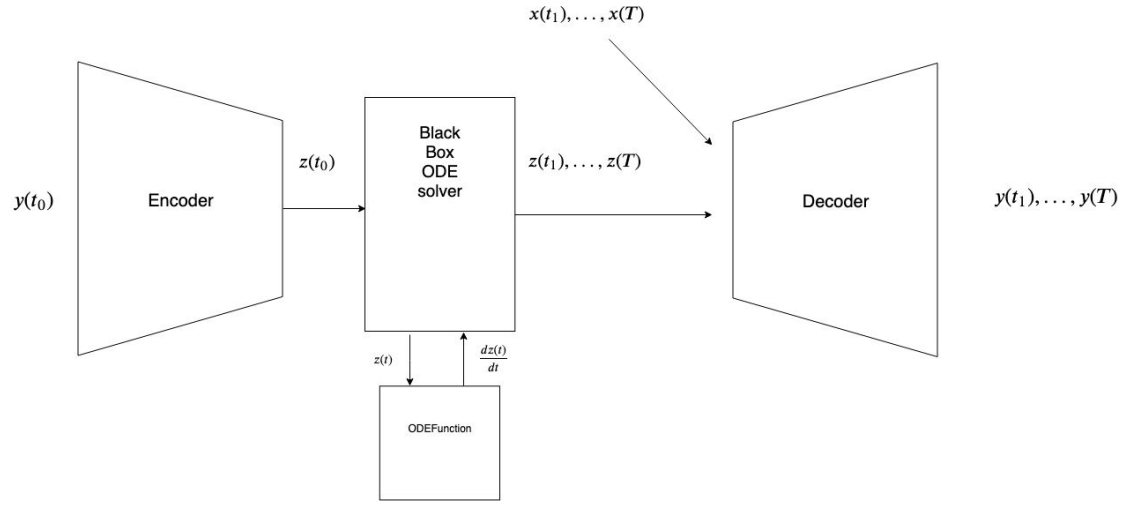


Figure 3.5: The Latent ODE Regressor v2 is an alteration of the v1 model. This model moves the input of the observed features from the encoder to the decoder.

Latent ODE Regressor Version 3

The Latent ODE Regressor v3 in figure 3.6 is a slight alteration to the Latent ODE Regressor v2. In this model the encoder is provided additional data. Now the data input to the decoder is also input to the encoder at the corresponding time step. The model splits $\mathcal{F}_t$ into two mutually exclusive subsets $\mathcal{X}_t$ and $\mathcal{Y}_t$, and specifies the input features to the encoder as the data from $\mathcal{X}_t$ and $\mathcal{Y}_t$ at times $t = [S, 0]$. The data input to the decoder are the features from $\mathcal{X}_t$ at times $t = [1, T]$. The target are the features from $\mathcal{Y}_t$ at times $t = [1, T]$. The input/target tuples are thus $((\mathcal{X}_{S:t}, \mathcal{Y}_{S:t}, \mathcal{X}_{t+1:T}), \mathcal{Y}_{t+1:T})_t$. The experiments use recurrent units in the encoder and fully connected layers in the ODE Function and decoder.

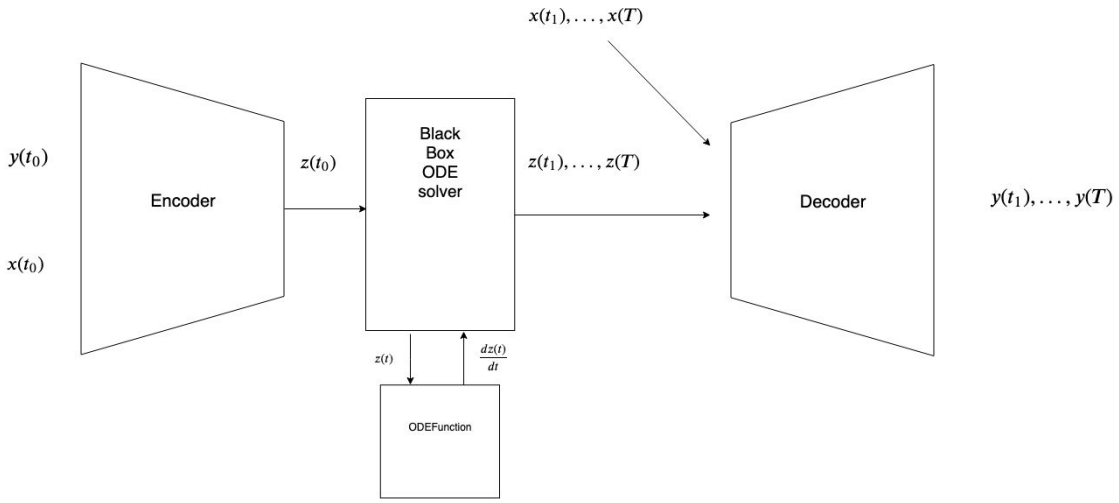


Figure 3.6: The Latent ODE Regressor v3 is an alteration of the v2 model. This model also inputs observed features to the encoder.

3.2 Experiment Plan

The experiments test the NODE models presented in section 3.1 on three different problems presented in section 2.1.1. To evaluate the NODE models they are compared to the existing forecasting models at TrønderEnergi presented in section 2.1.3 and suiting baseline models. A common baseline model is the persistence model[1], which will be used in the experiments. The three problems are:

P1: The first problem is intraday one hour ahead forecasting of windmill power production for the windmill park in Ytre Vikna, Trøndelag, Norway.

P2: The second problem is one hour ahead forecasting of energy consumption in TrønderEnergi's office building "Energibygget" located at Lerkendal, Trondheim.

P3: The third problem is day-ahead forecasting of energy consumption in the NO3 region located in central Norway (figure 2.2).

In each experiment one or more of the research questions are targeted, and a different subset of models is used. An overview of the correspondence between experiment and models, and experiment and targeted research questions can be seen in table 3.1 and table 3.2.

[1] A persistence model predicts the value at the current time step to be the same as a value at an earlier time step. For instance, a one-step persistence model predicts the value to be the same value as the previous step.

	E1	E2	E3
Latent ODE	x	x	
Regressor ODE	x	x	
Latent ODE Regressor v1	x		
Latent ODE Regressor v2		x	
Latent ODE Regressor v3			x
LSTM	x	x	x
DNN	x	x	
Gradient boosting	x	x	

Table 3.1: An overview of which models from section 3.1 will be used in which experiments.

	E1	E2	E3
RQ1	x	x	x
RQ2	x	x	x
RQ3	x		
RQ4		x	
RQ5			x

Table 3.2: Each experiment is designed to answer specific research questions.

3.2.1 Experiment 1: Intraday One Hour Forecast of Windmill Power Production

In the first experiment the research target is to (1) compare accuracy of NODE models on the specific problem against themselves and existing models at TrønderEnergi. (2) Investigate the claim of NODE needing less training data to achieve the same accuracy as other models.

To address (1) each model is trained with the same data, and MAE on an equivalent test set across each run is reported. As stability of the NODE solution is a known problem [Gholami et al., 2019; Dupont et al., 2019], the loss from NODE models is also reported. To address (2) the experiment is run varying the amounts of data in the training set, keeping the model parameters, the input features and the test set the same. This way it will be clearly seen how the training data affects the final accuracy. Specifically, the experiment runs three different sizes of training sets:

- 1 month
- 6 months
- 12 months

The test set is the 8 months following the end of the 12th month, with a gap of one month between the end of the 12th month and the start of the 8 month period. Also for the one and six month periods the experiment uses the data in the months closest to the test months. Figure 3.7 clarifies this. In general, data closer in time gives more information about the target feature. Therefore, to compare the accuracy provided by additional data, the additional data should be added to what is assumed to be the most valuable data. This way it is easier to evaluate whether the difference in accuracy is due to additional data, or simply due to more relevant data. Also, to ascertain the increase in accuracy is related to the amount of available data, there has to be reference points measuring the relative improvement with more data. Only this way will it be seen whether the improvement in accuracy is due to more data or due to other factors, such as model properties.

The baseline for this problem is a two-step persistence model. Recall from section 2.1.1 this is the latest available value, as the one-step value is not available at the time of prediction.

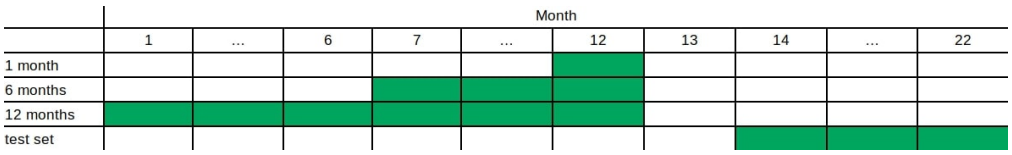


Figure 3.7: Intervals of the 1, 6, and 12 months datasets used in experiment 1. Green color indicates data in the set. The data closest to the test set is used first.

3.2.2 Experiment 2: One Hour Ahead Forecast of Energibygget Office Building Energy Consumption

In the second experiment the research target is to (1) compare accuracy of NODE models on the specific problem against themselves and existing models at TrønderEnergi. (2) Investigate the claim of NODE needing less trainable parameters to achieve the same accuracy as other models.

To address (1) each model is trained with the same data, and MAE on an equivalent test set across each run is reported. The NODE models' loss is also reported. To address (2) the experiment is run varying the model parameters, but keeping the input features, training set, and the test set the same. This way it will be clearly seen how the number of parameters affect the final accuracy. Specifically, the experiment runs three different model variations, categorized into ”small”, ”medium”, and ”large”. It should be noted the models, by nature, have very varying number of parameters. Therefore, simply measuring the total parameter count will not answer the research question entirely. This categorization makes it easier to compare how varying the number of parameters affect the accuracy.

For this problem the baseline is a one-step persistence model. Recall from section 2.1.1 the consumption is continuously measured by a smart sensor which is connected to the internet, and values from the previous time step are available.

3.2.3 Experiment 3: Day-Ahead Forecast of NO3 Region Energy Consumption

In the third experiment the research target is to (1) compare accuracy of NODE models on the specific problem against themselves and existing models at TrønderEnergi. (2) Investigate what unique NODE features of the Latent ODE Regressor v3 model are most important for accuracy.

The third problem stands out in that the forecasts are further forward in time. However, (1) is still addressed the same way as previously.

During the run of this experiment, I noticed behaviour different from what I expected. This raised a new question about the Latent ODE Regressor and the role of the latent state. One could argue the Latent ODE Regressor v3 is equivalent to the ODE Regressor with an ODE Block providing the prediction for a lagged latent (target) feature. This makes the only difference between the two models the loss training function. It seemed that the latent state actually made the performance worse. Therefore to address (2), after running the NODE model with a normal latent state and loss function, the model was also run with a small latent state and a loss function which does not use KL-divergence.

During the experiment, a surprising result between the 1st hour ahead and the 24th hours ahead was discovered. The LSTM had a significantly better performance for the 1st hour, while the NODE had a significantly better performance for the full 24 hours. To further investigate this, an evaluation at the 24th hour of prediction was also included in the experiment.

For this problem the baseline is a one-step persistence model for the 1st hour, and a 24-step persistence model for the 24th hour. No baseline is used for the full 24 hours.

3.3 Experiment Setup

This section describes the environment in which the experiments were conducted.

3.3.1 Databricks Environment

The experiments were run in the Databricks [Microsoft, 2019] environment. This is a cloud based solution in the Azure stack. All the experiments where run on a cluster with the following settings:

- Cluster mode: Standard
- Databricks Runtime Version: 5.3 ML (include Apache Spark 2.4.0, GPU, Scala2.11)
- Python version: 3
- Number of Workers: 0
- Driver Type: Standard_NC12 (beta), 112.0 GB Memory, 2GPUs, 3 DBU

3.3.2 Model Implementations

The following libraries were used for implementing the models:

- All the NODE models were implemented based on the torchdiffeq library [Chen, 2019].
- The LSTM and DNN model implementations were provided by TrønderEnergi, built with the Keras API [Keras-Team, 2019], a high level neural-network API.
- The gradient boosting model implementation was provided by TrønderEnergi, built using the Catboost library [Dorogush et al., 2018].

3.3.3 Model Parameter Settings

NODE Models

In the experimental setup phase I noticed the NODE models were highly unstable during training. Therefore, all the NODE models were run three times. Each run trained for 2000 iterations with a learning rate of 0.001, and the model with the best loss on the test data was chosen for visualization. The models were trained with the loss functions specified previously. In addition to this, the models were also evaluated (but not trained) using MAE for a sound comparison across models.

Across all experiments, the ODE Regressor model has only linear layers with ReLU/ELU activation functions, while the width and number of layers is varied. For exact details on the models the reader is referred to appendix A.

The Latent ODE and Latent ODE Regressor versions also use similar models across experiments. All the models have one recurrent unit in the encoder, where the number of hidden units is varied across experiments. The ODE Block and the decoder has linear layers with ReLU/ELU activations. Across experiments the size of the latent state is half of the observed input features, expect where there is only one observed feature (Latent ODE Regressor v2) or when the latent state size is explicitly tested (E3). For exact details on the layers in the models, the reader is referred to .

Benchmark Models

During the experimental setup phase the benchmark models for E1 and E2 were observed to be much more stable than the NODE models. Therefore, they were run only once. The LSTM in E3 had more unstable training (manifesting in

volatile results) and was run three times. Both the DNN and LSTM models adopt early stopping when the loss stops improving. For E1 the ”medium” models from E2 were used. Next, each model is described.

DNN. The DNN models are trained for 1000 iterations, with a batch size of 1024, and a learning rate of 0.001. The network configurations ”large”, ”medium”, and ”small” had the following hidden layers with the digits indicating the number of nodes in the hidden layer:

- ”large”: [16, 12, 8, 8, 8, 8, 4]
- ”medium”: [16, 8, 8, 8, 4]
- ”small”: [8, 8]

Gradient boosting. The ”medium” gradient boosting model uses 2000 trees with a maximum depth of 6 for each tree. The learning rate is chosen adaptively by the Catboost library, but seems to be set to 0.030 for all models. The model configurations ”large”, ”medium”, and ”small” were:

- ”large”: 2000 trees max depth 10
- ”medium”: 2000 trees max depth 6
- ”small”: 2000 trees max depth 3

LSTM. The LSTM models use the same high-level encoder-decoder architecture as the Latent ODE Regressor v3. The encoder consists of two hidden layers and the decoder consists of two hidden layers. The models are trained for a maximum of 1000 iterations, a batch size of 256, and with a learning rate of 0.001. The network configurations ”large”, ”medium”, and ”small” had the following number of LSTM units in the hidden layers ([encoder; decoder] format):

- ”large”: [24, 24; 24, 24]
- ”medium”: [16, 16; 16, 16]
- ”small”: [8, 8; 8, 8]

For E3 a similar model with 18 LSTM units was used:

- E3 LSTM: [18, 18; 18, 18]

3.3.4 Data

All the data was provided by TrønderEnergi. These data files are proprietary company information and have not been published. All the data was preprocessed with min-max scaling and entries with missing data points were removed. The data is discretized into hourly values. Each experiment uses data from the following dates (YYYY-MM-DD HH format):

E1: Training data is between 2017-12-01 00 and 2018-03-05 00.
Test data is between 2018-04-06 00 and 2018-12-31 00.

E2: Training data is between 2018-01-01 00 and 2018-08-31 00.
Test data is between 2019-02-01 00 and 2019-04-01 00.

E3: Training data is between 2016-11-01 01 and 2017-11-01 23.
Test data is between 2018-02-01 00 to 2018-02-28 23.

Integrating Sequential Information

Sequential information across time is integrated in two different ways. The first method is to concatenate information from several time steps into the same time step. The second way is to input several time steps at the same time. In models using recurrent units s.a. the Latent ODE or LSTM the second method manifests as recurrent activations and the information is propagated through the hidden state. In non-recurrent models s.a. the ODE Regressor or DNN this

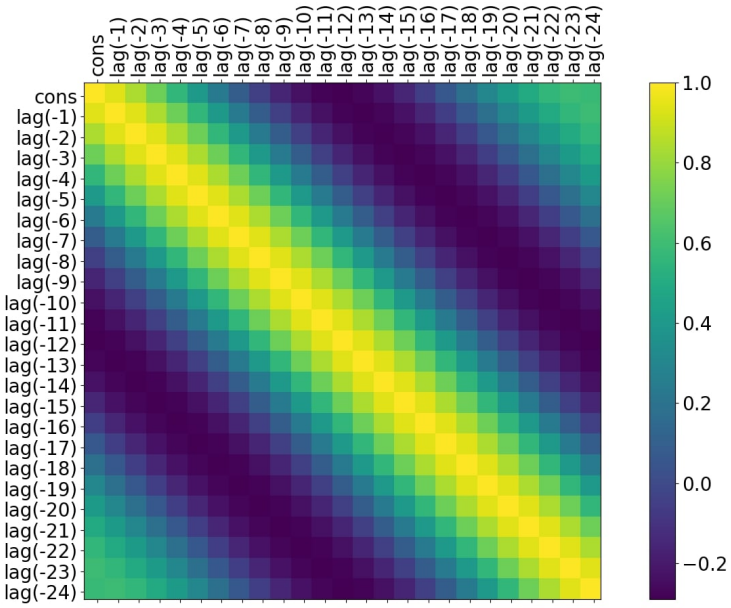


Figure 3.8: Correlation between lagged values of the consumption feature.

manifests as a larger input space, and thus an increase in model parameter. These two methods are summarized by the notation of a tuple $(x_t, y_t)_t$. The inner time stamps x_t and y_t corresponds to the first method, while the outer time stamp on the entire tuple corresponds to the second method.

The experiments incorporate both of these methods for integrating sequential information. Information of used features at each time step (inner features) is described in the next sections. When it comes to the second method (outer tuple), entries from the last S time steps are used, and predictions for the next T time steps are made (recall these are the same indices specifying tuples in 3.1). For E1 and E2 $S = -2$, while for E3 $S = -3$. Likewise, $T = 1$ for E1 and E2, and $T = 24$ for E3.

Input and Output Features

The input features were chosen on recommendation from TrønderEnergi, and were not experimented with. In general, there is a trade-off between adding more features and the speed of training. Therefore, it is desirable to only use features which add value to the predictions. A simple measure for feature value is correlation between the observed and target feature. Figure 3.8 shows correlation between different lagged consumption values used in E2. The plot indicates values closer to the current time step are more correlated. This pattern is equal for all lagged values, relative to each other, as well. Also note the cyclic pattern for values at a 24 hour lag.

Experiment 1: In E1 each x_t uses the 26 observed features in table 3.3, and each target y_t and baseline uses the features also shown in table 3.3.

Experiment 2: In E2 each x_t uses the 23 observed features in table 3.4, and each target y_t and baseline uses the features shown in table 3.5.

Experiment 3: In E3 each x_t uses the 68 observed features in table 3.6, and each target y_t and baseline uses the features shown in table 3.7.

Feature	Type	lag range
production	float	[-3, -2]
wind speed source 1	float	[-3, 2]
wind direction source 1	int	[-3, 2]
wind speed source 2	float	[-3, 2]
wind direction source 2	int	[-3, 2]

	Feature	Type	lag range (hours)
Target	production	float	[0, 0]
Baseline	production	float	[-2, -2]

Table 3.3: The observed features (left) at each time step x_t, and the target and baseline features (right) at each time step y_t in E1.

Feature	Type	lag range (hours)
Consumption one week earlier	float	[-24 * 7, -24 * 7]
Consumption rolling mean, one week earlier	float	[(-24 * 7) - 3, -24 * 7]
Consumption, 24 hours earlier	int	[-24, -24]
Consumption rolling mean, 24 hours earlier	float	[-27, -24]
Consumption rolling mean, 1 hour earlier	float	[-4, -1]
Consumption	float	[-4, -1]
Consumption difference	float	[-4, -1]
Consumption difference one week earlier	float	[(-24 * 7) - 3, -24 * 7]
Hour of day	int	[0, 0]
Day of week	int	[0, 0]
Workday	boolean	[0, 0]
Weekend	boolean	[0, 0]
Holiday	boolean	[0, 0]
Outside temperature	float	[0, 0]

Table 3.4: The observed features at each time step x_t in E2.

	Feature	Type	lag range
Target	Consumption	float	[0, 0]
Baseline	Consumption	float	[-1, -1]

Table 3.5: The target and baseline features at each time step y_t in E2.

Feature	Type	lag range (hours)
Consumption at previous hour	float	[-1, -1]
Consumption difference between last two hours	float	[-1, -1]
Holiday	bool	[0, 0]
Weekend	bool	[0, 0]
One-hot-encoding of month	binary vector	[0, 0]
One-hot-encoding of weekday	binary vector	[0, 0]
One-hot-encoding of season(winter, spring, summer, autumn)	binary vector	[0, 0]
One-hot-encoding of hour	binary vector	[0, 0]
Temperature from 17 locations in NO3 region	vector of floats	[0, 0]

Table 3.6: The observed features at each time step x_t in E3.

	Feature	Type	Target range	Baseline range
1 hour	Consumption	float	[0, 0]	[-1, -1]
24 hours	Consumption	float	[23, 23]	[-1, -1]*
next 24 hours	Consumption	float	[0, 23]	N/A

Table 3.7: The targets and baseline features at each time step y_t for the three evaluations in E3. *Note the baseline is in relation to the target. This means the baseline value is a 24 hour persistence model.

3.4 Predictive Properties of Models

To wrap up this chapter the predictive properties of all models, in relation to the available data in all of the experiments, are summarized in one diagram shown in figure 3.9. This diagram has a lot of embedded information, and the reader is encourages to spend some time to fully understand the figure. Note that each of the models has certain properties which affect what data can be used in the predictions, as well as what can be predicted. What data can be used for prediction will impact the accuracy of the model. Based on these properties the models are divided into *time-agnostic* and *time-aware* models.

The time-aware models are LSTM, Latent ODE, and all Latent ODE Regressors. These models have recurrent units with a hidden state representing the past input. The time-aware models accept a varying number of time step inputs and are in this sense flexible. Being time-aware models they are able to give predictions for multiple time steps.

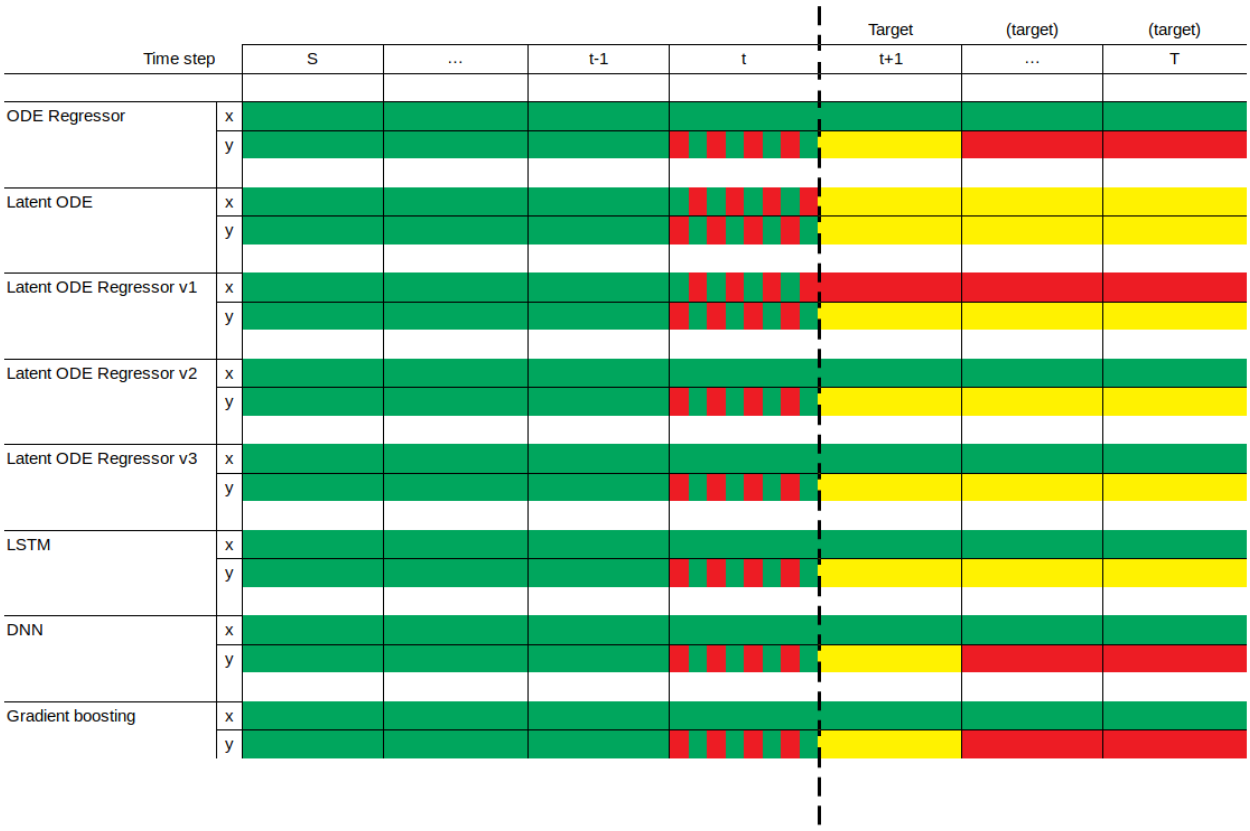


Figure 3.9: Each of the models have inherent properties when it comes to using available data at different time steps. Solid green color means the data is available to the model for prediction. Solid yellow means the data can be predicted. Solid red means the data can not be predicted. Green and red stripes means the data is only available for certain problems. In P1 data at time t is not available, but for P2 and P3 it is. Yellow and green stripes means that the data can be predicted and then reused for later predictions. This is especially useful for problem P3 where we want to predict values at time steps up to T. Note that the ODE Regressor, DNN, and gradient boosting models could be altered to also predict for time steps greater than $t + 1$. However, this would require multiple calls to the model. The value at the current time step would first have to be predicted, then input back to the model for the next time step. To show this limitation, the models are shown in red.

The time-agnostic models are ODE Regressor, DNN, and gradient boosting. These models do not keep an explicit hidden representation which changes over time, and are thus considered more rigid than time-aware models. Being time-agnostic models they are not able to give predictions for multiple time steps. This said, it should be noted that time-agnostic models can integrate input and output from several time steps. As already mentioned, this can be done by

expanding the input/output dimensions with more parameters. However, once the input size is set for the time-agnostic models, it can not be varied.

4 Results

This chapter presents the results of the experiments planned in the previous chapter. The first section elaborates on noteworthy aspects of the models to keep in mind when studying the results. Each experiment has been dedicated its own section. Throughout this chapter you will see several plots of forecasts from different models. All plots showing time series have a discrete time step of one hour. The plots use consistent coloring with green, red, and blue corresponding to target, baseline, and model forecasts. Each section starts with summarizing the results from all models, followed by the results from individual models against the targeted research question. By the nature of this work the number of plots quickly grows large. To keep this chapter uncluttered, a number of these plots have been moved to the appendix. The reader is referred to appendix B for additional results.

4.1 Comperance of Models

The main research target is to investigate the performance of NODE in terms of accuracy. However, when comparing the models there are some additional properties which should be kept in mind for a reflected comparison. The first comparison is the time-agnostic vs. time-aware property introduced in section 3.4. In addition to this it should be noted the models have a very varying number of trainable parameters. Therefore, each NODE model should be specifically compared against its equivalent benchmark model.

Comperance on Time

The time comperance is a division on whether the model has an explicit representation of the time dimension. The time-aware models are LSTM, Latent ODE, and all Latent ODE Regressors. The time-agnostic models are ODE Regressor, DNN, and gradient boosting.

Comperance on Model Size

In addition to the time-agnostic and time-aware comparison, it also makes sense to compare the models on the number of trainable parameters. In this comparison each NODE model should be paired with at least one benchmark model. The number of parameters is related to the overall model architecture, which in terms is also related to the time comparison. Therefore, the parameter comparison is in many ways similar to the time comparison. In general, DNN and ODE Regressor have a similar number of parameters, and the Latent ODE, Latent ODE Regressors and LSTM have similar number of parameters.

The gradient boosting model is harder to fit into this comparison. With the library used, it was not possible to extract the exact number of parameters. Also, the training algorithm does not train parameters the same way as stochastic gradient descent. However, if we define each node in a decision tree as a trainable parameter we can make an estimation of the number. In general, a gradient boosting algorithm with N trees and maximum tree depth d has $\mathcal{O}(N2^d)$ nodes. The true number is likely lower. Depending on the data used, the decision tree might terminate before the maximum depth is reached, or building of additional trees might halt because the accuracy stops improving.

4.2 Experiment 1: Ytre Vikna Wind Production

In the Ytre Vikna wind production forecasting problem the goal is to predict the total hourly production of energy one hour ahead. In addition to this, the experiment is focused on answering whether NODE models can reach the same accuracy as non-NODE models, with less training data. Each model was trained with one, six and twelve months of

data. The models were then tested on the following eight months. Each NODE model was tested three times on the respective variation of data, and plots from the best model were saved for visualization. Error from the benchmark models is also reported. These models showed significantly less variation during the experimental setup phase and were only run once each. The baseline for this problem is a one hour persistence model.

4.2.1 Overview of Results

We start by looking at results for all models, shown in table 4.1 and visualized in figure 4.1. Observe the ODE Regressor has the lowest average for 1 month, while gradient boosting has the lowest for 6 and 12 months.

Further, observe the time-agnostic models have a lower loss across all variations of data, and that time-aware NODE models struggle to reach errors below the baseline.

Comperance		DNN	Catboost	LSTM	Baseline
	# input variables	26	26	26	N/A
	# params	841	2000 trees depth of 6	8145	N/A
1 month	test mae run1	4.2351	2.9220	3.0550	3.0624
6 months	test mae run1	2.3186	2.2878	2.4970	3.0624
12 months	test mae run1	2.2904	2.2288	2.4147	3.0624

Comperance		Latent ODE	ODE Regressor	Latent ODE Regressor v1
	# input variables	26	26	26
	# params	8389	729	8093
1 month	test mae run1	3.3078	2.9096	4.0733
	test mae run2	3.0314	2.7123	5.6400
	test mae run3	3.5547	2.9813	4.7911
	avg mae	**3.2980**	**2.8677**	**4.8348**
6 months	test mae run1	3.5920	2.3568	3.9802
	test mae run2	3.5852	2.3783	4.5011
	test mae run3	3.6138	2.4108	3.6404
	avg mae	**3.597**	**2.3820**	**4.0406**
12 months	test mae run1	2.9949	2.3561	4.8679
	test mae run2	2.9394	2.3668	3.3798
	test mae run3	3.1617	2.3779	5.2030
	avg mae	**3.0320**	**2.3669**	**4.4836**

Table 4.1: MAE (MW) of benchmark models (left) and NODE models (right).

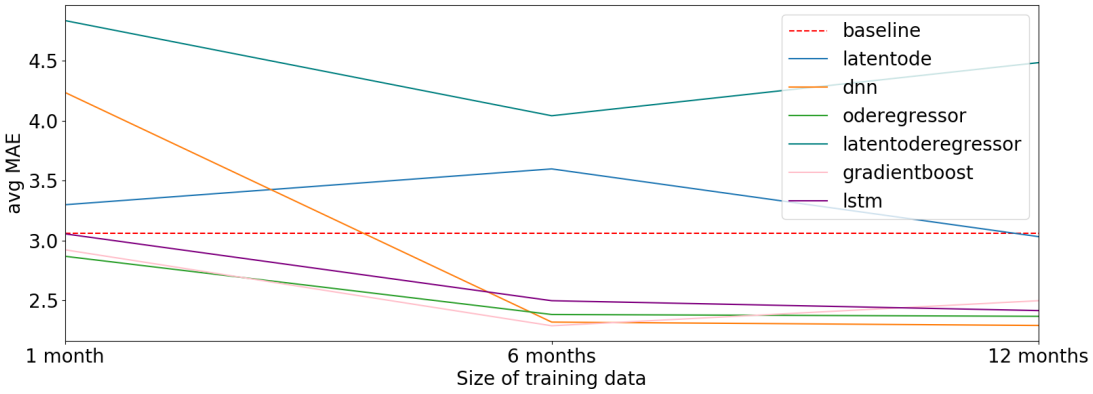


Figure 4.1: Summary of table 4.1 showing average loss (MW) of all models over training data size. All top performing models have a steady decrease in accuracy, with the exception of DNN.

Variations on Data

Except for the time-aware NODE models we see a similar performance across variations of training data for all models. The general trend is that the accuracy increases with increased data. However, the ratio of this increase is similar for each of the top-performing models, except for the DNN model. The DNN model has a much worse performance on 1 month, but catches up to the other models at 6 and 12 months.

4.2.2 Separate Model Data Variations

Next, the attention turns to the results of each separate model, studying the predicted trajectories in detail.

Latent ODE

The Latent ODE model had one of the worst performances and beat the baseline only at 12 months, barely. The trajectories in figure 4.2 are able to smoothly estimate the target, but are missing out on finer-grained details. The model shows improved accuracy with 12 months of training data, but shows worse accuracy going from 1 to 6 months.

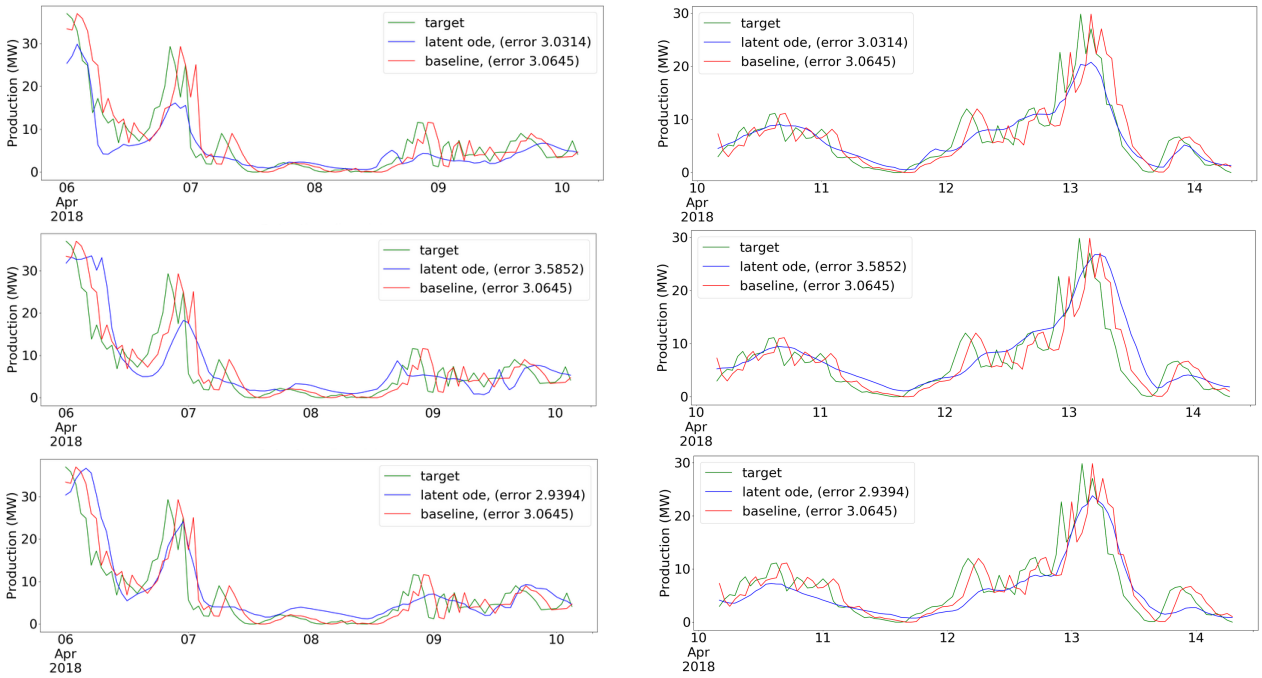


Figure 4.2: Sampled Latent ODE predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

ODE Regressor

The ODE Regressor has the lowest error among the NODE models, reaching errors below baseline value on each variation. A sample of predicted trajectories can be seen in figure 4.3. For the 1 month plot there are seemingly random spikes in the prediction. Further, there is a significant improvement in error from 1 month to 6 months. From 6 to 12 months the error does not improve significantly, which is also consistent with the values in table 4.1. The ODE Regressor model is sometimes able to predict values better than the baseline, while at other times it seems it predicts the same value as the baseline.

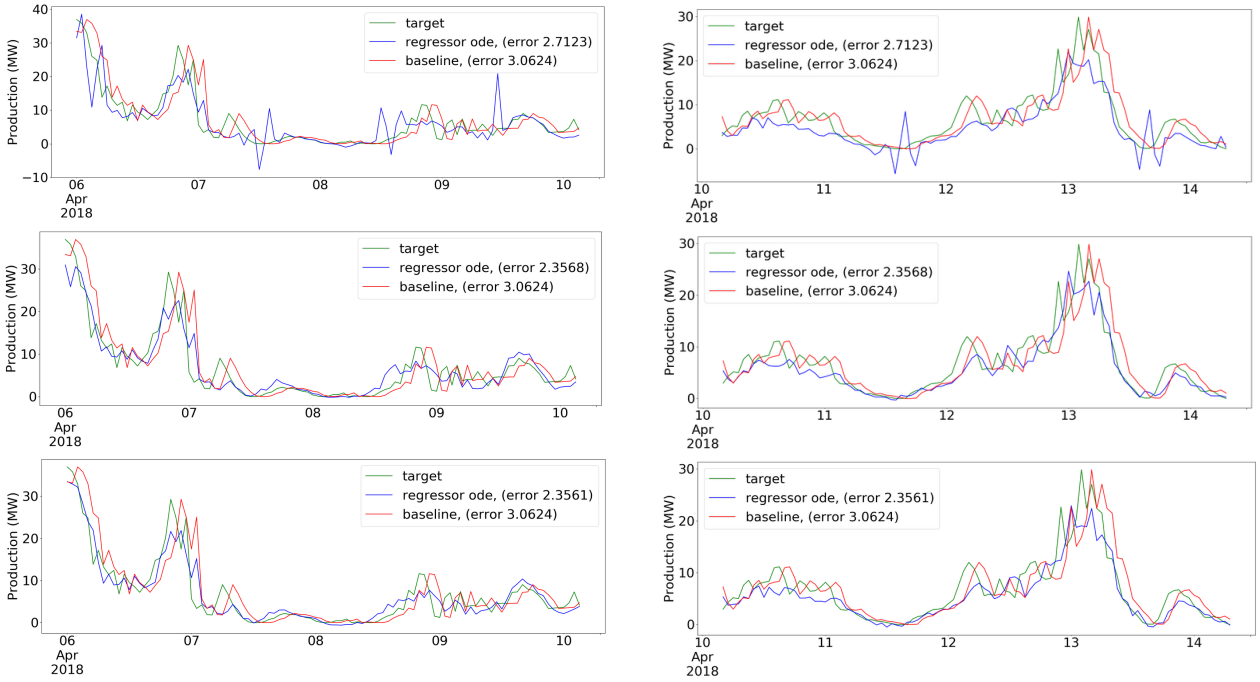


Figure 4.3: Sampled ODE Regressor predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

Latent ODE Regressor v1

The Latent ODE Regressor has by far the worst performance of all model, not coming close to the baseline in any variation. In figure 4.4 we see a similar pattern as for the Latent ODE; the model is only able to smoothly approximate the target, following its general trend. For this model run a general increase in accuracy with data size can be seen, but this is not consistent with average values in table 4.1. This said, in figure 4.4 there is a clear increase in accuracy for the peaks as more data is added. However, this is not consistent for other parts of the testing data, suggesting the model has specialized on certain parts of the time series.

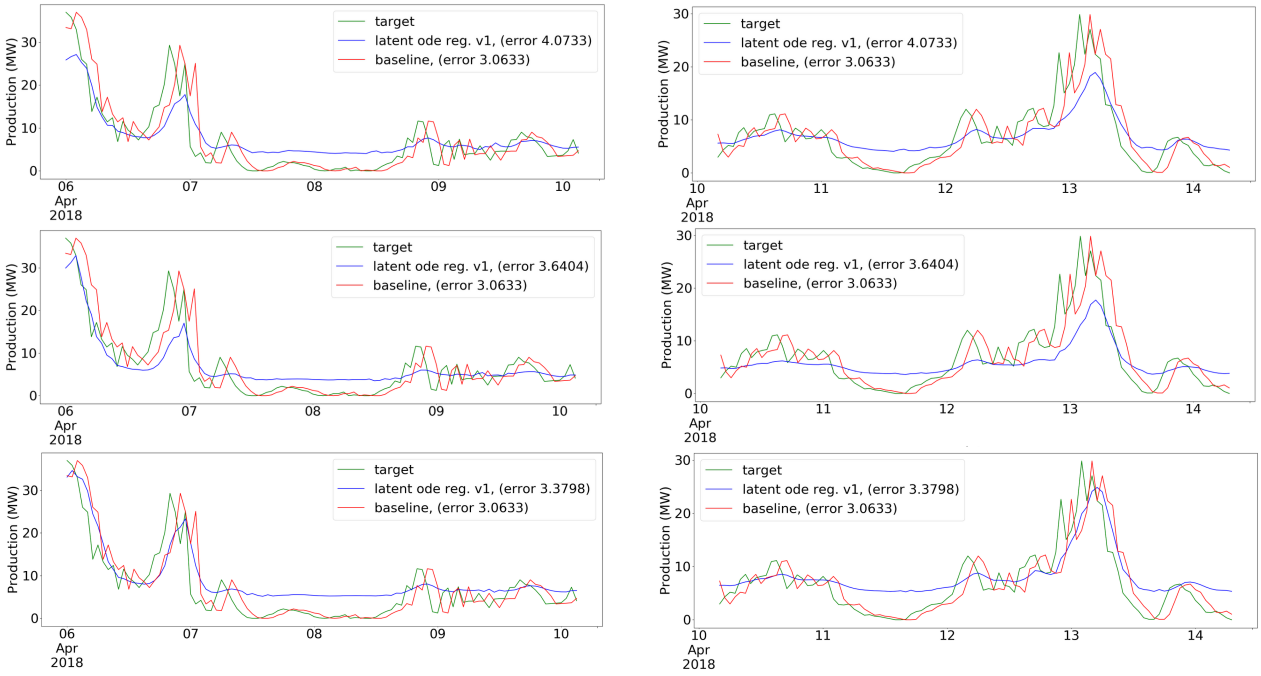


Figure 4.4: Sampled Latent ODE Regressor v1 predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

LSTM

LSTM is the one time-aware model which has a comparable performance to the time-agnostic models, beating the baseline for all variations. Similar to the ODE Regressor we see the most significant improvement in accuracy from 1 month to 6 months, but going to 12 months the improvement stagnates. At specific points, the predictions are able to improve upon the provided baseline, while at other points the model predicts the baseline value.

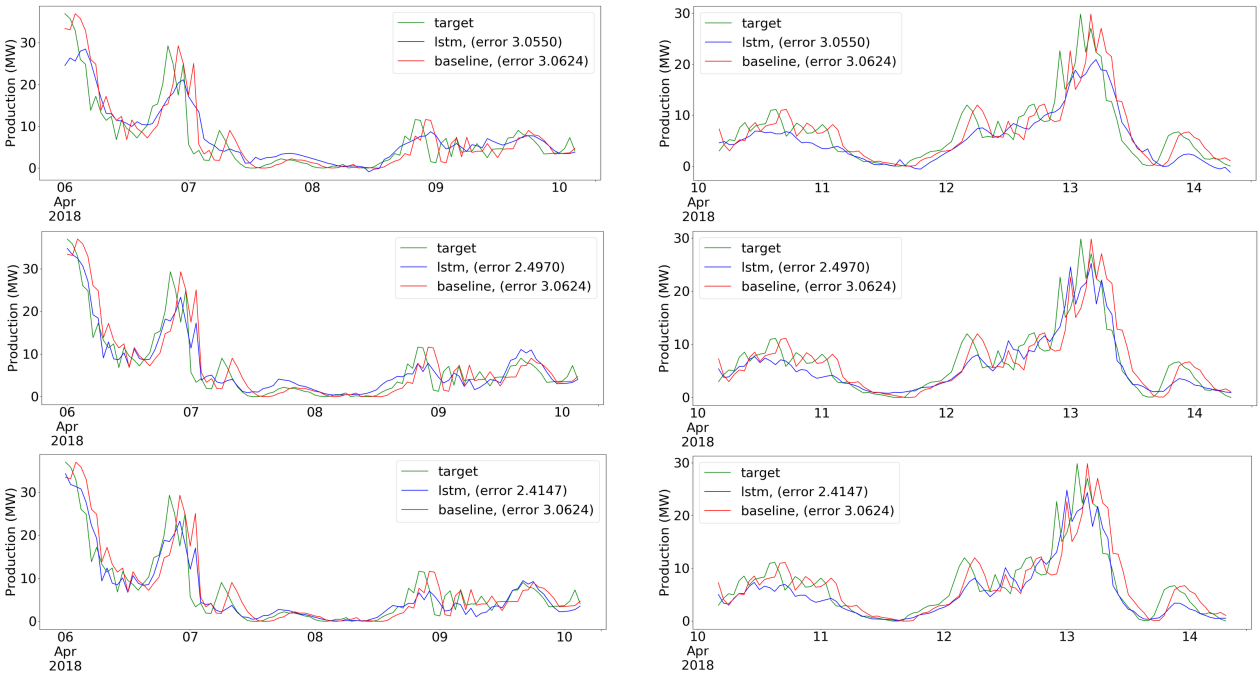


Figure 4.5: Sampled LSTM predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

DNN

The trajectories of the DNN models show an interesting evolution with more data. From figure 4.6 spikes can clearly be seen in the predictions with poor performance for 1 month data. This improves significantly with 6 and 12 months. The same pattern as for other models is observed, where the improvement is biggest when going from 1 to 6 months.

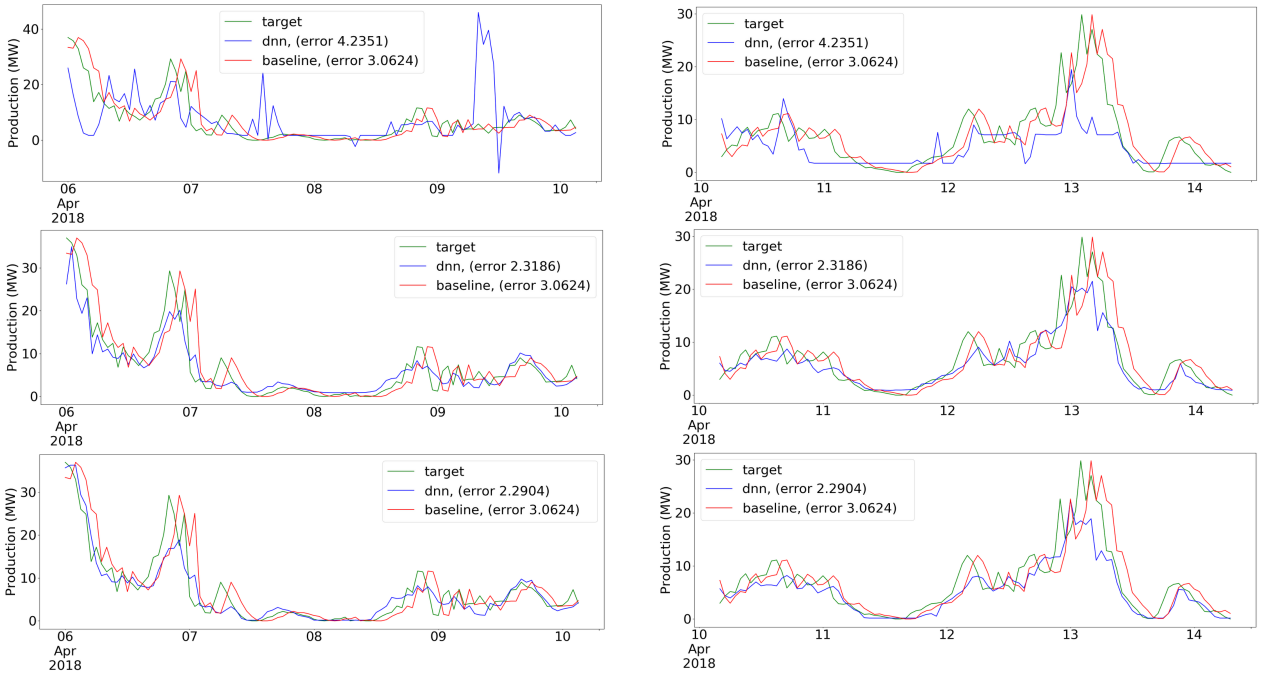


Figure 4.6: Sampled DNN predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

Gradient Boosting

The trajectories of gradient boosting models are the most consistent in terms of accuracy, beating the baseline every time. In figure 4.7 a steady improvement of the predictions as more data is provided can be seen. Like for the other models the improvement is greatest going from 1 to 6 months.

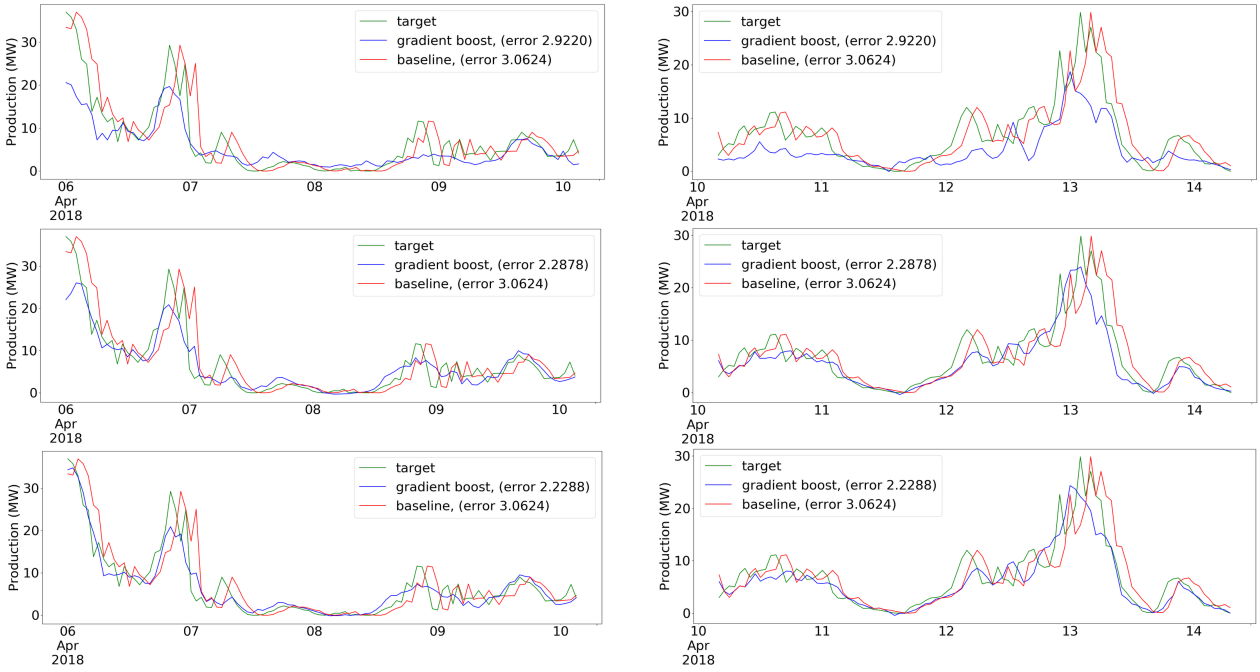


Figure 4.7: Sampled gradient boosting predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

Comperance of all Models Over Months

We now look at predictions from each model at different sizes of training data up against each other. Observe that some spikes in the 1 month predictions occur at the same time steps across different models, and that some of these spikes disappear with increased data. However, around April 17 the spike prevails for several of the time-agnostic models. Also note that the baseline error values differ for the time-aware NODE models, but stay equal for each specific model on each variation.

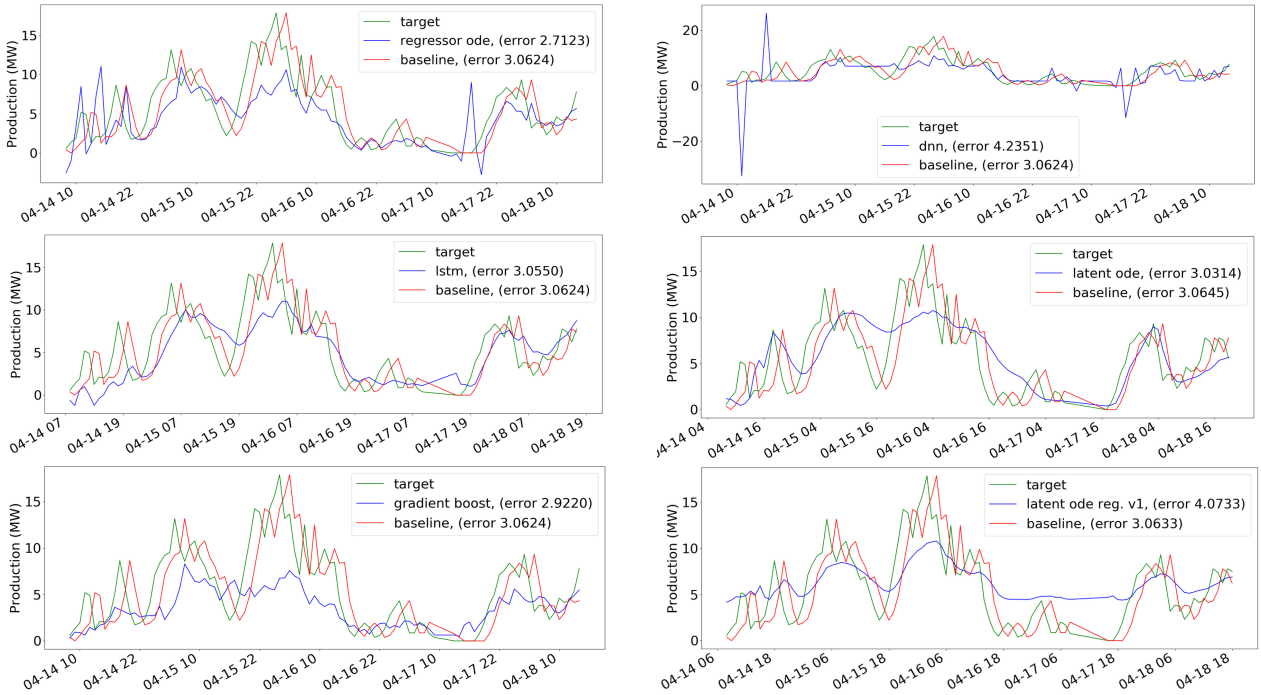


Figure 4.8: 1 month prediction comperance of all models.

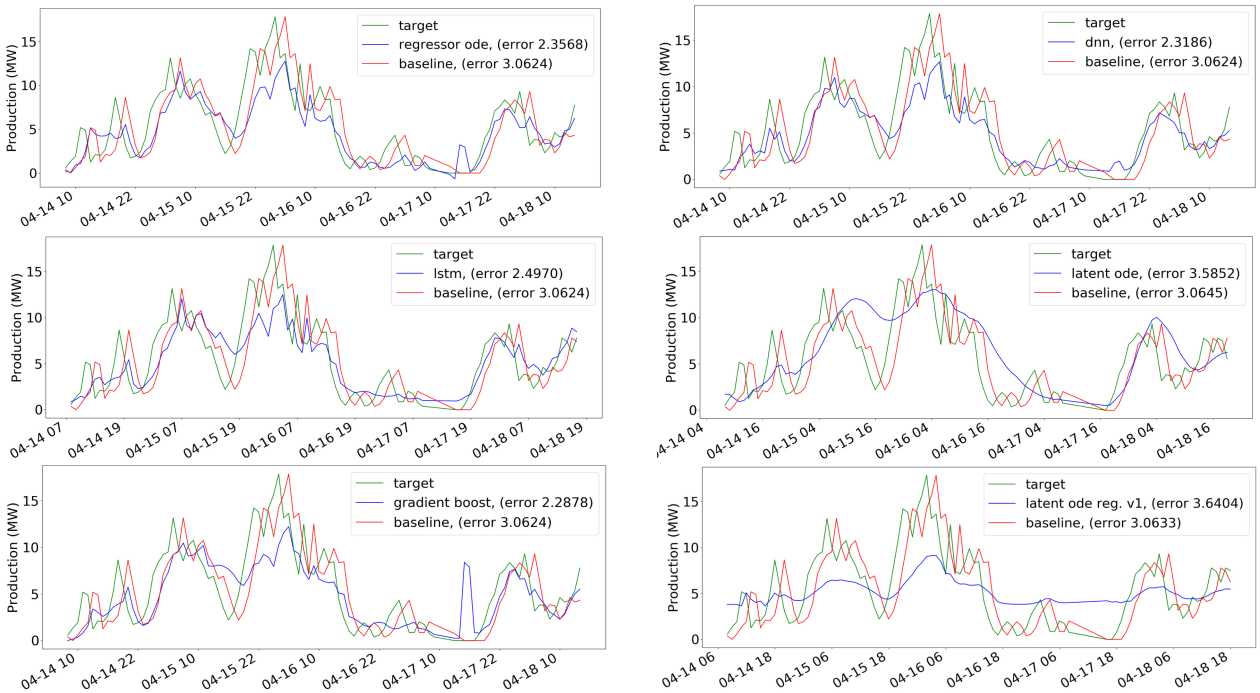


Figure 4.9: 6 months prediction comperance of all models.

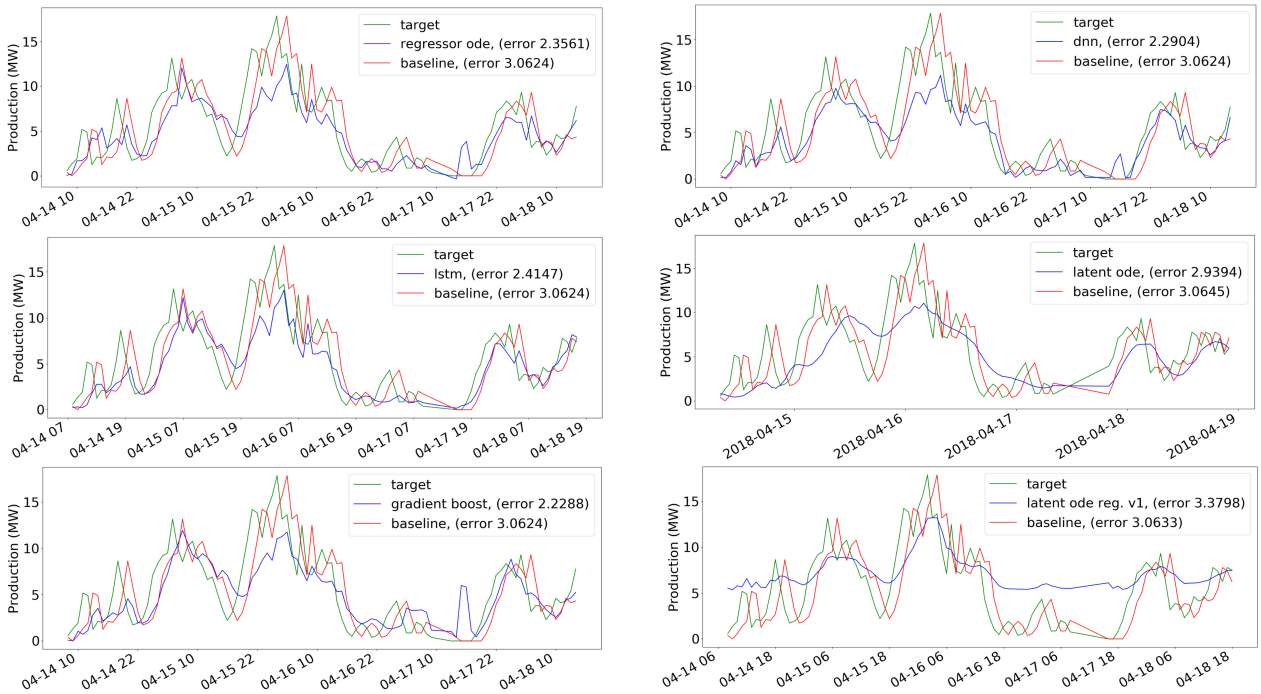


Figure 4.10: 12 months prediction comperance of all models.

4.2.3 Loss of NODE Models

During training an instability of the NODE models was observed. This instability is reflected in the loss of the models during training as seen in figure 4.11. The instability decreases with increased amount of training data. It is highest for the Latent ODE and Latent ODE Regressor, and is almost negligible for the ODE Regressor.

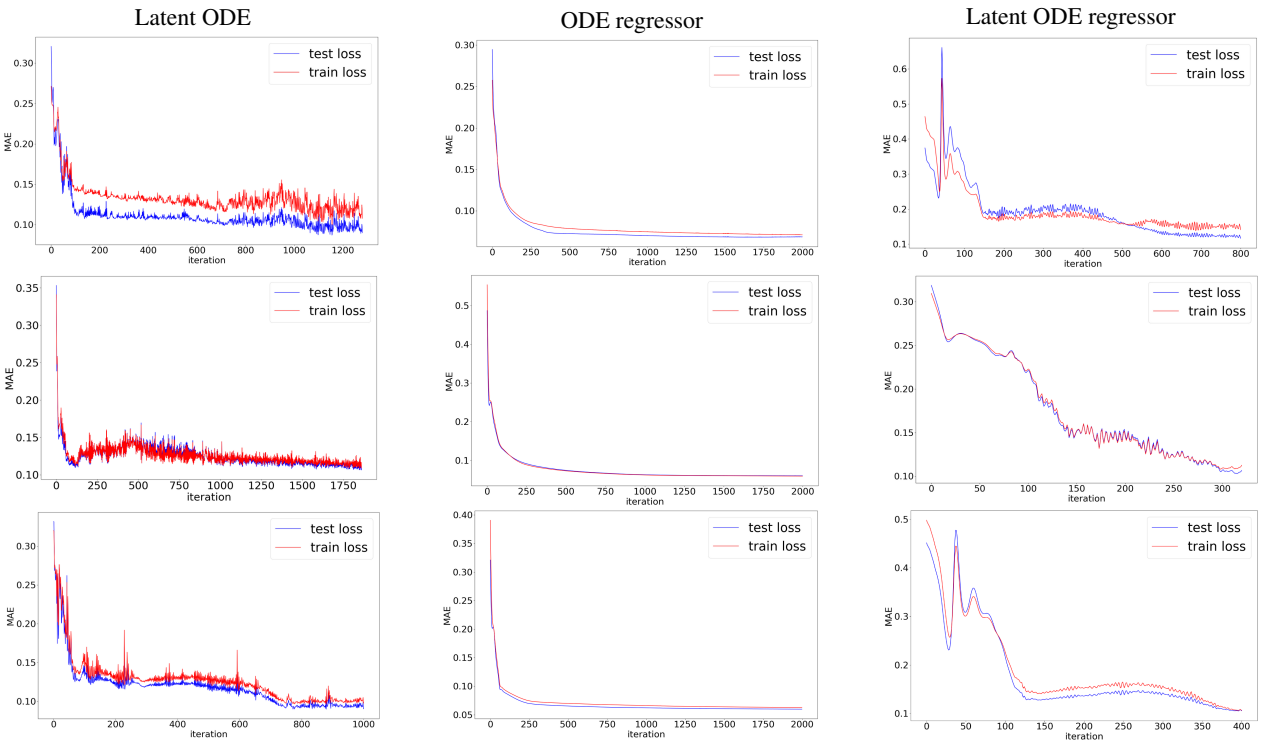


Figure 4.11: MAE on the target (production) of NODE models up to the iteration with best accuracy. Top 1 month. Middle 6 months. Bottom 12 months. The visualized loss is calculated on normalized test and training sets. Note the best models were found at various iterations during the training.

4.3 Experiment 2: Energy Consumption Energibygget

In the Energy Consumption Forecasting for Energibygget problem the goal is to predict the total hourly consumption of energy one hour ahead in Energibygget located at Lerkendal, Trondheim. Recall from section 2.1.1 forecasts at the peaks are especially critical for this problem. In addition to this, the experiment investigates how the number of model parameters affect the prediction accuracy. Because of the vast difference in model parameters, for each model a ”large”, ”medium”, and ”small” model was defined. All of the models were run with the same training data from eight months, and tested with data from following two months, with a gap between train and test set. Further, each NODE model was run three times on the respective variation, and plots from the model with the highest accuracy on the test set were saved. Error from the benchmark models is also reported. These models showed significantly less variation during the experimental setup phase and were only run once each. The baseline for this experiment is a one hour persistence model.

4.3.1 Overview of Results

Table 4.2 shows the results from this experiment. Observe the DNN model has the lowest error for all model sizes, and that all time-agnostic models perform better than the time-aware models. The best NODE model is the ODE Regressor, followed closely by the Latent ODE Regressor v2. Observe that the models with most parameters have the worst performance out of all models, and the LSTM and Latent ODE models do not even beat the baseline persistence model. Note the underflow in the ”large” and ”medium” Latent ODE models. The underflow occurs in the torchdiffeq NODE library [Chen, 2019] when the ODE Block takes a smaller step than is numerically possible.

Comperance		DNN	Catboost	LSTM	Baseline
	# input variables	23	23	23	N/A
	# training samples	3903	3903	3903	N/A
small	# params	617	2000 trees depth 3	2441	
	test mae	**9057**	**10008**	**13321**	11317
medium	# params	793	2000 trees depth 6	7953	
	test mae	**8560**	**10235**	**14209**	11317
large	# params	1077	2000 trees depth 10	16537	
	test mae	**9114**	**10434**	**13956**	11317

Comperance		Latent ODE	ODE Regressor	Latent ODE Regressor v2
	# input variables	23	23	23
	# training samples	3903	3903	3903
small	# params	2112	576	525
	test mae run1	20038	9316	10229
	test mae run2	20095	10145	10160
	test mae run3	22483	9827	10305
	avg mae	**20872**	**9762**	**10231**
medium	# params	7648	940	2427
	test mae run1	18124	9181	9727
	test mae run2	underflow	9889	9527
	test mae run3	21028	9204	9613
	avg mae	**19576**	**9425**	**9622**
large	# params	15726	1128	8711
	test mae run1	underflow	9388	9596
	test mae run2	underflow	9418	9589
	test mae run3	underflow	9272	9542
	avg mae	**N/A**	**9359**	**9576**

Table 4.2: MAE (W) of benchmark models (left) and NODE models (right).

Variations on Number of Parameters

Looking at error vs. the number of parameters in figure 4.12 there is some correspondence between the two factors for the NODE models with performance better than the baseline. However, the ”large” DNN and gradient boosting actually have higher error than the ”small” and ”medium” models. Note also that the gradient boosting error is not plotted as the exact number of parameters is not known. In table 4.2 we see that the performance of the gradient boosting models is stable, but decreasing, across all models. Comparing DNN, ODE Regressor, and Latent ODE Regressor models it can be seen that the DNN achieves a lower loss with a comparable number of trainable parameters. Note also that the numerical underflow of the ”large” Latent ODE is plotted for visual effect, even though the value is actually non-existing.

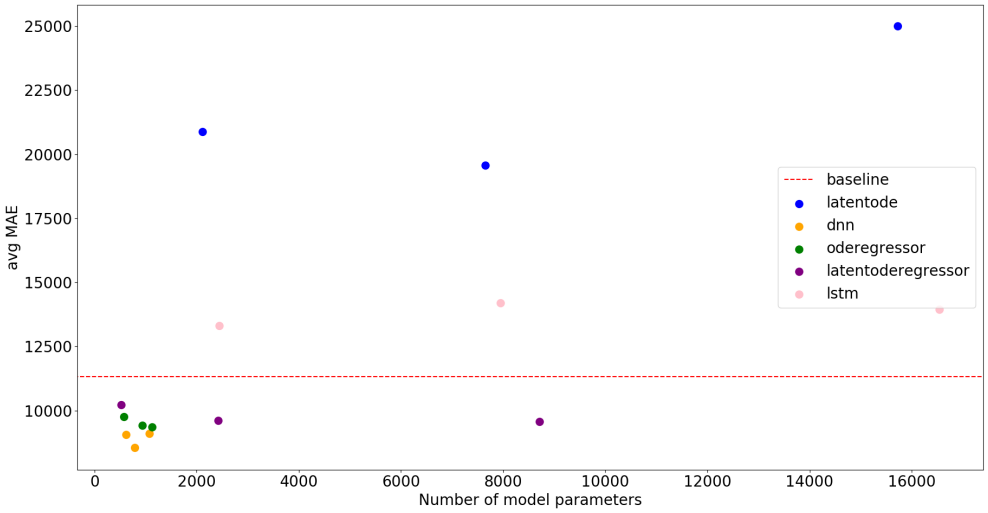


Figure 4.12: Average MAE (W) of all models over number of parameter based on values in table 4.2.

4.3.2 Separate Model Size Variations

Next, the attention turns to the results of each separate model, studying the predicted trajectories in detail. For some dates, there is a discrepancy between the baseline and the target values (can be seen around 02-07 15). This is due to missing data for the specific dates, causing a ”jump” in the plot.

Latent ODE

As we see in figure 4.13, the Latent ODE models fail to predict the target feature trajectories. The model is only able to follow the general trend, but misses out on the finer details. The ”large” model plot is missing as the experiment failed to run to the end.

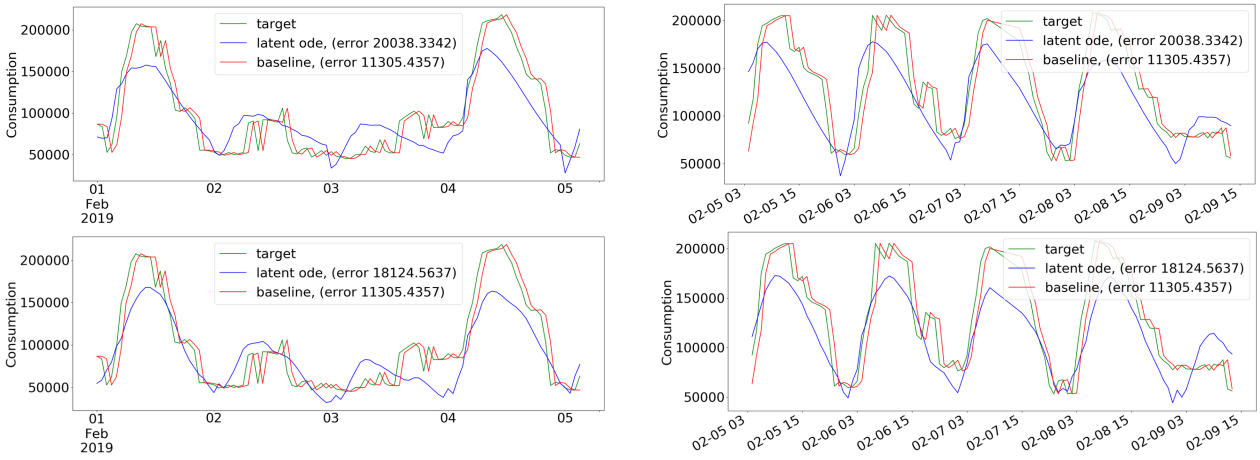


Figure 4.13: Sampled Latent ODE predictions on the test set. Top ”small”. Bottom ”medium”. No plot exist for ”large” due to numerical underflow during the experiment. Consumption and MAE in W.

ODE Regressor

The ODE Regressor model has the lowest error of the NODE models. Observe that there is no clear connection between the loss and model size. This is inconsistent with the average values in table 4.2 which indicate the error decreases with model size. The trajectories in figure 4.14 show that all sizes are able to extract more information than mimicking the baseline. Note that the forecasts at the consumption peaks seems to improve with more parameters.

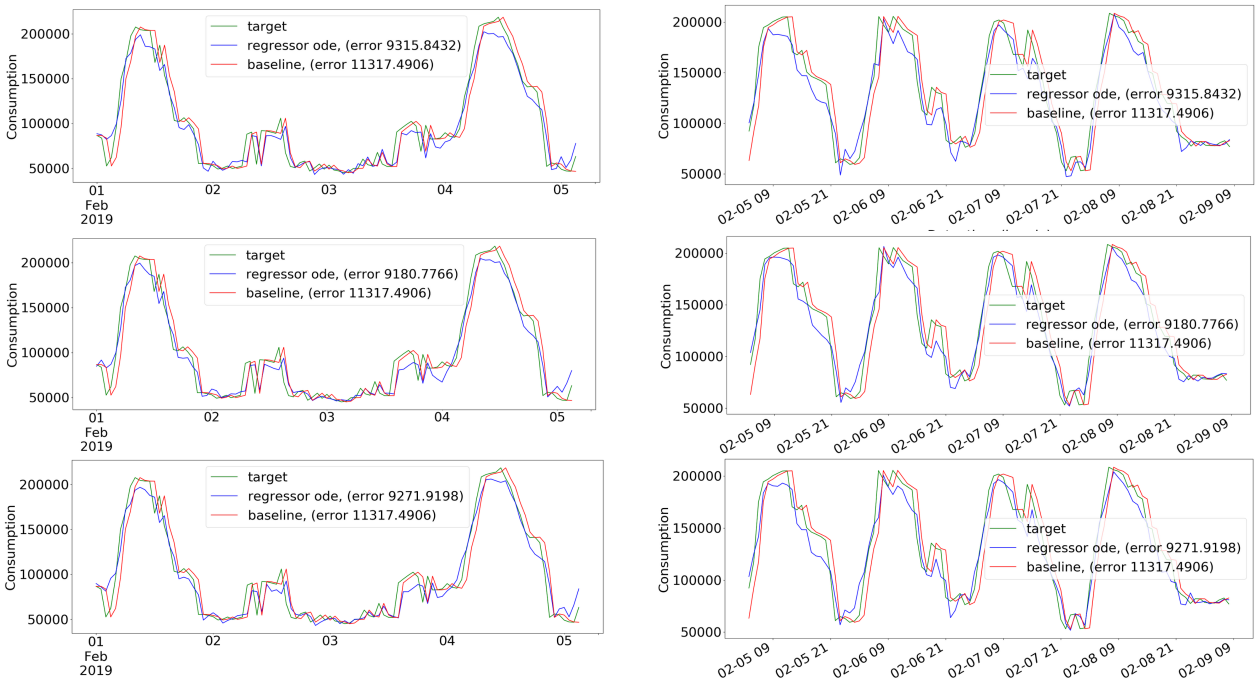


Figure 4.14: Sampled ODE Regressor predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”. Consumption and MAE in W.

Latent ODE Regressor v2

The predicted trajectories of the Latent ODE Regressor models in figure 4.15 all have a better performance than the baseline. Observe the ”small” model has a significantly higher error than the other two. This is consistent with the average value in table 4.2. The ”large” model has a higher error than the ”medium” model, which is inconsistent with values in table 4.2. Peak forecasts are similar for all model variations.

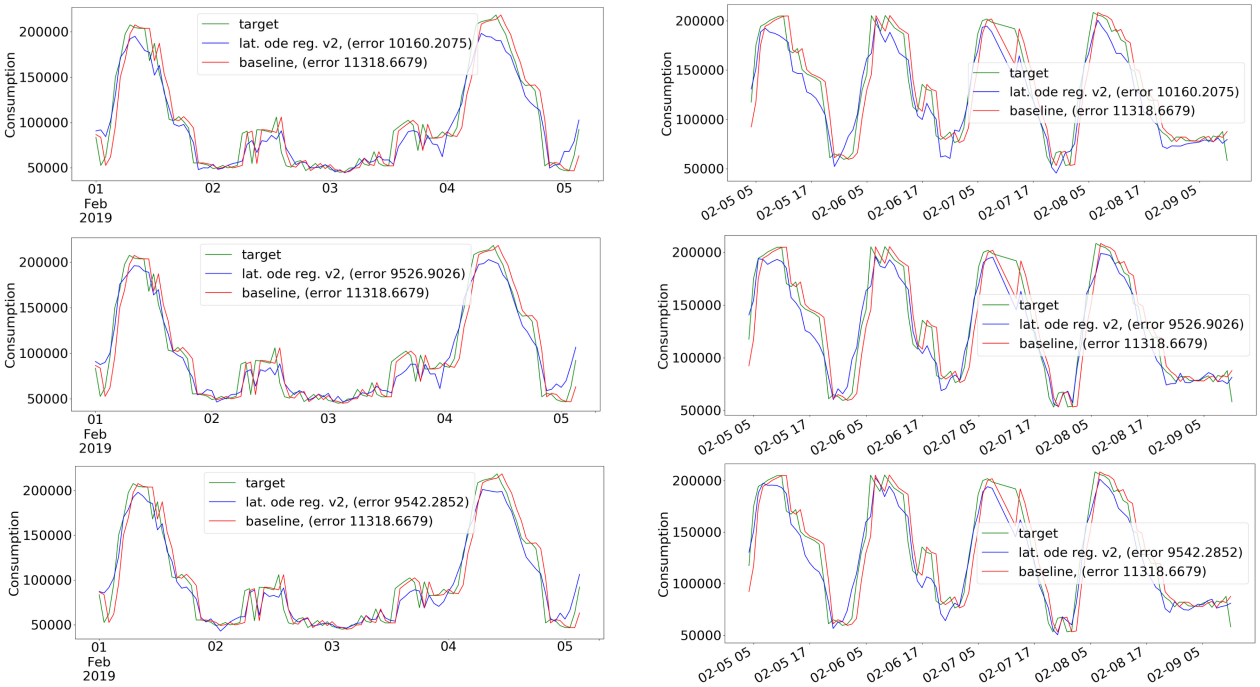


Figure 4.15: Sampled Latent ODE Regressor v2 predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”. Consumption and MAE in W.

LSTM

All LSTM models perform worse than the baseline. Figure 4.16 shows the predicted trajectories follow the target relatively well, but under-shoots the peaks. Seemingly, there is nothing indicating model size and accuracy are correlated.

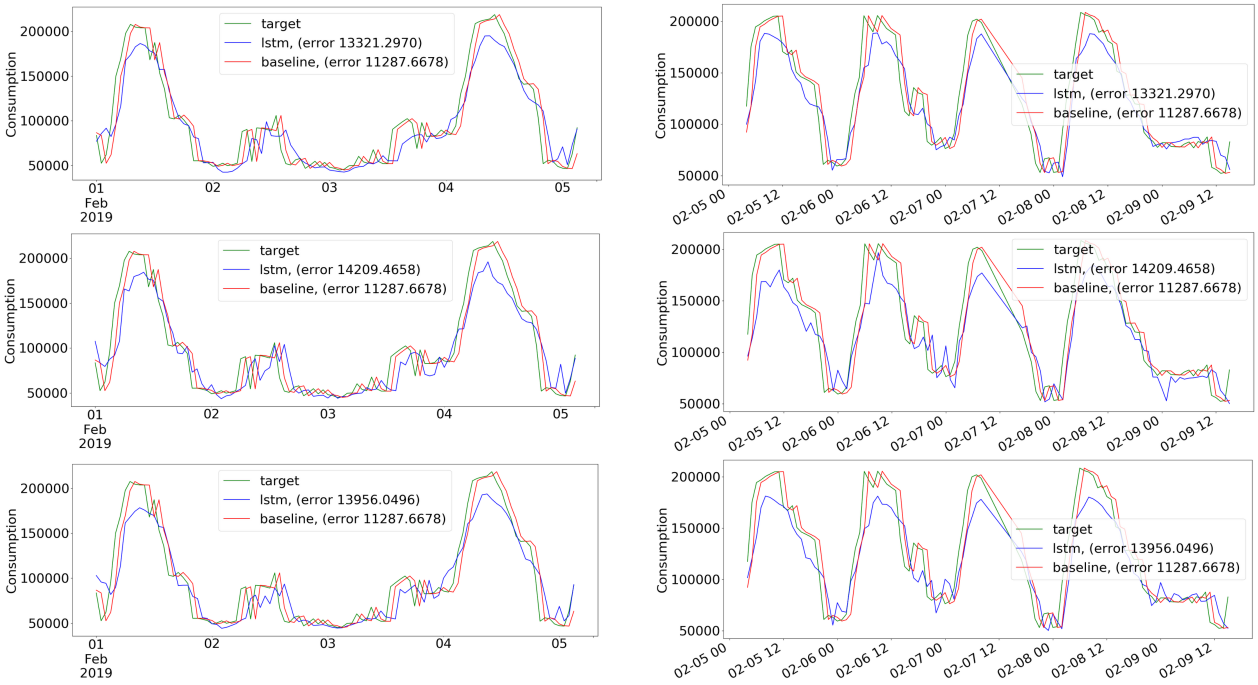


Figure 4.16: Sampled LSTM predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”. Consumption and MAE in W.

DNN

The DNN is the best performing model for this problem. There is no clear connection between model size and total error. The predicted trajectories in figure 4.17 follow the target and are able to hit most peaks accurately. The ”small” model forecasts the peaks worse than the other two.

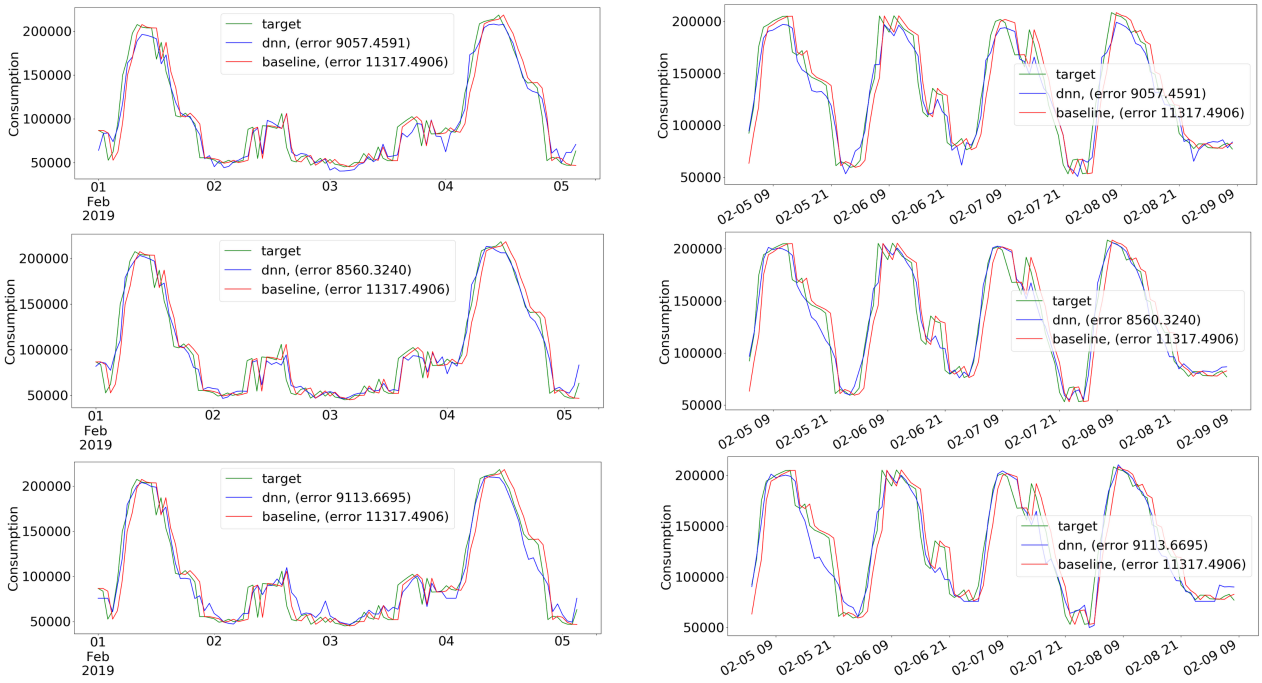


Figure 4.17: Sampled DNN predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”. Consumption and MAE in W.

Gradient Boosting

Trajectories from the gradient boosting models were able to beat the baseline in all cases. But from the trajectories in figure 4.18 it can be seen the models have a hard time predicting the peaks of consumption.

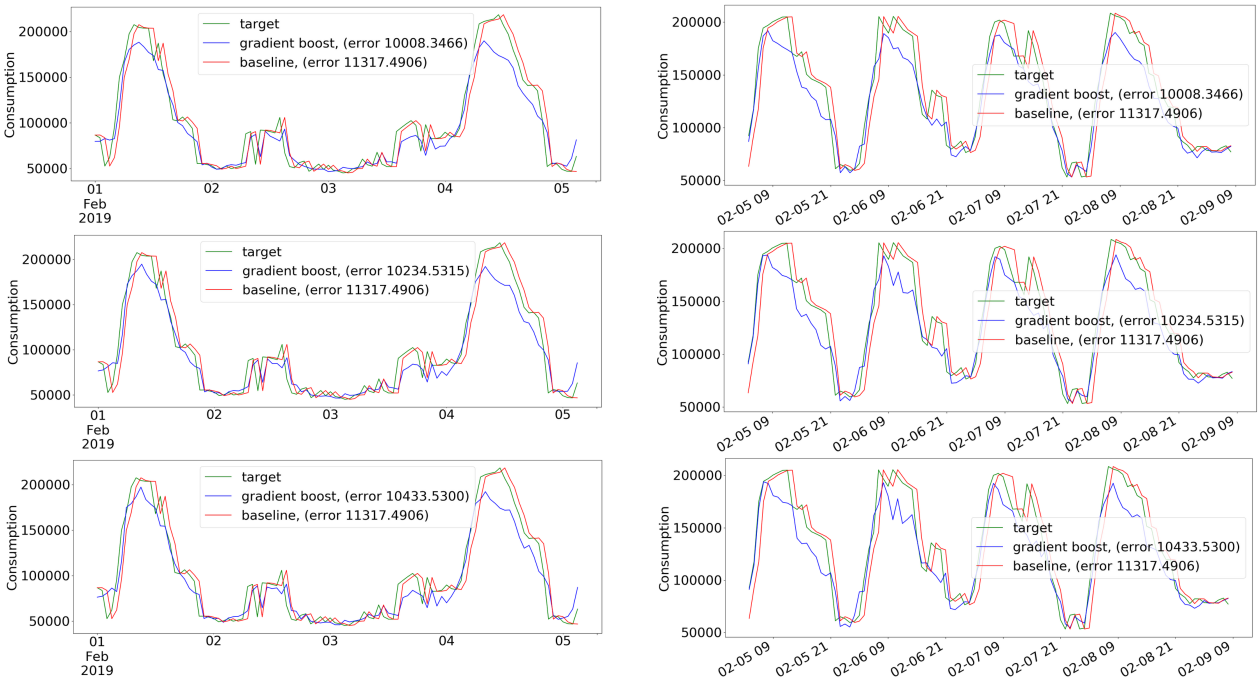


Figure 4.18: Sampled gradient boosting predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”. Consumption and MAE in W.

Comperance of all Models Over Size

We now look at predictions from each model at different sizes of model parameters up against each other. The Latent ODE stands out negatively, and is the only model not able to predict fine-grained details. Also note that the baseline error values differ for the time-aware NODE models, but stay equal for each specific model on each variation.

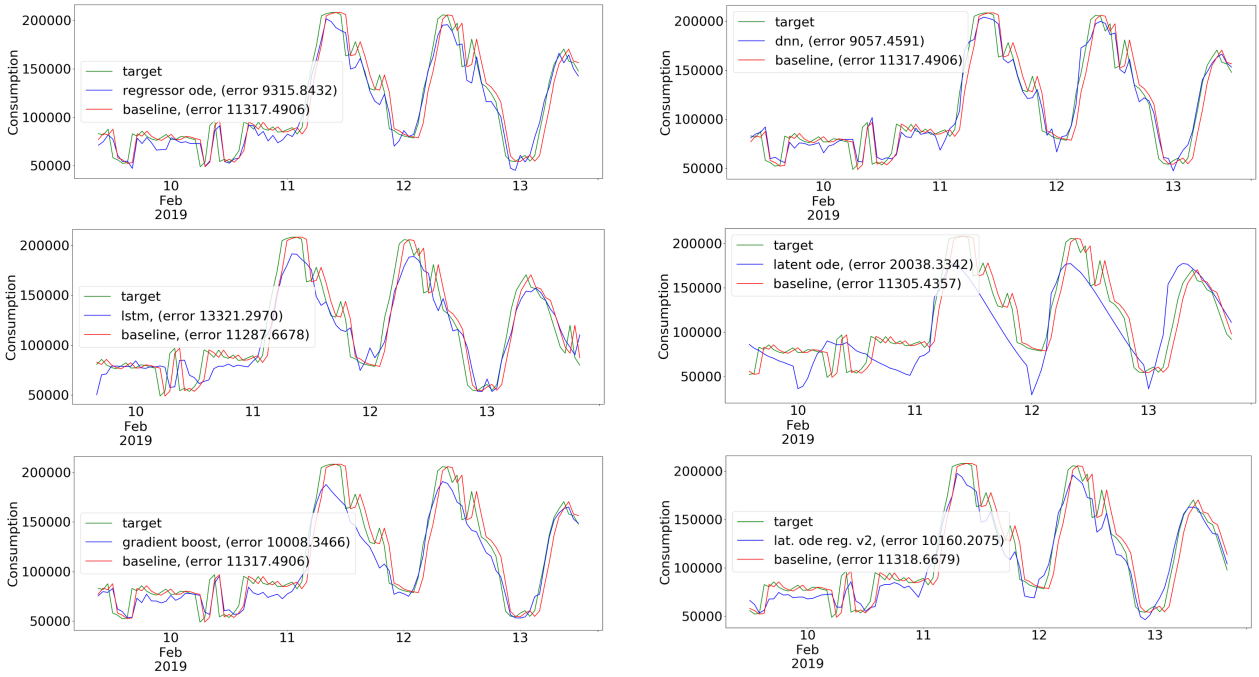


Figure 4.19: ”Small” prediction comperance of all models. Consumption and MAE reported in W.

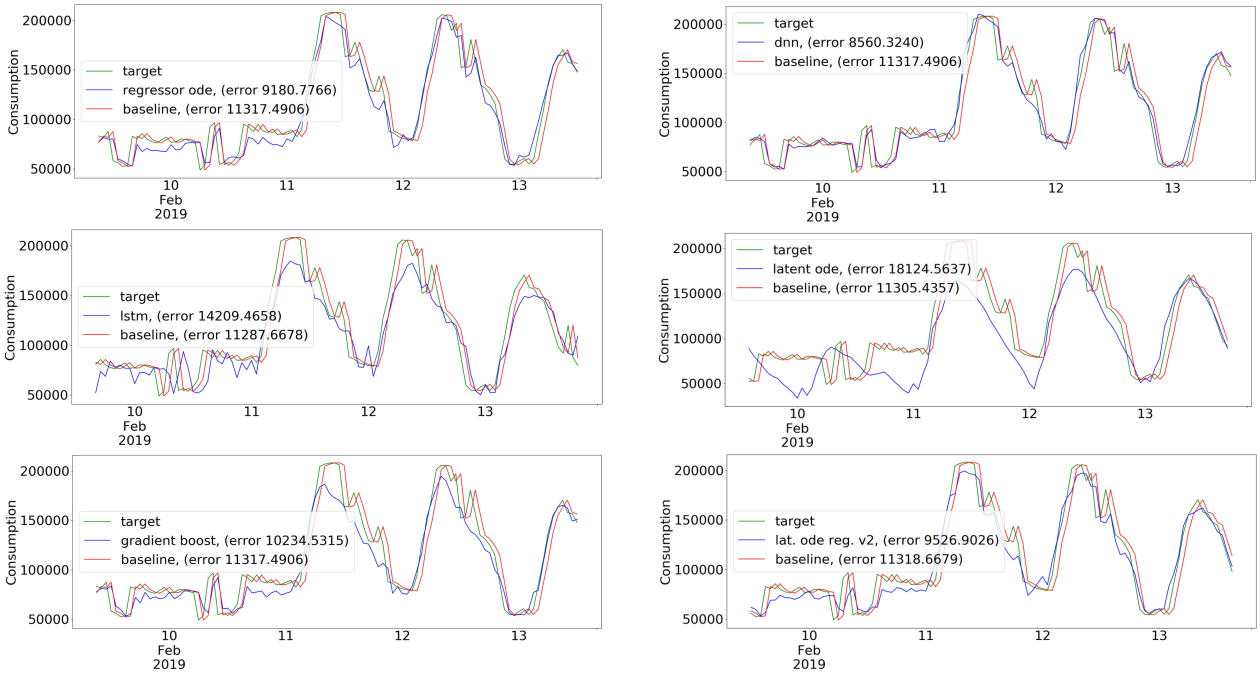


Figure 4.20: "Medium" prediction comperance of all models. Consumption and MAE reported in W.

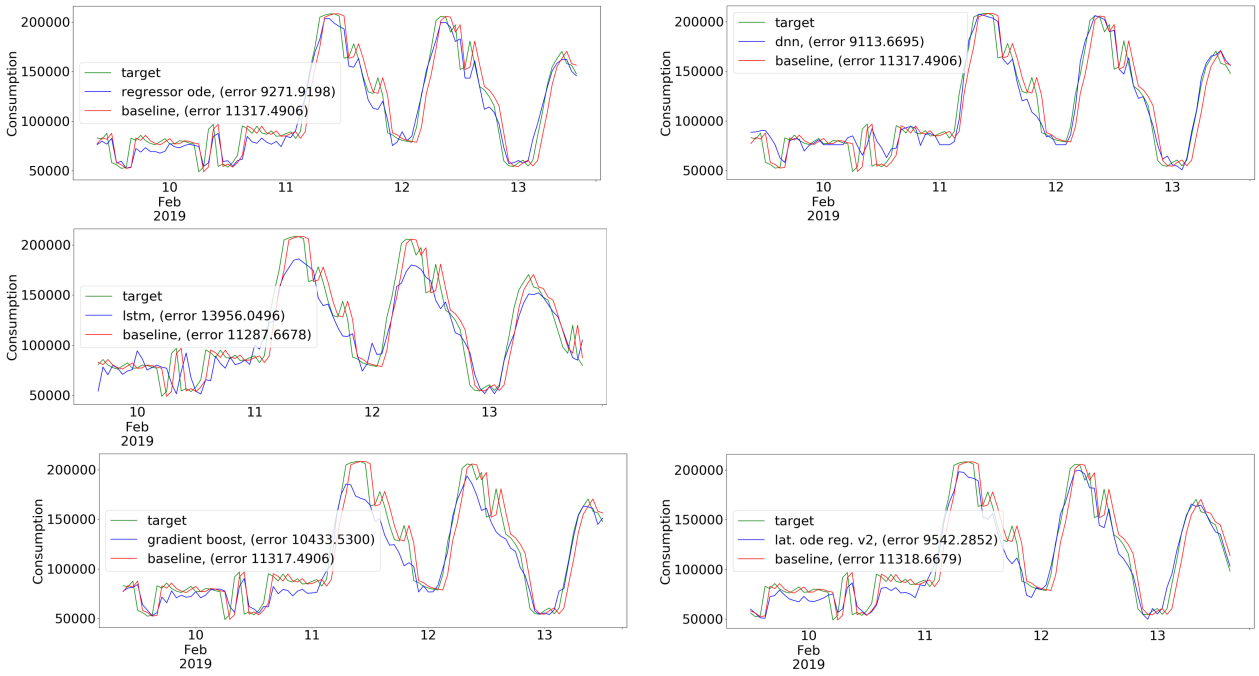


Figure 4.21: "Large" prediction comperance of all models. Plot for the Latent ODE was not obtained. Consumption and MAE reported in W.

4.3.3 Loss of NODE Models

Similar to experiment 1 instability in the loss function was observed during training of the NODE models. The Latent ODE model has the biggest instabilities, followed by the Latent ODE Regressor. The ODE Regressor has significantly less instability. Note that loss for the ”large” Latent ODE model is missing because of the numerical underflow.

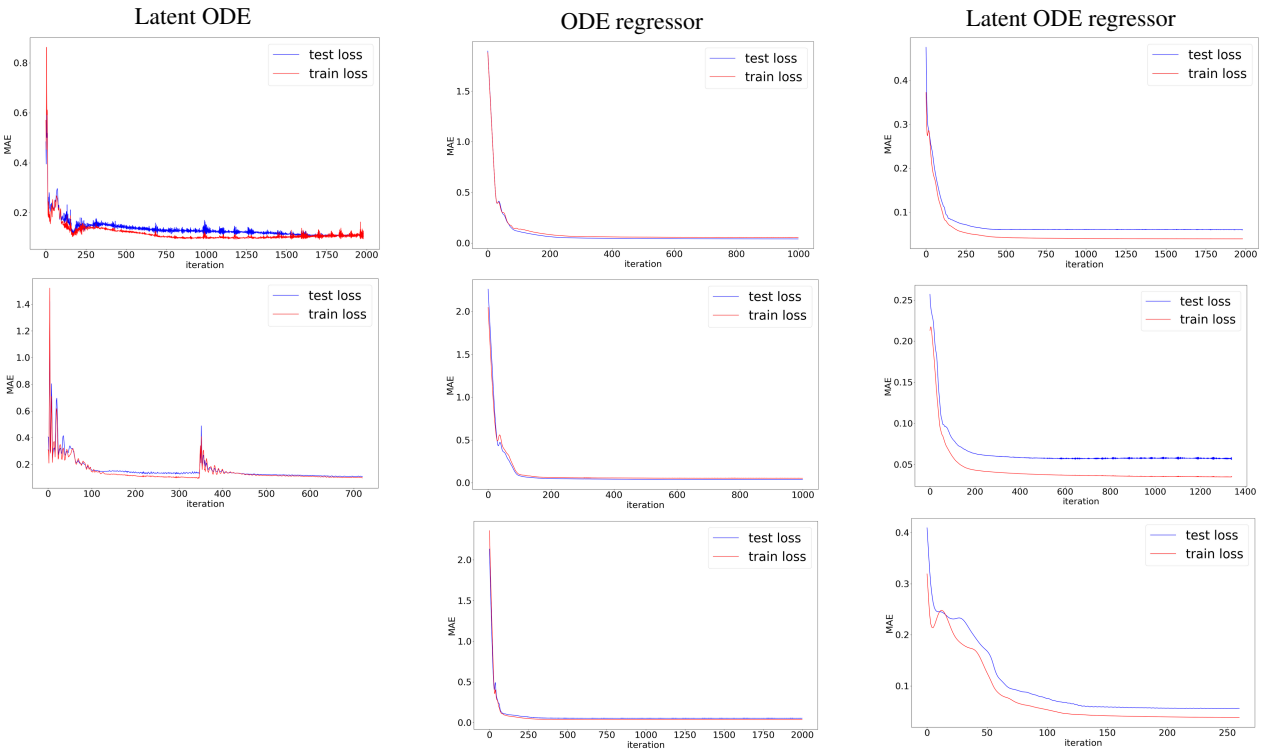


Figure 4.22: MAE on the target (consumption) of NODE models up to the iteration with best accuracy. Top ”small”. Middle ”medium”. Bottom ”large”. The visualized loss is calculated on normalized test and training sets. Note the best models were found at various iterations during the training.

4.4 Experiment 3: Energy Consumption NO3 Region

In the Energy Consumption Forecasting for NO3 Region problem the goal is to predict the total hourly consumption of energy for the next 24 hours in the NO3 region located in Norway (figure 2.2). In addition to this, the experiment investigates how the latent state and loss function impact the predictions of the Latent ODE Regressor v3. Four Latent ODE Regressors are compared along two variations. The first variation is a small and normal latent state size. The second variation is on only reconstruction loss versus reconstruction and KL-divergence loss. The only benchmark model for this problem is LSTM. All models use the same training and test set. Further, each model is run three times, and plots from the model with the highest accuracy on the test set is saved. Loss for 1st hour ahead, 24th hour ahead, and the next 24 hours ahead is reported. For the 1st hour ahead prediction the baseline is a one hour persistence model. For the 24th hour ahead prediction the baseline is a 24 hour persistence model.

4.4.1 1st and 24th Hour Forecasts

The most interesting discovery in table 4.3 is that LSTM outperforms the other models on a one hour ahead prediction, reaching an error which is half of the best NODE model, but on the 24th hour ahead it is opposite: The best NODE model reaches an error which is half of LSTM. Also, as can be seen from each separate run, the volatility for the LSTM at the 24th hour ahead is significantly larger than for one hour ahead. Another observation is that the NODE models trained with only reconstruction loss have a lower average error, consistent across both the small and large latent state. Figure 4.23 visualizes losses for each model at the 1st and 24th hour. Interestingly, observe that the first run for the Latent ODE Regressor normal latent state full loss has a lower loss on the 24th hour forecast than for the 1st hour. This is the only run where this is the case. Finally, there is a consistency in the best performing model run for each model. The run with the lowest error on the 1st hour also has the lowest error on the 24th hour.

Comperance		LSTM	Latent ODE Regressor small latent state reconstruction only	Latent ODE Regressor small latent state reconstruction + KL-div	Latent ODE Regressor normal latent state reconstruction only	Latent ODE Regressor normal latent state recontruction + KL-div	Baseline
	# input variables	68	68	68	68	68	N/A
	# training samples	8755	8755	8755	8755	8755	N/A
	# params	17659	6654	6654	10245	10245	
1h	test mae run1	**27**	**68**	86	56	115	58
	test mae run2	**27**	87	93	**51**	**83**	
	test mae run3	28	83	**83**	53	98	
	avg mae	**27**	**79**	**87**	**53**	**99**	
24h	test mae run1	182	**78**	**99**	**77**	100	128
	test mae run2	**133**	97	103	**77**	**91**	
	test mae run3	160	88	**99**	78	102	
	avg mae	**158**	**88**	**100**	**77**	**98**	

Table 4.3: MAE (MW) of all models.

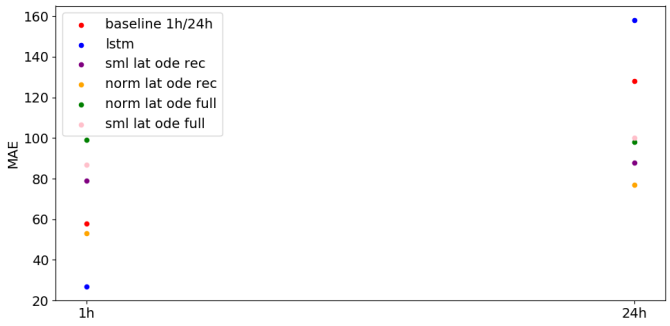


Figure 4.23: Average MAE (MW) of all models at 1st and 24th hour respectively. Interestingly, LSTM goes from lowest to highest error when moving from 1st to 24th hour forecast.

1st Hour Trajectories

Figure 4.24 shows 1st hour predicted trajectories of the best run for all models. LSTM has the lowest error which is also reflected in the forecast trajectories. It is clearly able to follow the target closer than the other models. Further, the NODE models with full loss have higher errors, and only the normal latent state with reconstruction loss is able to reach errors below the baseline.

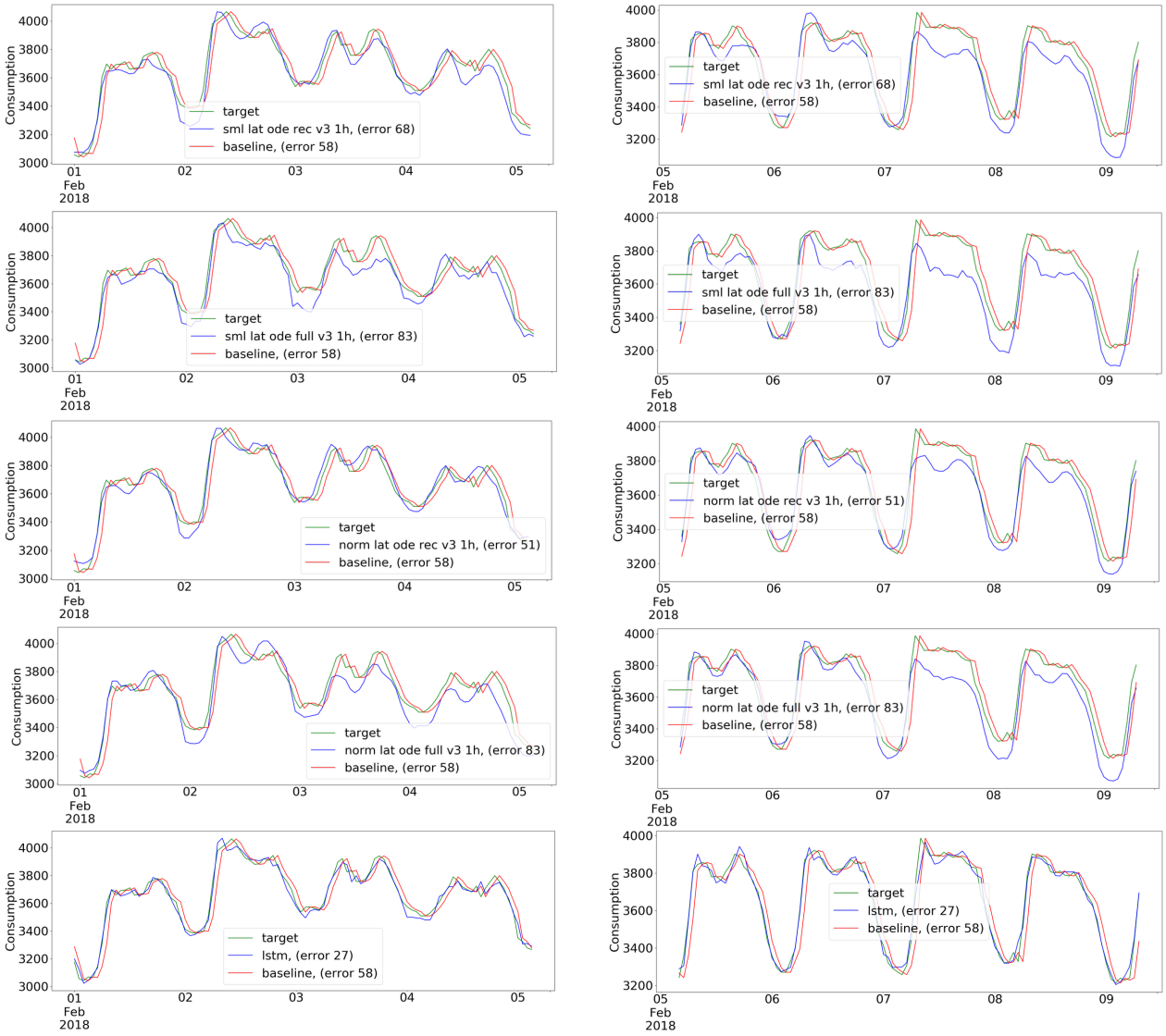


Figure 4.24: Predicted trajectories 1 hour ahead for all models. Consumption and MAE shown in MW.

24th Hour Trajectories

Figure 4.25 shows 24th hour predicted trajectories of the best run for all models. The NODE models have the lowest error which is also reflected in the forecast trajectories. Further, there is a clear relation between the error and the loss function used. NODE models with only reconstruction loss have lower errors. The size of the latent state does not have the same impact, although this is not consistent with the average errors in table 4.3 which indicate a bigger latent state yields a lower error.

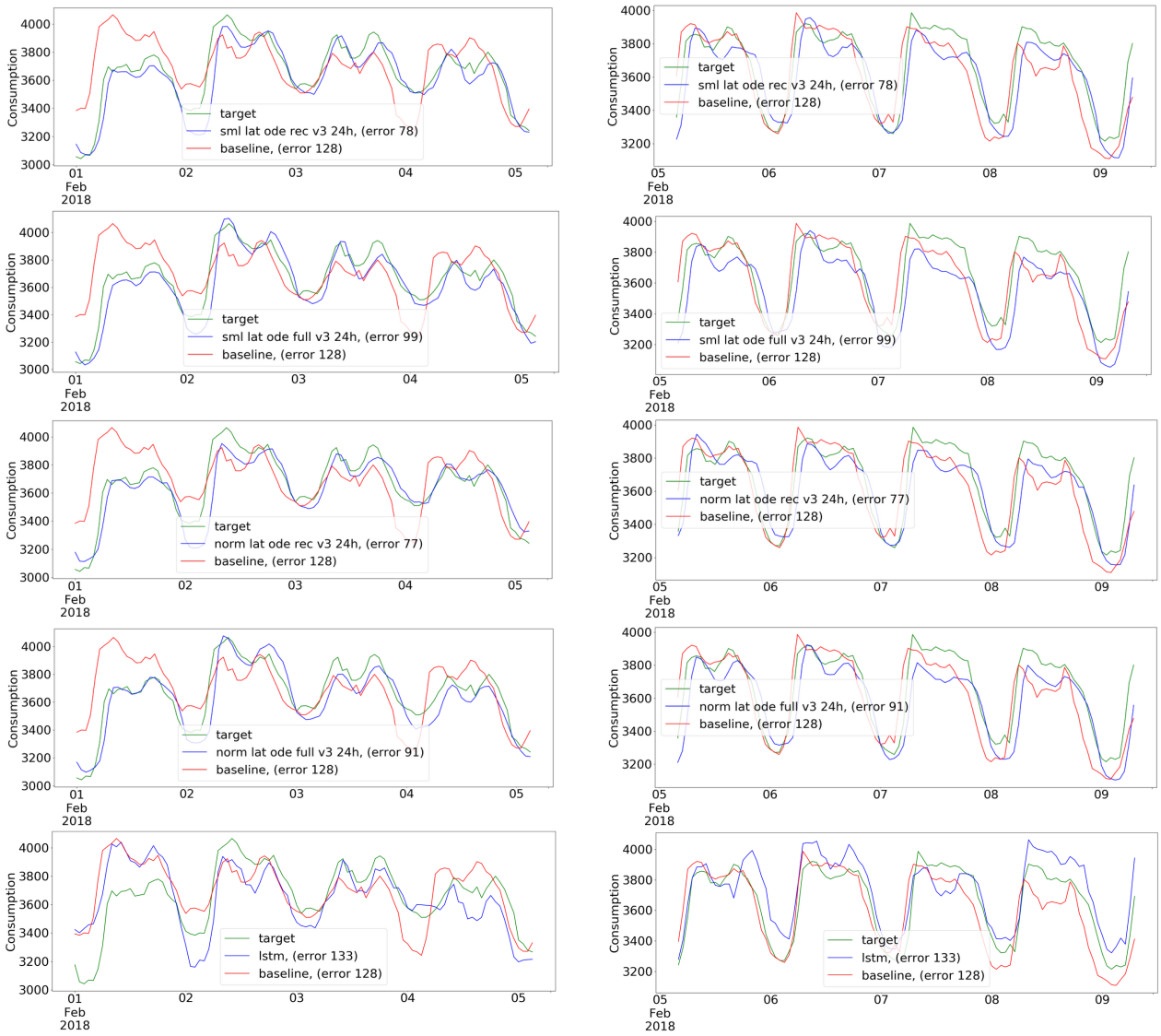


Figure 4.25: Predicted trajectories 24th hour ahead from all models. Consumption and MAE shown in MW.

4.4.2 Next 24 Hours Forecasts

The next 24 hours ahead problem is different from the previous in that predictions are reported for all hours, and not only the endpoints. However, it is still predictions from the same models, using the same training and test data as previously. One could say the loss is reported ”horizontally” in the time direction, instead of ”vertically” across all samples. Five different dates are chosen randomly from the test set, and the forecast for the next 24 hours is reported. Table 4.4 summarizes the results of this part of the experiment. The best performing model, in terms of average error and number of individual best, is the Latent ODE Regressor with a normal latent state trained with reconstruction loss only. This is consistent with findings for the 1st hour and 24th hour predictions in the previous section. In second place comes LSTM with two individual bests, but LSTM is not better on average than the three remaining NODE models. Note the volatility in errors between the models. A model that predicts well on one time step, may have a much worse prediction on another time step. This is especially seen by comparing LSTM and the Latent ODE Regressor with small latent state trained with reconstruction loss only. A visualization of loss over all five chosen time steps is shown in figure 4.26.

	LSTM	Latent ODE Regressor small latent state reconstruction only	Latent ODE Regressor small latent state reconstruction + KL-div	Latent ODE Regressor normal latent state reconstruction only	Latent ODE Regressor normal latent state recontruction + KL-div
# input variables	68	68	68	68	68
# training samples	8755	8755	8755	8755	8755
# params	17659	6654	6654	10245	10245
2017-12-01 11:00:00	90	136	190	**76**	203
2017-12-05 05:00:00	116	50	91	**42**	67
2017-12-09 09:00:00	**30**	51	63	41	81
2017-12-13 13:00:00	**40**	94	107	77	100
2017-12-17 17:00:00	197	98	167	**76**	161
# of individual best	2	0	0	**3**	0
avg mae	89	81	93	**65**	114

Table 4.4: MAE (MW) of all models on the next 24 hours at five random dates from the test set.

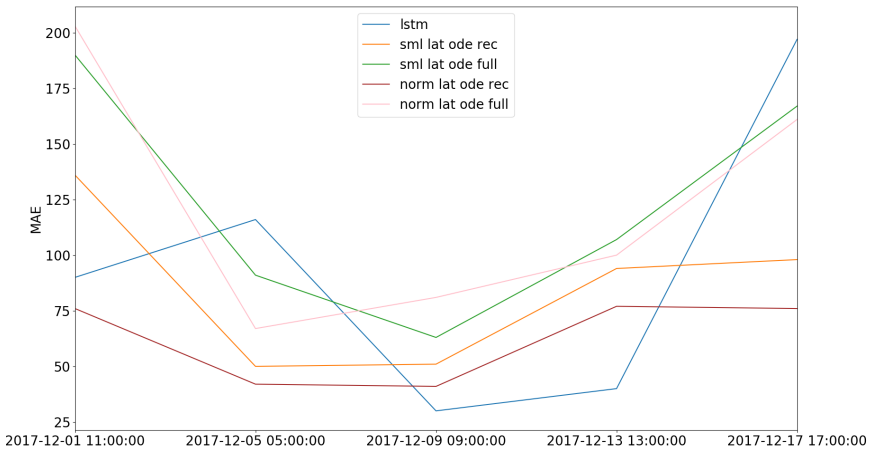


Figure 4.26: Average MAE (MW) of the next 24 hours at random discrete dates from the test set. Note the errors are not continuous from one date to the next as the line suggests. The line makes it easier to compare the overall performance between the discrete dates.

Next 24 Hours Trajectories

Figures 4.27 show two sampled trajectories from table 4.4. Note how the LSTM follows the target for the first hours, and later diverges from the target. This is somewhat opposite for the NODE models. This observation is consistent with the findings in table 4.3 which suggest NODE models are better at predicting the last hours.

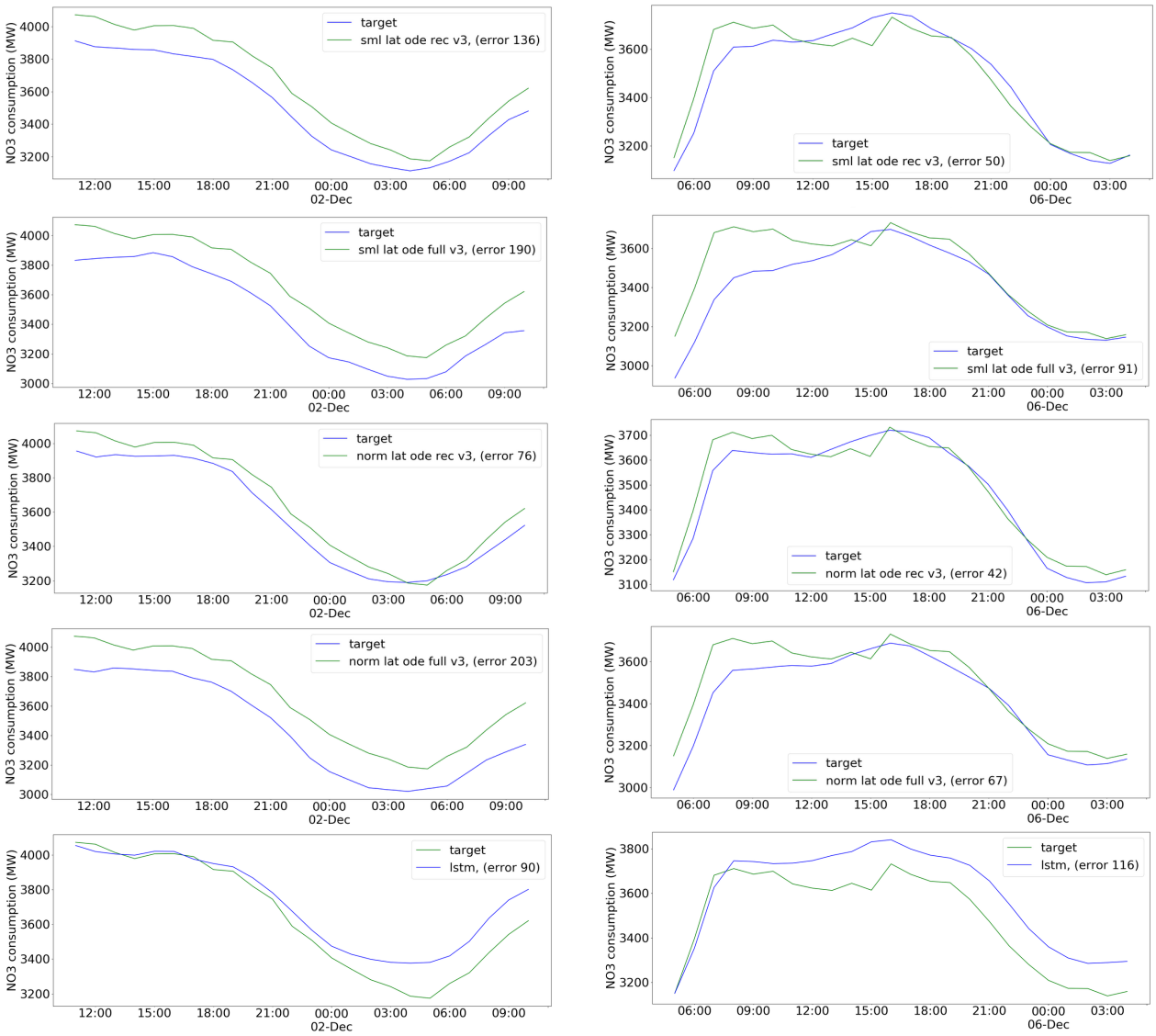


Figure 4.27: Predicted next 24 hour trajectories from different models. Consumption and MAE shown in MW.

4.4.3 Loss of NODE Models

Finally, the loss from each NODE model is shown in figure 4.28. The models were trained with the same learning rate. Observe that the models with a bigger latent state have higher instabilities and that the models with reconstruction loss only have a more frequent instability.

Figure 4.29 shows an additional test on the small latent state reconstruction only model. The model is also tested with a smaller learning rate, to compare how this affects the instability. A smaller instability is observed, but it can be argued the same oscillation is present, only with a smaller amplitude which is proportional to the learning rate.

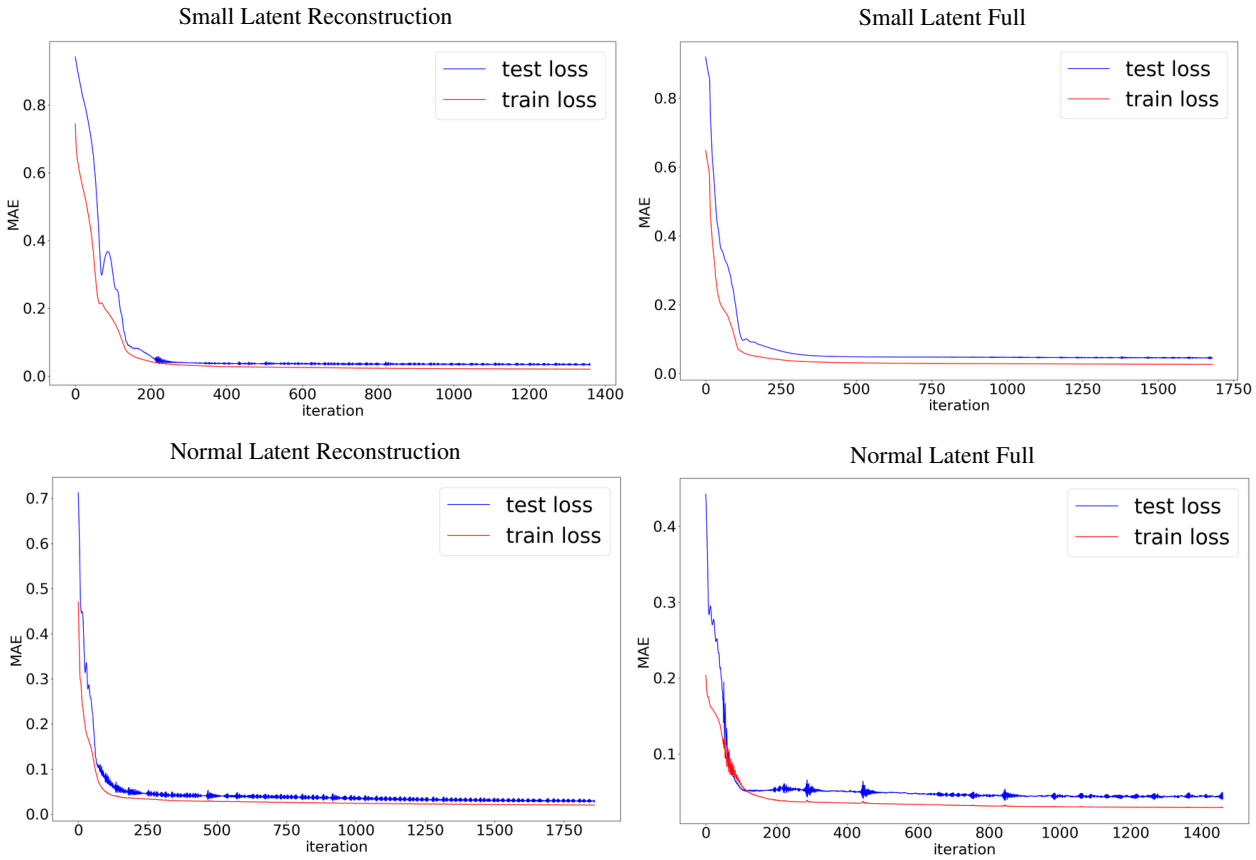


Figure 4.28: MAE on the target (consumption) of NODE models up to the iteration with best accuracy. Note the visualized loss is on the normalized test and training sets.

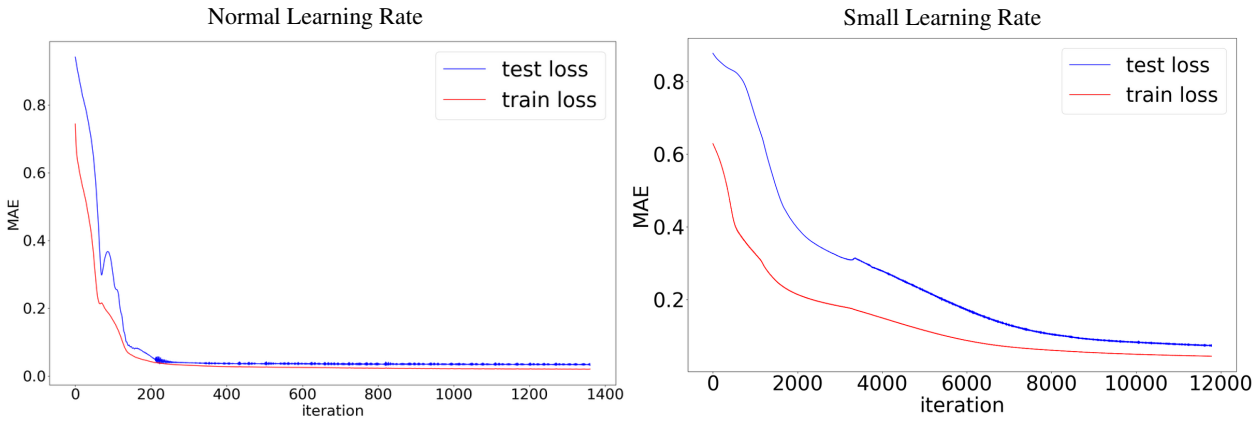


Figure 4.29: MAE of the small latent state reconstruction only model with two different learning rates (0.001 and 0.00001). Although smaller in magnitude, the same instabilities are still observed.

4.5 Summary

The main take-aways of the results from the experiments can be summarized as follows:

Experiment 1: Ytre Vikna Wind Production

1. Time-agnostic models had the best overall performance. The best NODE model had worse, but comparable, performance to the top performer which was gradient boosting.
2. No significant findings indicate NODE models reach a higher accuracy with less training data. The accuracy increases at the same speed for all the top performing models.
3. Findings show a higher instability in the training of time-aware NODE models than the ODE Regressor.

Experiment 2: Energy Consumption Energibygget

1. Time-agnostic models had the best overall performance. The best NODE model had worse, but comparable, performance to the top performer which was DNN. The Latent ODE Regressor v2 had a comparable accuracy to the time-agnostic models.
2. No significant findings indicate NODE models reach a higher accuracy with fewer parameters. In fact the opposite is shown, where DNN reaches a higher accuracy with the same amount of parameters.
3. Findings show a higher instability in the training of time-aware NODE models than the ODE Regressor.

Experiment 3: Energy Consumption NO3 Region

1. NODE models have a lower average error than LSTM for predictions on the next 24 hours and for the 24th hour ahead, but is beaten by LSTM on 1 hour ahead predictions.
2. Findings indicate training the Latent ODE Regressor v3 model with only reconstruction loss, using a bigger latent state, achieves higher accuracy.
3. Findings indicate instabilities during training are reduced when using a small latent state and using only the reconstruction error. Also, the results indicate that the instabilities are not due to a too large learning rate, although the learning rate affects the instability.

5 Evaluation and Discussion

This chapter holds an evaluation discussing to what extent the results can be trusted. What follows is a discussion of the results, and how these relate to the research questions presented in section 1.3. Finally, the chapter concludes with a discussion of the limitations of the work.

5.1 Evaluation

The biggest threat to the validity of the experiment results has to do with handling of the data. First of all, it has to be ensured (1) that data from the test set is not leaked into the training set. Second, when preparing the data, it has to be ensured (2) that target features y are not leaked into the observed features x. Third, related to the two above, it has to be ensured (3) that data points are mapped to the same time stamp across all data arrays, or at least ensure this internally for each model.

The data was handled with TrønderEnergi software, built on top of the pandas library [McKinney, 2010]. This is a widely used and well documented library, which has built in functions for correctly mapping and splitting data on date-time indices. This means splitting data into test and training data (threat 1) is easily handled and controlled. The same functionality makes threat 2 easily controlled. Further, comparing the trajectories from the machine learning models and the persistence models indicate that these threats have been controlled. The machine learning models resort to forecast the baseline value when there is no additional information to be gained from other features. It can also be observed that the baseline and target trajectories correspond well. Also, the dates in the plots correspond to the intended date range from the experiment setup, indicating the test and training sets have been split correctly. The conclusion is that, in isolation, each single run of the models is valid.

Even though the pandas library and TrønderEnergi software handles time series data well, there is one observed threat to the validity which has to be addressed. This is threat 3. In some of the results in E1 and E2 a discrepancy in the baseline value is observed. This indicates that different data was used in the test set. This was discovered after the results were run. Investigations show there were two reasons for the discrepancy. Both had to do with additional data points in the test set used for error computation. The baseline alteration could come from one or both of these faults:

1. The first fault came from different ways to handle NaN inputs. Some of the models handled this by removing the NaN entries, while others inserted a preset dummy value.
2. The second fault came from calculating lagged values. When calculating lagged values, entries are shifted according to the lag range. As a result, NaN entries are introduced in the final dataset, and the same problem as stated above occurs.

Unfortunately, due to time constraints, the results were not adjusted for this. However, investigations show that each internal run was mapped properly. This means that the target feature, baseline, and observed features were at the correct time steps relative to each other. Further, the investigation shows that these faults result in $\sim$40 additional data points at most. This is equivalent to under two days worth of data. This respectively amounts to around 0.5% and 2.5% of the test data in E1 and E2. The fact that slightly different data was used for each model, makes it harder to compare the models to each other. The investigations did not find the exact dates that are missing. These additional data points could be valid, causing no significant alteration to the MAE, or they could be corrupted points cluttering the MAE and the final reported error. However, the results might indicate that these points do not affect the final errors in a major way. To see this compare the loss of the time-aware NODE models with the time-agnostic models in E1. Assuming the random spikes seen for the time-agnostic models correspond to additional data, these points are not the main driver behind the error. This logic comes from observing that the time-aware models have higher error while not exhibiting the same spikes. This said, the spikes do contribute to the error, which means the true error for these models could be even lower. The point is that these spikes are not the *main* contributor to the loss.

Even so, it should be noted the error from the additional data only affects the comparison of the results across models, and does not compromise comparisons internally on the same model. This is relevant when studying RQ3 and RQ4. Also, as discussed, it does not compromise the validity of each single run. Considering the above, the conclusion is that the problem of additional data is not highly significant, and the results are not invalidated.

In retrospect, more time should have been dedicated to phase III (explained in section 1.4). However, as I decided to stretch out the dependant phases I and II to include a third experiment, time for phase III was suddenly scarce. It was deemed more valuable to include a third experiment, at the cost of less time for phase III. This prioritization turned out to have a small effect on the results, and weakens the validity of the findings. Either way, this prioritization was consistent with the main goal of this work, which was to perform an initial study of NODE applicability.

5.2 Discussion

The discussion evaluates each research question from section 1.3 in light of the findings from the experiments. Each of the five questions are evaluated in turn, and an answer is given based on the results obtained in the previous chapter.

RQ 1: Can NODE models consistently beat existing methods in forecasting?

This question tries to find a decisive answer on the performance of NODE models, and is the question of the biggest interest. To assess this, one has to look at results across all experiments. For a decisive answer, a NODE model would need to have the top performance in all experiments.

As can be seen in the results from E1 and E2, this is not the case. Interestingly, in E3 this is true for predictions after the 1st hour. This rises the question whether the same can be said for E1 and E2 with an extended prediction horizon. This finding is somewhat consistent with the spiral experiment by Chen et al. [2018], where the NODE model is able to extrapolate better than the RNN for time steps longer into the future. A theory is that the ODE Block in E3 is able to learn the dynamics of the trajectories better than the LSTM units.

The hypothesis was that the time-aware NODE models would outperform other models because the ODE Block would learn the dynamics of the target trajectories. However, the experiments show the opposite. In both E1 and E2 the time-aware models perform worse than the time-agnostic ones. In E3 there are no comparisons between the two. The next four paragraphs discuss possible reasons for the under-performance of the time-aware NODE models. After this, each paragraph discusses a general observation related to RQ1.

Structure of Time Series. In E1 the target trajectory resembles a random walk. This might explain, in part, why the time-aware NODE models perform worse than the time-agnostic models; there is simply no additional information to be gained from previous time steps. However, this explanation is flawed because we observe the LSTM model is able to perform comparably to time-agnostic models.

Time-Aware Representation. Although the target trajectory in E1 is close to a random walk, in E2 the trajectory has a clear seasonal pattern, and the Latent ODE still struggles with low accuracy. This further weakens the random walk argument. In E2 the Latent ODE Regressor v2 comes close to the time-agnostic models in terms of accuracy. This can be explained by the latent state of size 1 (only the target feature), for this model. In this way, the Latent ODE Regressor v2 can be considered a time-aware version of the ODE Regressor, with the only difference being the loss function. As we know from E3, a loss function without the KL-divergence results in more accurate models. This can explain why the ODE Regressor has a slightly better performance than the Latent ODE Regressor v2, as it is essentially the loss that separates the two models.

Stationary Time Relations. One reason time-aware models are needed in i.e. natural language processing is because two different input sequences might have the same semantic meaning. The time-aware models need to keep track of the importance of each word in the sequence, relative to the previous words, to decipher the meaning of the sequence. As an example, consider the sentences ”Today I learned something new” and ”I learned something new today”. Semantically, these have the same meaning, but the sequence of the words is different. Time-aware models learns the relative meaning of the past and present input, and is able to alter the relative importance. A time-agnostic model has no such capability,

and will thus struggle once these relative dependencies change places in the sequence (i.e. the two sentences above). The point of this example is to show that the words in a sentence have a different and changing importance. The language model can not know beforehand which sequence indices are the most important. Therefore, it needs to have a selective memory, processing the words and their meaning as they arrive.

For the problems in this thesis, the sequence index importancies are always more or less the same. For consumption prediction, we know that the previous value is the most important, followed by the one before that, and the one before that, following a 24 hour pattern as shown in figure 3.8. Therefore, one might argue that time-aware models have no advantage over the time-agnostic models. The time-agnostic models can just learn a rigid sequence index importancy, as the importancies are more or less equal for all inputs. This mitigates the value gained from the embedded "time-awareness" in the time-aware models. One could argue the time-aware models learn a hidden "languague" which is not understandable in the problems in this thesis. However, the results suggest this is not the case since the errors are larger for the time-aware models. This said, for E1 we see spikes for time-agnostic models wheras the time-aware models do not have these, suggesting added value from incorporating time-awareness. On the other hand, these spikes might be corrupted values originating from the additional data points problem discussed in section 5.1, which were removed from the test data used on the time-aware models.

Model Prediction Properties. An alternative explanation to the under-performance of the time-aware models is revealed by looking at the underlying model properties in figure 3.9. Since the Latent ODE is trying to predict all the input features (wind speed, wind direction, and production) there is a trade-off between the true target feature and the other inputs. Thus, the target feature predictions are cluttered, since the model prioritizes other features. This cluttering is reflected by the high instability in the MAE plotted in figure 4.11, which only considers the target feature. The multiple predictions argument is valid for the Latent ODE, but the Latent ODE Regressor v1 eliminates this problem by only predicting the target feature, yet the loss is still unstable. A clue to an explanation is given in E3. From E3 we know the instability is higher for models with a larger latent space and a loss function which includes both reconstruction error and KL-divergence. This does not explain the under-performance, since the error from different loss functions in E3 is not of such a large magnitude as is exhibited here. However, it tells us the performance might be related to the ODE Block in the latent model. Findings from related work might suggest this is the case. Dupont et al. [2019] and Gholami et al. [2019] also investigate instability in the ODE Block. Dupont et al. [2019] show that there are some simple functions which an ODE Block can not learn, while Gholami et al. [2019] show there exists scenarios which causes erroneous gradient calculations. These explanations might also be valid for the time-aware NODE models.

Complex Model Operations. Where the LSTM performance is good in E1, the LSTM model struggles with high errors in E2. Since the random walk pattern is changed to a seasonal pattern, one could imagine P1 would be easier than P2. However, there are more factors than the target trajectory pattern that decide the difficulty of a problem. The natural first explanation to look for in any machine learning problem is the dataset. In the results section we observed the LSTM models had difficulties predicting the peaks in consumption. This is not the case for the Latent ODE Regressor v2 and the DNN. So what is the difference in how these models handle the input data? One could say the LSTM is a more advanced and complicated model. The LSTM units in the encoder (as opposed to the RNN units in the NODE models) keep track of multiple inner states and also perform multiple operations. Therefore, an explanation is that these advanced operations clutter the predictions. This suggests simpler models components such as the ones used in the Latent ODE Regressor v2 are more optimal than LSTM components for this problem.

Recurrent Units in the Decoder. A similar LSTM dip in performance is observed in E3 when forecasting the 24th hour ahead, where the LSTM is beaten by the Latent ODE Regressor v3. Again, one can look to model architecture for an explanation. The LSTM has recurrent units in the decoder, while the NODE model only has feed-forward layers. If the dynamics are not learned, these recurrent units can confuse the signal and result in a larger error. This again suggests simpler units in the decoder yield a better performance.

Optimizing for All Hours at the Same Time. A surprising result in E3 occurs on the first run of the Latent ODE Regressor with normal latent state and full loss. This run has a higher error for the 1st hour predictions than for the 24th hour. Intuitively, predicting one hour ahead is easier than predicting 24 hours ahead. This is also reflected in the baseline error being higher for the 24th hour. This indeed seems to be the case, as all the other runs and models have a higher error for the 24th hour. To explain how this can happen we look at how the model is trained. At each iteration the model predicts values for all the next 24 hours at the same time. Then, the parameters of the model are

updated according to the loss function and the calculated gradients. This means the final loss does not discriminate on hours, but treat them as equally important. Remember the weights are initialized randomly, and the optimization algorithm is greedy. Taking all these factors into account means that it is possible for the algorithm to pursue a path which ”prioritizes” the 24th hour, leaving the 1st hour forecast unoptimized. As long as the total loss for the other hours is lower, this is a rational choice for the algorithm. However, the results suggests the parameter space where this is the case is small, as this scenario only occurred once during the 15 runs.

RQ 2: Which NODE model is the most suiting?

While the previous question relates NODE models up against existing models, this question seeks to find out which of the NODE models are best among each other. The question of ”suiting” is a bit vague. Therefore, and consistent with the goal of this thesis, accuracy is the primary consideration when evaluating what model is the most suiting. This said, the model properties in figure 3.9 have to be taking into consideration as well, especially the possibility to forecast multiple hours. To summarize, the optimal model achieves a high accuracy, with few parameters, less training data, and is able to forecast multiple hours. Considering only a single hour forecast as in E1 and E2, the answer is decisive in favour of the ODE Regressor. It achieves a higher accuracy, with fewer parameters, smaller deviation in average values, and more stable training. However, it is unfortunate that the only experiment where a NODE model has the lowest error, there are no comparisons between NODE models. In E3 the Latent ODE Regressor v3 beat the benchmark LSTM model. This model is also flexible in predictions for multiple hours. Depending on the problem at hand, we are thus left with the ODE Regressor and the Latent ODE Regressor v3. Previously, an argument was made for the two models being the same model, only separated by the loss function. Therefore, it seems the most suiting model would be the Latent ODE Regressor with reconstruction loss only. This would possibly render the two models equivalent in terms of accuracy, with one having the ability to predict multiple time steps. Alternatively, based on the observation that time-agnostic models have a higher accuracy, one can consider multiple ODE Regressor models, each specialized at forecasting a specific hour. Parameter-wise this is not totally intractable, as the time-agnostic models have about 10-20 times fewer parameters in general. However, a multiple model solution is not the most elegant one. It might also introduce additional error related to coordinating the models.

RQ 3: Does the claim on NODE needing less training data hold?

This research question tries to verify the findings made by Chen et al. [2018] claiming NODE can achieve the same accuracy with less training data. If this claim holds, NODE models should show a higher accuracy with less data. To ensure accuracy is indeed because of available data, the increase in accuracy when adding more data should be less for the NODE models than for other models. This was investigated in E1, and the results show that this claim holds to a certain degree. Comparing the ODE Regressor to DNN the accuracy at 1 month of data is higher for the ODE Regressor, while the accuracy is comparable for 6 and 12 months. This said, comparing the ODE Regressor to LSTM and gradient boosting the same observation does not hold. There exist some possible explanations for this. First of all, the problem studied by Chen et al. [2018] was synthetic, consisting of two observed features with known dynamics. Problems in this thesis have 10 times and more observed features. Also, it is unknown if exact underlying dynamics even exist. A second explanation for the different observations can be the data range which was tested. The range which was tested was $[\sim 720, \sim 8640]$ (for comparison the NODE used by Chen et al. [2018] was tested on the range $[\sim 300, \sim 1000]$). Had E1 tested a different data range, the results might have had a different outcome.

RQ 4: Does the claim on NODE needing fewer parameters hold?

This research question tries to verify the findings from Chen et al. [2018] claiming NODE can achieve the same accuracy with fewer parameters. This was investigated in E2, and the results show that this claim does not hold. One explanation for this could be the size of the input feature space. Chen et al. [2018] tested NODE on image data. Images have a bigger feature space than the problems investigated in this thesis. In general, this means a machine learning model needs a greater number of parameters. With a greater parameter space there is also a greater room for different configurations of parameters. Assuming the distributions of these spaces are the same, there is a greater room for efficient representation in the bigger space, although this means the space with poor representation is also

bigger. However, greedy optimization algorithms are efficient at eliminating the poorest spaces. In general, finding local minimas is easy, but finding global optimas is hard. This effect can be seen by looking at the reported loss of the models. The loss drops quickly to begin with, but stagnates once a certain accuracy is achieved.

Another explanation closely related to the previous, regards what parameter range was explored. The lowest range tested was $[\sim 500, \sim 1100]$ (for comparison the models used by Chen et al. [2018] had 0.22M parameters). Maybe if the models had been tested with less, say 200, parameters the results would have shown that NODE indeed is able to reach a higher accuracy. However, the "small" ODE Regressor was at the smallest possible configuration, in terms of layers and parameters, and shrinking it further would be artificial.

A third argument that might have altered the outcome is model configurations. It is possible to create models with other configurations, but with the same number of parameters. One way to do this is using other model components. This work uses other model components than those used by Chen et al. [2018]. This difference is not reflected in the parameter count. Chen et al. [2018] uses convolutional components, while this work uses fully connected, and recurrent layers. Another way to alter the network configuration, while keeping the parameter count the same, is by adding more layers and reducing the number of nodes in each layer.

RQ 5: What part of the latent representation in the Latent ODE Regressor v3 model is the most important for accurate forecasts?

This question arose during E3. It was noticed the Latent ODE Regressor v3 had considerably worse performance on a 1 hour ahead forecast compared to v2 of the same model in E2. Although P3 is different from P2, they are both consumption forecasting problems with comparable input data. The main difference from model v1 and v2 is the size of the latent state. Later, it was also observed that the ODE Regressor has similarities with the two models, with the main difference being the defined loss function. These considerations led to the comparisons on latent state size and loss definition in E3. Results from E3 indicate that a larger latent space with only reconstruction loss gives the lowest error. It is well known that a model with more parameters is able to approximate more complex functions. The function mapping the input data to the consumption is thus approximated better when the larger state space is used. In that sense, these findings make sense. Although, as results from E1 and E2 show this is not always the case, where the smaller models have a better approximation. This can be explained by overfitting. The larger models learn a more complex function approximation from the training set, which is a worse *generalizing* function to unseen samples. For an explanation of why the loss matters, we look at what is achieved by including the KL-divergence as a model regularizer. The KL-divergence is meant to force the distributions of the latent state $P(z)$ to be Gaussians with mean 0 and deviation of 1. This ensures the latent state features are continuous in space, and similar observed samples x lie close in latent space. This objective is incorporated so that generating synthetic samples is less problematic. However, in the problems studied in this thesis, generating synthetic samples is not an objective. This means the KL-divergence optimizes for an additional objective we are not interested in, possibly resulting in a poorer performance on the objective of interest. Note that a deviation from the objective has occurred earlier. The Latent ODE model optimizes for all input features, causing a disturbance to the target feature of interest. The KL-divergence might be a more subtle way of interfering with the objective.

5.3 Limitations

This section highlights some of the limitations of this work.

Data

The most important factor in any machine learning problem is the data used. Without good data, the results will not be satisfying. The experiments used three datasets provided by TrønderEnergi. The validity of the provided data is an underlying assumption throughout this work.

Number of Runs for Statistical Significance

A known challenge when training machine learning models is the stochastic behaviour leading to the final trained model. This work addresses this stochasticity by running the experiments several times and reporting the average. However, it should be noted that three times is likely not enough for a statistically sound evaluation. This said, it does give an indication of the volatility of the method. Due to time constraints a trade-off between adding E3 versus completing more runs was made.

Impact of Additional Data Points

As discussed in section 5.1 there was some discrepancy in the data points used for testing. The impact these additional data points have on the final results is not entirely known. However, results suggest the impact is not major. The discussion also arguments it only affects certain parts of the results.

6 Conclusion and Future Work

6.1 Conclusion

This thesis performed an initial feasibility study of a novel family of neural nets called Neural Ordinary Differential Equations applied to three forecasting problems in the energy sector. The hypothesis was that NODE models will have a special advantage by learning the underlying dynamics of the relation between input and target data. Over the course of three experiments, five NODE models were tested and compared against existing benchmark models. The results indicate the Latent ODE Regressor v3 models can outperform other methods on 24 hours ahead forecasts, and perform comparably to existing machine learning models on 1 hour ahead forecasts. However, as all existing models were not tested for 24 hours ahead forecasts, to verify this more experimentation is needed. Also, findings suggest NODE models have a higher accuracy when using only reconstruction loss during training. Implementing this in future experiments could yet increase NODE performance on 1 hour ahead forecasts. Property claims on amount of data and number of parameters were not confirmed for the problems studied in this work. Finally, as the findings are promising, the conclusion is that further experimentation on NODE could lead to valuable results.

6.2 Contributions

1. Five different NODE models are tested on three different energy forecasting problems, and compared against existing machine learning models. Results indicate the NODE models can outperform existing models on 24 hour ahead forecasts, and perform comparably on 1 hour ahead forecasts.

6.3 Future Work

This section provides ideas for low hanging fruits (in no specific order) of future work:

- **Experiment with existing models.** Feature engineering of machine learning models is as much of an art as a science. Although the field has evolved a lot over the past years, in many ways feature engineering of the layers and hyperparameters is still a try-and-fail process. One could experiment with settings of the different networks to try and improve the accuracy of the models. For instance, adding recurrent components in the ODE regressor would make them more suitable for day-ahead forecasting. The results also suggest using the Latent ODE Regressor v2 and v3 with a different loss function might increase the accuracy of P1 and P2.
- **Implement new NODE models.** Over the course of this thesis new papers have been published. An interesting direction for future work would be to implement these new models. One of the challenges with the NODE models in this thesis was the instability during training. This instability is addressed by papers mentioned in section 2.3, proposing AugNODE and AdNODE. Implementing these networks could show additional accuracy improvements.
- **Research training time.** This work explores only how the number of training parameters and training data affects accuracy compared to other models. Another direction to take would be to compare training time to other models. This requires a higher level of technical detail from the experiment environment, and was not emphasized in this work. However, it is noted that observed training time of the time-aware NODE models was, in general, considerably higher than for the other models. An objective for future work could be to optimize training time for performance.

- **Apply NODE to New Problems.** This work only examines three specific problems. It could be interesting to apply NODE models to new problems. We have seen that NODE shows promising results on day-ahead forecasting. Testing for even longer time periods or the same period on different problems could be a target for future work. Also, investigating whether specific time series patterns (other than the ones studied in this thesis) impact the performance could be of interest.

Bibliography

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Chang, B., Meng, L., Haber, E., Ruthotto, L., Begert, D., and Holtham, E. (2017). Reversible architectures for arbitrarily deep residual neural networks. *CoRR*, abs/1709.03698.

Che, Z., Purushotham, S., Cho, K., Sontag, D., and Liu, Y. (2018). Recurrent Neural Networks for Multivariate Time Series with Missing Values. *Scientific Reports*.

Chen, R. (2019). Torchdiffeq. https://github.com/rtqichen/torchdiffeq/.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 6572–6583. Curran Associates, Inc.

Choi, E., Bahadori, M. T., Schuetz, A., Stewart, W. F., and Sun, J. (2016). Doctor ai: Predicting clinical events via recurrent neural networks. In Doshi-Velez, F., Fackler, J., Kale, D., Wallace, B., and Wiens, J., editors, *Proceedings of the 1st Machine Learning for Healthcare Conference*, volume 56 of *Proceedings of Machine Learning Research*, pages 301–318, Children's Hospital LA, Los Angeles, CA, USA. PMLR.

Ciccone, M., Gallieri, M., Masci, J., Osendorfer, C., and Gomez, F. J. (2018). Nais-net: Stable deep networks from non-autonomous differential equations. *CoRR*, abs/1804.07209.

Doersch, C. (2016). Tutorial on variational autoencoders. cite arxiv:1606.05908.

Dorogush, A. V., Ershov, V., and Gulin, A. (2018). Catboost: gradient boosting with categorical features support. *CoRR*, abs/1810.11363.

Dupont, E., Doucet, A., and Whye Teh, Y. (2019). Augmented Neural ODEs. *arXiv e-prints*, page arXiv:1904.01681.

Friedman, J. H. (2002). Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378.

Gers, F. A., Schmidhuber, J., and Cummins, F. A. (2000). Learning to forget: Continual prediction with lstm. *Neural Computation*, 12:2451–2471.

Gholami, A., Keutzer, K., and Biros, G. (2019). ANODE: unconditionally accurate memory-efficient gradients for neural odes. *CoRR*, abs/1902.10298.

Gholami, A., Kwon, K., Wu, B., Tai, Z., Yue, X., Jin, P. H., Zhao, S., and Keutzer, K. (2018). Squeezenext: Hardware-aware neural network design. *CoRR*, abs/1803.10615.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Grathwohl, W., Chen, R. T. Q., Bettencourt, J., and Duvenaud, D. (2019). Scalable reversible generative models with free-form continuous dynamics. In *International Conference on Learning Representations*.

Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778.

Ha, D., Dai, A. M., and Le, Q. V. (2016). Hypernetworks. *CoRR*, abs/1609.09106.

Haber, E. and Ruthotto, L. (2017). Stable architectures for deep neural networks. *CoRR*, abs/1705.03341.

Hairer, E., Nørsett, S., and Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Identity mappings in deep residual networks. *CoRR*, abs/1603.05027.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer, S. C. and Kolen, J. F., editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Jimenez Rezende, D. and Mohamed, S. (2015). Variational Inference with Normalizing Flows. *arXiv e-prints*, page arXiv:1505.05770.

Keras-Team (2019). Keras. https://github.com/keras-team/keras/.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *CoRR*, abs/1312.6114.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Kutta, W. (1901). Beitrag zur näherungsweisen integration totaler differentialgleichungen. *Zeitschrift für Mathematik und Physik*, 46:435–453.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323.

Liao, Q. and Poggio, T. A. (2016). Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *CoRR*, abs/1604.03640.

M4-Competition (2019). M4 forecasting competition web page. https://www.mcompetitions.unic.ac.cy.

Makridakis, S., Spiliotis, E., and Assimakopoulos, V. (2018). The m4 competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*.

McKinney, W. (2010). Data structures for statistical computing in python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

Microsoft (2019). Azure databricks. https://docs.microsoft.com/en-us/azure/azure-databricks/.

Mitchell, T. M. (1997). *Machine Learning*. The McGraw-Hill Book Companies.

Ng, A. Y. and Jordan, M. I. (2002). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In Dietterich, T. G., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems 14*, pages 841–848. MIT Press.

NVE (2019). Wholesale market structure. https://www.nve.no/energy-market-and-regulation/wholesale-market/wholesale-market-timeframes/.

Pontryagin, L. S., Boltyanskii, V. G., Gamkrelidze, R. V., and Mishchenko, E. F. (1962). *The mathematical theory of optimal processes*. Wiley, New York, NY.

Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1278–1286, Bejing, China. PMLR.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Runge, C. (1895). Ueber die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178.

Russell, S. and Norvig, P. (2014). *Artificial Intelligence A Modern Approach*. Pearson Education Limited, 3rd edition.

Ruthotto, L. and Haber, E. (2018). Deep neural networks motivated by partial differential equations. *CoRR*, abs/1804.04272.

Smyl, S., Ranganathan, J., and Pasqua, A. (2019). M4 forecasting competition: Introducing a new hybrid es-rnn model. https://eng.uber.com/m4-forecasting-competition/.

Starmer, J. (2019). Gradient boost part 1: Regression main ideas. https://www.youtube.com/watch?v=3CC4N4z3GJc.

Weinan, E. (2017). A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1).

Zhang, T., Yao, Z., Gholami, A., Keutzer, K., Gonzalez, J., Biros, G., and Mahoney, M. W. (2019). ANODEV2: A coupled neural ODE evolution framework. *CoRR*, abs/1906.04596.

Appendix

NODE Model Details

Experiment 1

ODE Regressor

```
feature_layers = [ODEBlock(ODEfunc(num_vars))] if is_odenet else [ResBlock(64, 64) for _ in range(6)]
fc_layers = [nn.Linear(num_vars, 1)]

model = nn.Sequential(OrderedDict([
                                   ('feature_layers', *feature_layers),
                                   ('fc_layers', *fc_layers)
                                  ])).to(device, dtype=torch.float32)
```

```
class ODEfunc(nn.Module):

    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.h1 = nn.Linear(num_vars, num_vars)
        self.relu = nn.ReLU(inplace=True)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.h1(x)
        out = self.relu(out)
        return out


class ODEBlock(nn.Module):

    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc
        self.integration_time = torch.tensor([0, 1]).float()

    def forward(self, x):
        self.integration_time = self.integration_time.type_as(x)
        out = odeint(self.odefunc, x, self.integration_time, rtol=tol, atol=tol)
        return out[1]

    @property
    def nfe(self):
        return self.odefunc.nfe

    @nfe.setter
    def nfe(self, value):
        self.odefunc.nfe = value
```

Latent ODE

```python
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```python
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, obs_dim)

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```python
nhidden = 20 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Latent ODE Regressor v1

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc3(out)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```
class Decoder(nn.Module):

    def __init__(self, latent_dim, pred_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, int (nhidden/2))
        self.fc4 = nn.Linear(int (nhidden/2), int (nhidden/4))
        self.fc5 = nn.Linear(int (nhidden/4), pred_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        out = self.relu(out)
        out = self.fc5(out)
        return out
```

Experiment 2

Latent ODE ”small”

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden - 4)
        self.fc3 = nn.Linear(nhidden - 4, obs_dim)

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```
nhidden = 20 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 5 #number of hidden nodes in RNN recognition net
```

Latent ODE ”medium”

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, obs_dim)

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```
nhidden = 20 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Latent ODE "large"

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, obs_dim)

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, nhidden - 2)
        self.fc5 = nn.Linear(nhidden - 2, nhidden - 2)
        self.fc6 = nn.Linear(nhidden - 2, nhidden - 6)
        self.fc7 = nn.Linear(nhidden - 6, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc3(out)
        out = self.elu(out)
        out = self.fc4(out)
        out = self.elu(out)
        out = self.fc5(out)
        out = self.elu(out)
        out = self.fc6(out)
        out = self.elu(out)
        out = self.fc7(out)
        return out
```

```
nhidden = 35 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Experiment 2

ODE Regressor ”small”

```
class ODEfunc(nn.Module):

    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.h1 = nn.Linear(num_vars, num_vars)
        self.relu = nn.ReLU(inplace=True)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.h1(x)
        out = self.relu(out)
        return out



class ODEBlock(nn.Module):

    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc
        self.integration_time = torch.tensor([0, 1]).float()

    def forward(self, x):
        self.integration_time = self.integration_time.type_as(x)
        out = odeint(self.odefunc, x, self.integration_time, rtol=tol, atol=tol)
        return out[1]

    @property
    def nfe(self):
        return self.odefunc.nfe

    @nfe.setter
    def nfe(self, value):
        self.odefunc.nfe = value
```

```
feature_layers = [ODEBlock(ODEfunc(num_vars))] if is_odenet else [ResBlock(64, 64) for _ in range(6)]
fc_layers = [nn.Linear(num_vars, 1)]

model = nn.Sequential(OrderedDict([
                                   ('feature_layers', *feature_layers),
                                   ('fc_layers', *fc_layers)
                                  ])).to(device, dtype=torch.float32)
```

ODE Regressor ”medium”

```
class ODEfunc(nn.Module):

    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.h1 = nn.Linear(num_vars, num_vars-4)
        self.h5 = nn.Linear(num_vars-4, num_vars)
        self.relu = nn.ReLU(inplace=True)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.h1(x)
        out = self.relu(out)
        out = self.h5(out)
        return out



class ODEBlock(nn.Module):

    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc
        self.integration_time = torch.tensor([0, 1]).float()

    def forward(self, x):
        self.integration_time = self.integration_time.type_as(x)
        out = odeint(self.odefunc, x, self.integration_time, rtol=tol, atol=tol)
        return out[1]

    @property
    def nfe(self):
        return self.odefunc.nfe

    @nfe.setter
    def nfe(self, value):
        self.odefunc.nfe = value
```

```
feature_layers = [ODEBlock(ODEfunc(num_vars))] if is_odenet else [ResBlock(64, 64) for _ in range(6)]
fc_layers = [nn.Linear(num_vars, 1)]

model = nn.Sequential(OrderedDict([
                                   ('feature_layers', *feature_layers),
                                   ('fc_layers', *fc_layers)
                                  ])).to(device, dtype=torch.float32)
```

ODE Regressor ”large”

```
class ODEfunc(nn.Module):

    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.h1 = nn.Linear(num_vars, num_vars)
        self.h2 = nn.Linear(num_vars, num_vars)
        self.relu = nn.ReLU(inplace=True)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.h1(x)
        out = self.relu(out)
        out = self.h2(out)
        return out



class ODEBlock(nn.Module):

    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc
        self.integration_time = torch.tensor([0, 1]).float()

    def forward(self, x):
        self.integration_time = self.integration_time.type_as(x)
        out = odeint(self.odefunc, x, self.integration_time, rtol=tol, atol=tol)
        return out[1]

    @property
    def nfe(self):
        return self.odefunc.nfe

    @nfe.setter
    def nfe(self, value):
        self.odefunc.nfe = value
```

```
feature_layers = [ODEBlock(ODEfunc(num_vars))] if is_odenet else [ResBlock(64, 64) for _ in range(6)]
fc_layers = [nn.Linear(num_vars, 1)]

model = nn.Sequential(OrderedDict([
                                   ('feature_layers', *feature_layers),
                                   ('fc_layers', *fc_layers)
                                  ])).to(device, dtype=torch.float32)
```

Latent ODE Regressor ”small”

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, var_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim + var_dim, nhidden - 4)
        self.fc3 = nn.Linear(nhidden - 4, obs_dim)

    def forward(self, z, x):
        combined = torch.cat((z, x), dim=2)
        out = self.fc1(combined)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```
nhidden = 20 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 5 #number of hidden nodes in RNN recognition net
```

Latent ODE Regressor ”medium”

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, var_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim + var_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, obs_dim)

    def forward(self, z, x):
        combined = torch.cat((z, x), dim=2)
        out = self.fc1(combined)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc4(out)
        return out
```

```
nhidden = 20 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Latent ODE Regressor ”large”

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)


class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, var_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim + var_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, obs_dim)

    def forward(self, z, x):
        combined = torch.cat((z, x), dim=2)
        out = self.fc1(combined)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, nhidden - 2)
        self.fc5 = nn.Linear(nhidden - 2, nhidden - 2)
        self.fc6 = nn.Linear(nhidden - 2, nhidden - 6)
        self.fc7 = nn.Linear(nhidden - 6, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc3(out)
        out = self.elu(out)
        out = self.fc4(out)
        out = self.elu(out)
        out = self.fc5(out)
        out = self.elu(out)
        out = self.fc6(out)
        out = self.elu(out)
        out = self.fc7(out)
        return out
```

```
nhidden = 35 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Experiment 3

Latent ODE Regressor Small

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)
```

```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc3(out)
        return out
```

```
class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, var_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim + var_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden-10)
        self.fc3 = nn.Linear(nhidden-10, nhidden-20)
        self.fc4 = nn.Linear(nhidden-20, obs_dim)

    def forward(self, z, x):
        combined = torch.cat((z, x), dim=2)
        out = self.fc1(combined)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        return out
```

```
nhidden = 30 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Latent ODE Regressor Normal

```
class RecognitionRNN(nn.Module):

    def __init__(self, latent_dim, obs_dim, nhidden, nbatchtest, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.nbatchtest = nbatchtest
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h

    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)

    def initHiddenTest(self):
        return torch.zeros(self.nbatchtest, self.nhidden)
```

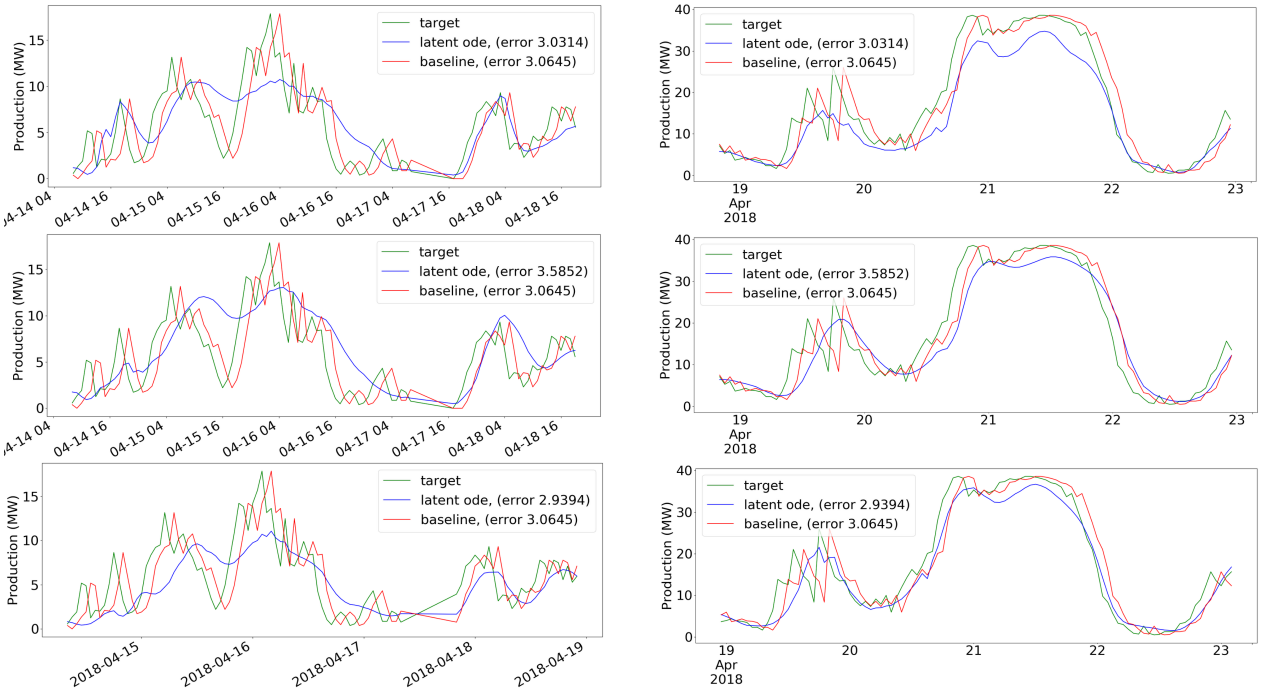
```
class LatentODEfunc(nn.Module):

    def __init__(self, latent_dim, nhidden):
        super(LatentODEfunc, self).__init__()
        self.elu = nn.ELU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.elu(out)
        out = self.fc2(out)
        out = self.elu(out)
        out = self.fc3(out)
        return out
```

```
class Decoder(nn.Module):

    def __init__(self, latent_dim, obs_dim, var_dim, nhidden):
        super(Decoder, self).__init__()
        self.latent_dim = latent_dim
        self.var_dim = var_dim
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim + var_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden-10)
        self.fc3 = nn.Linear(nhidden-10, nhidden-20)
        self.fc4 = nn.Linear(nhidden-20, obs_dim)

    def forward(self, z, x):
        combined = torch.cat((z, x), dim=2)
        out = self.fc1(combined)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        return out

nhidden = 30 # number of hidden nodes in ODEfunc and in decoder net
rnn_nhidden = 20 #number of hidden nodes in RNN recognition net
```

Appendix B: Additional Results

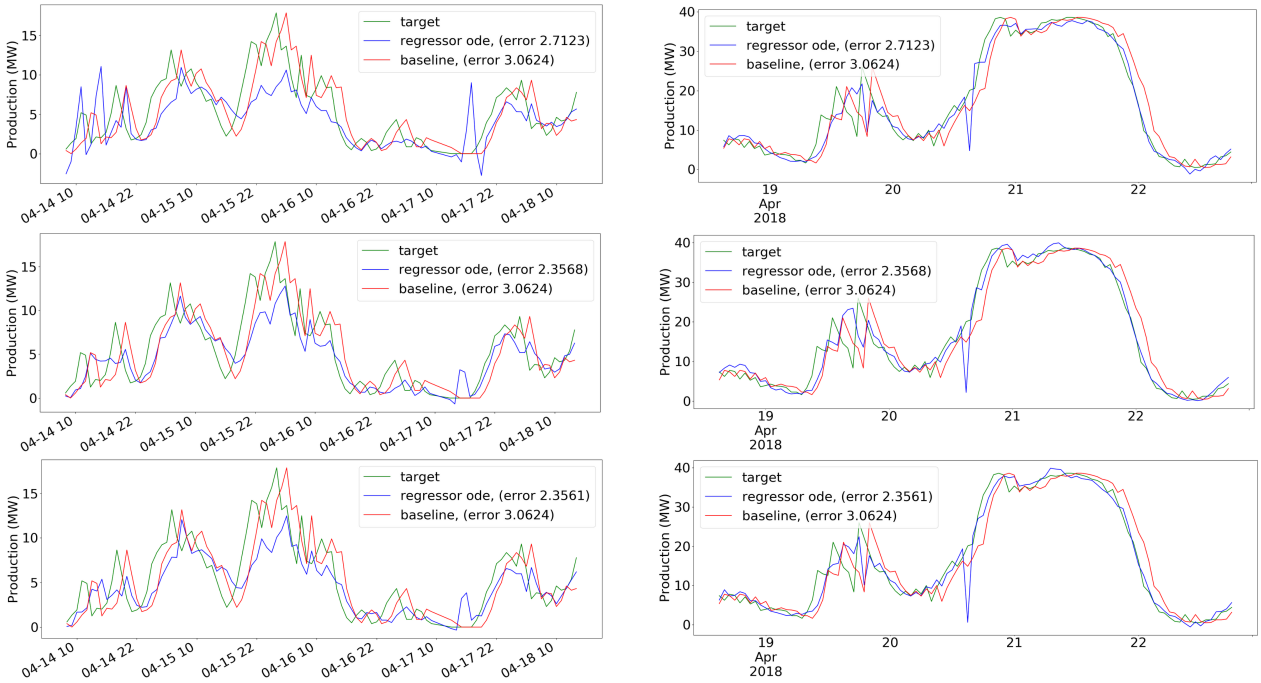
Experiment 1

Latent ODE



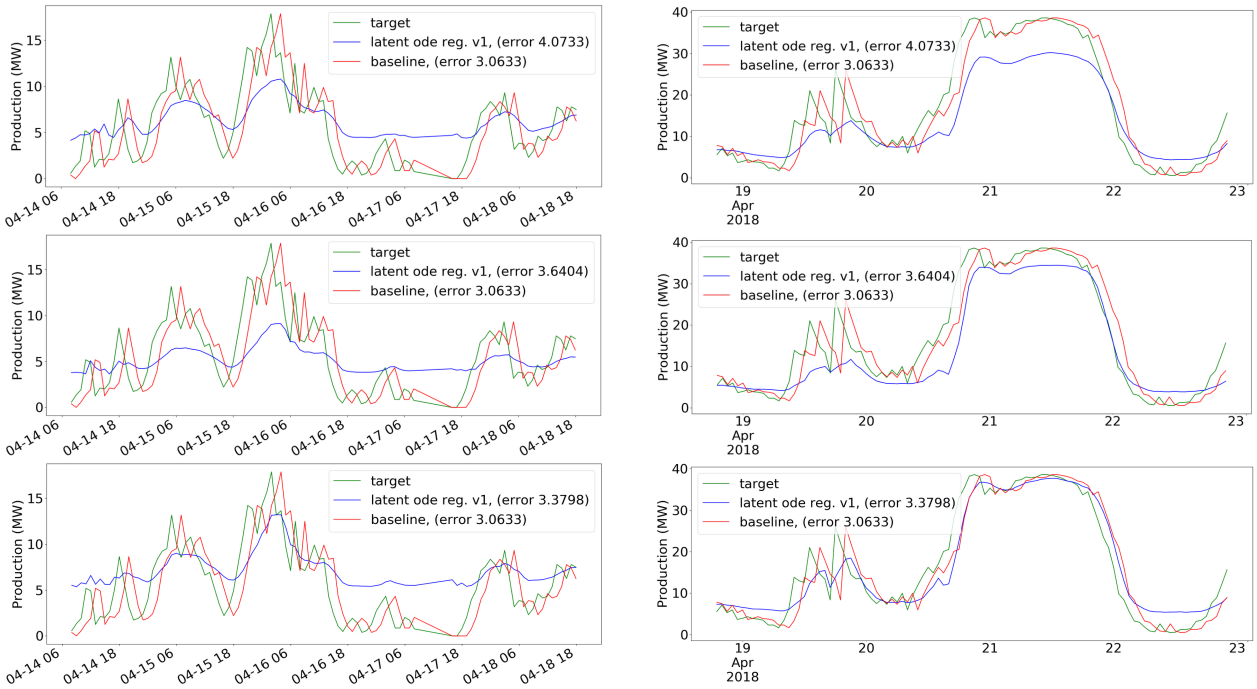
E1: Sampled Latent ODE predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

ODE Regressor



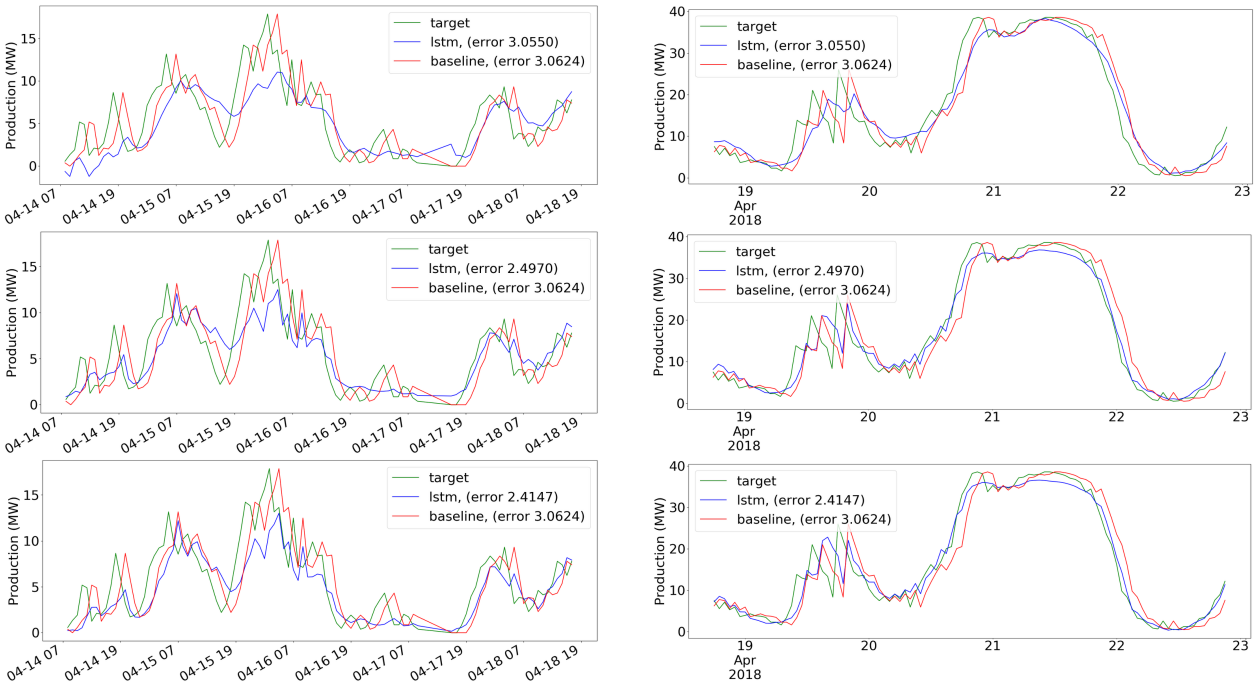
E1: Sampled ODE Regressor predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months.

Latent ODE Regressor v1



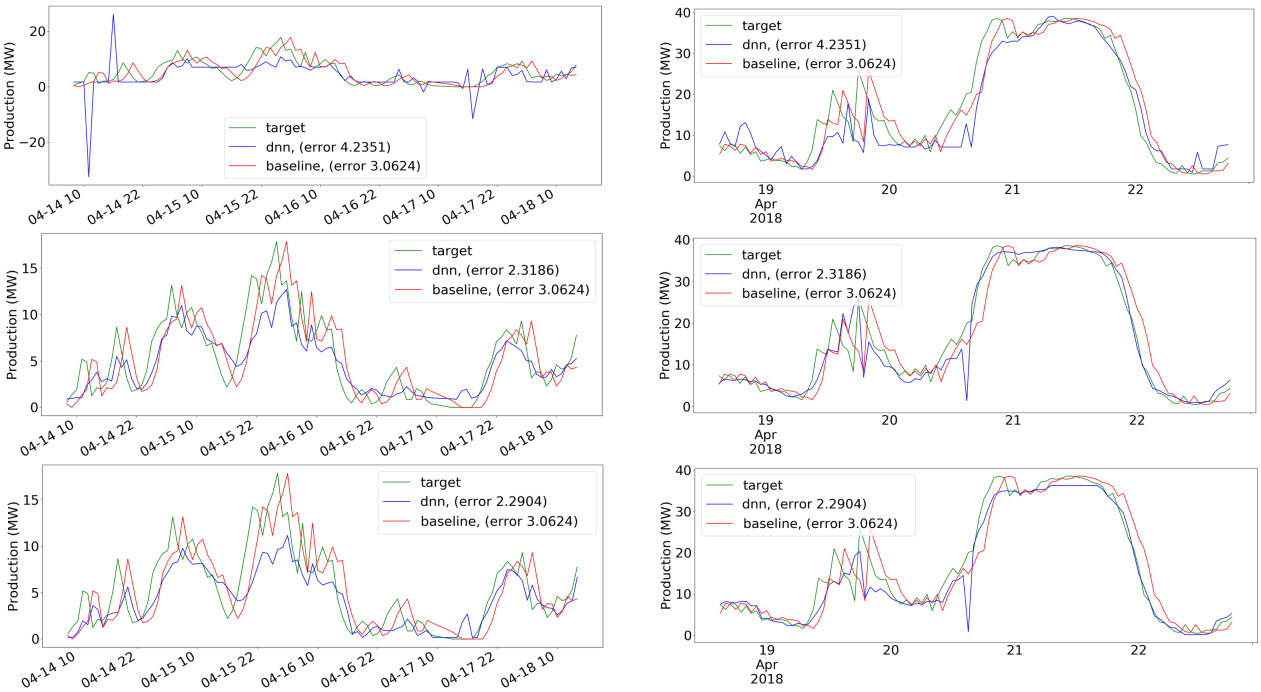
E1: Sampled Latent ODE Regressor v1 predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months

LSTM



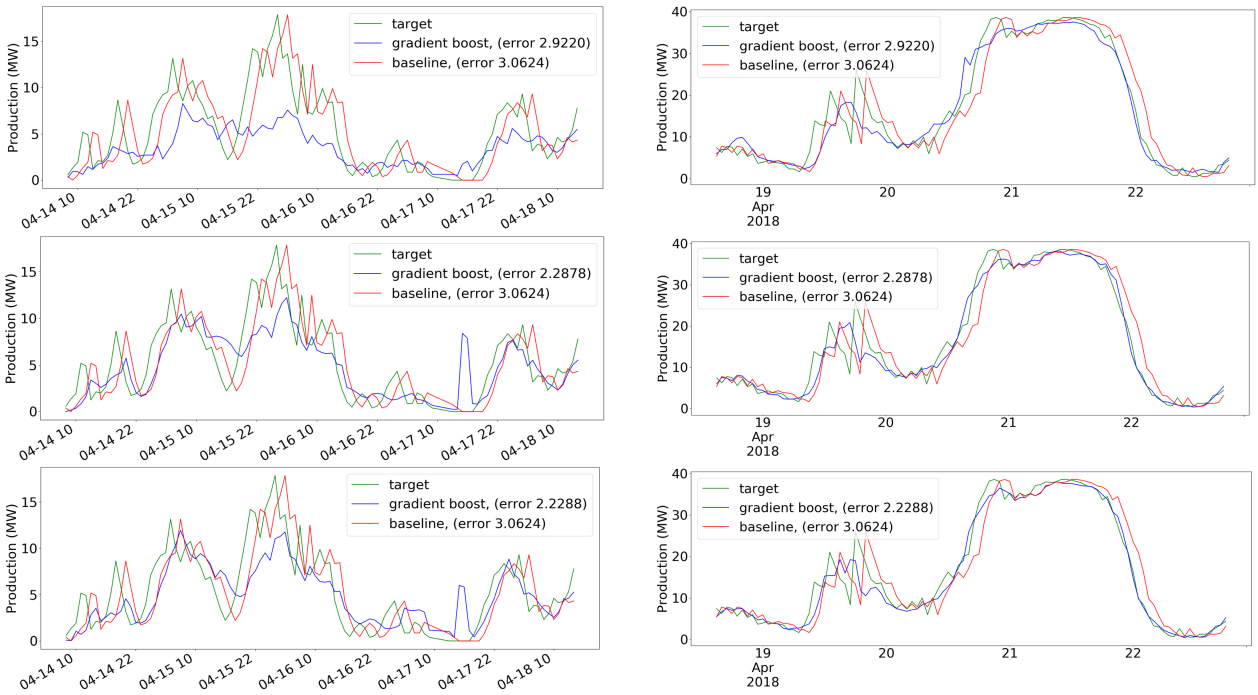
E1: Sampled LSTM predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months

DNN



E1: Sampled DNN predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months

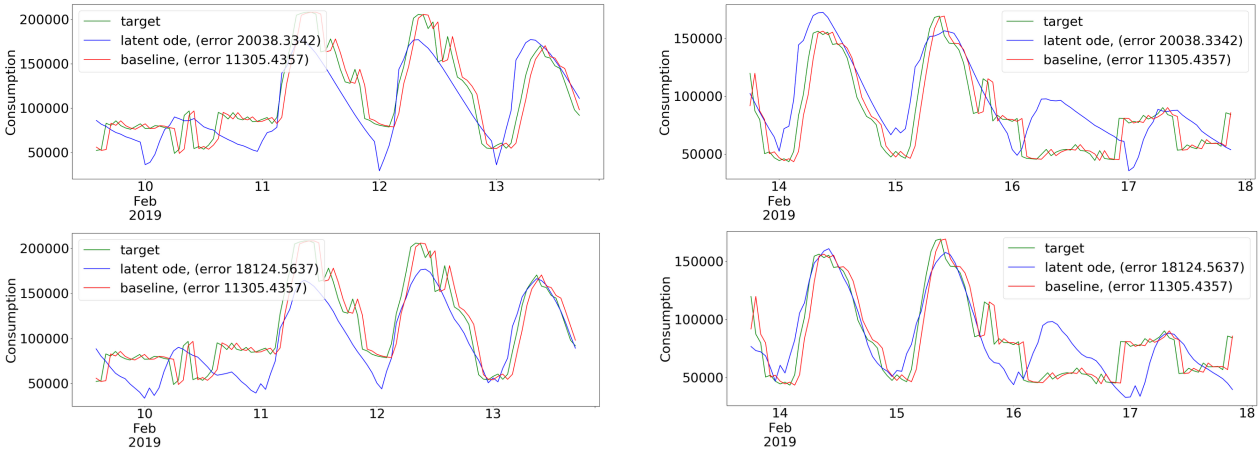
Gradient Boosting



E1: Sampled gradient boosting predictions on the test set. Top 1 month. Middle 6 months. Bottom 12 months

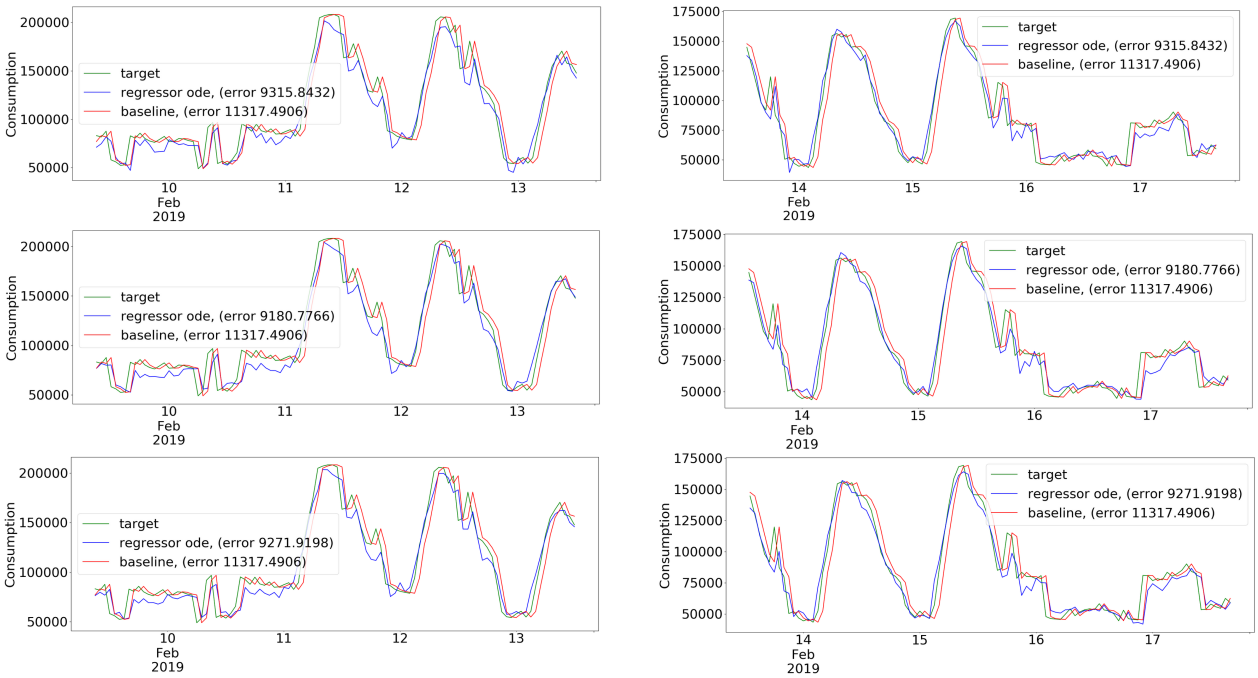
Experiment 2

Latent ODE



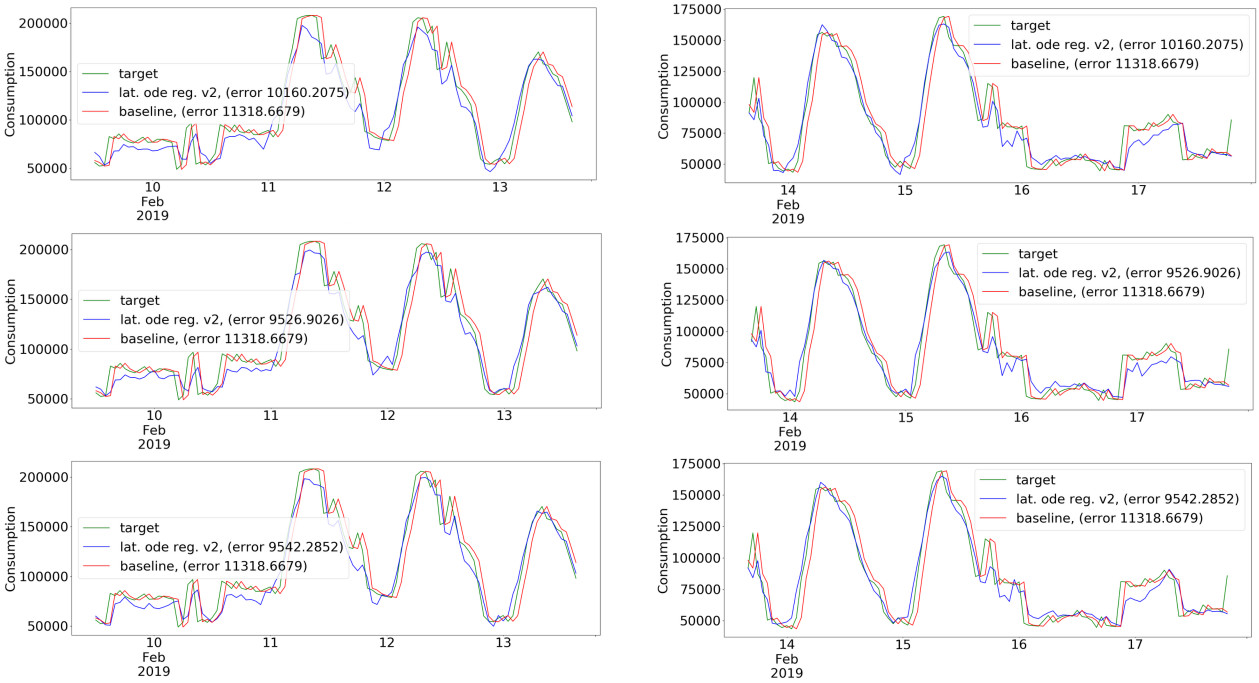
E2: Sampled Latent ODE predictions on the test set. Top ”small”. Bottom ”medium”. No plot exist for ”large” due to numerical underflow during the experiment.

ODE Regressor



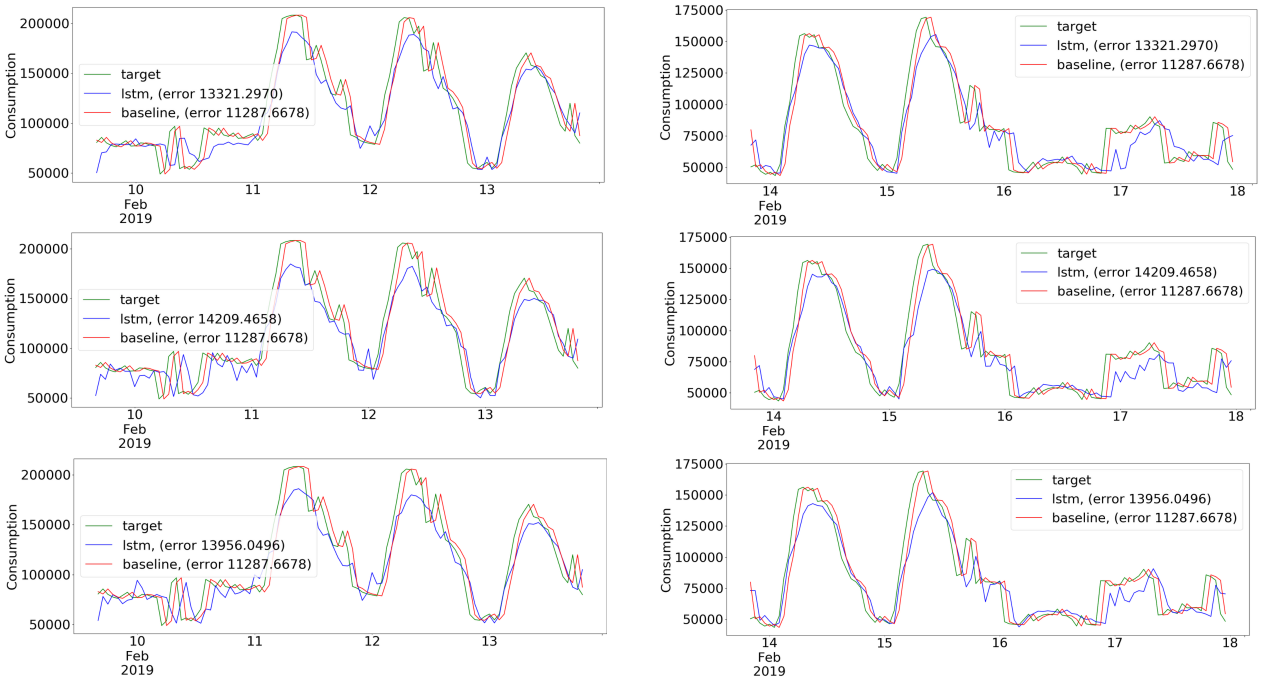
E2: Sampled ODE Regressor predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”.

Latent ODE Regressor v2



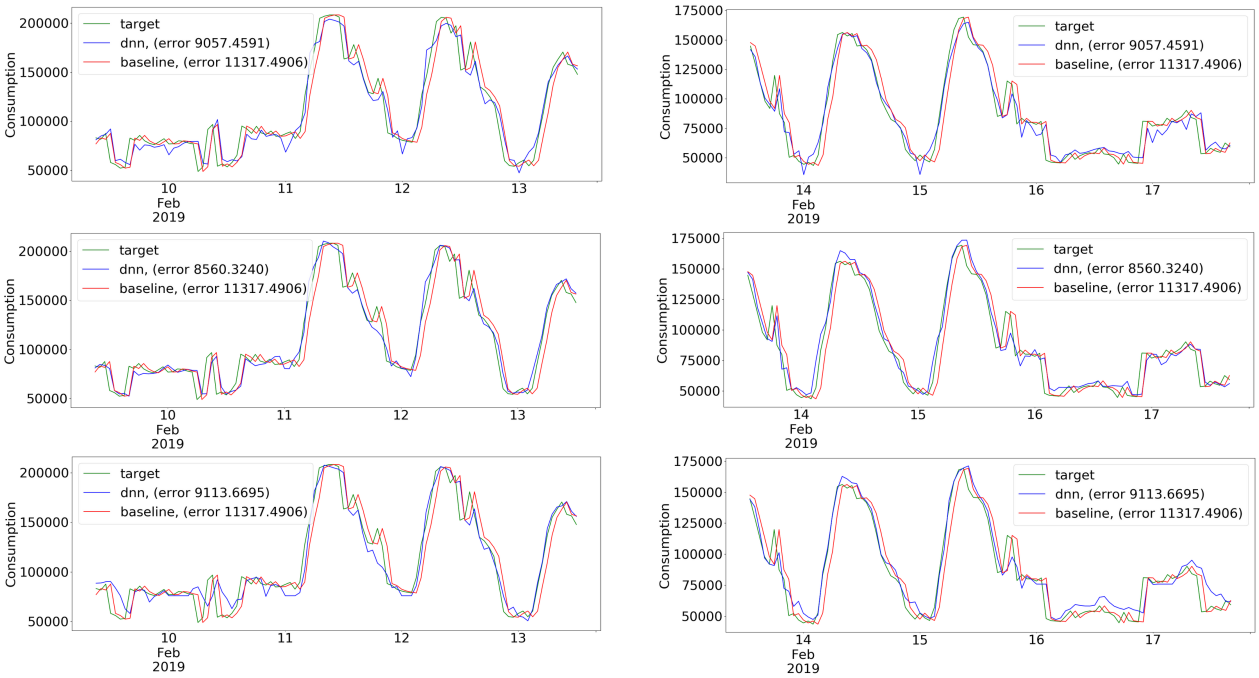
E2: Sampled Latent ODE Regressor v2 predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”.

LSTM



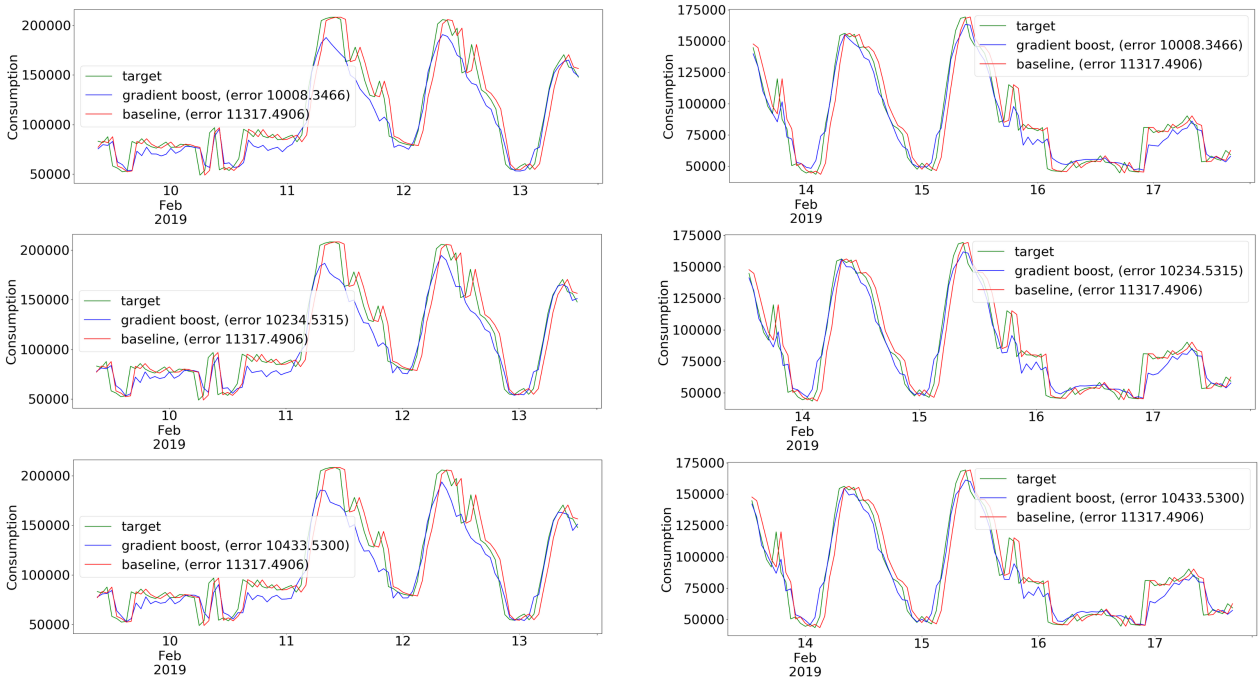
E2: Sampled LSTM predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”.

DNN



E2: Sampled DNN predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”.

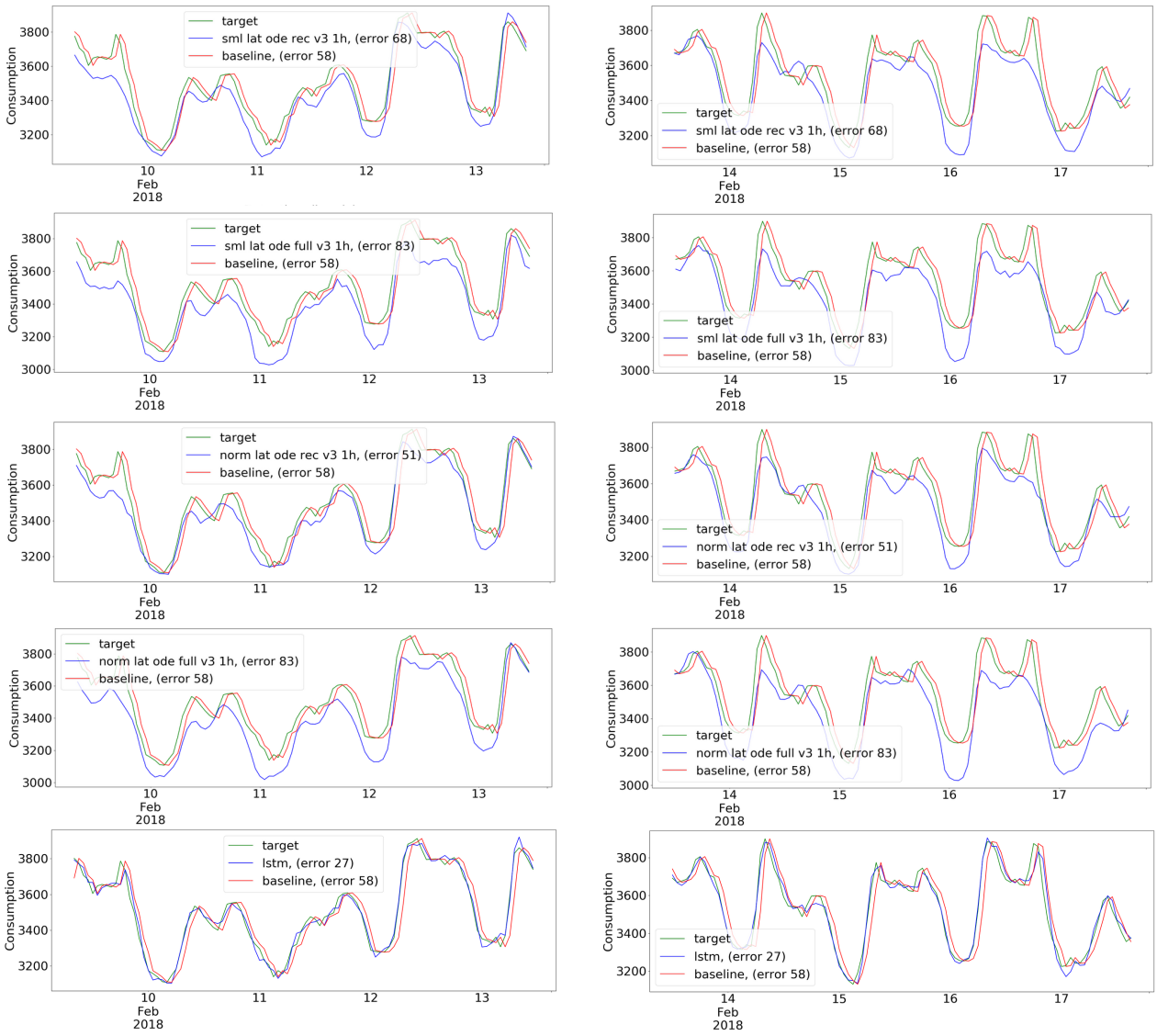
Gradient Boosting



E2: Sampled Gradient Boosting predictions on the test set. Top ”small”. Middle ”medium”. Bottom ”large”.

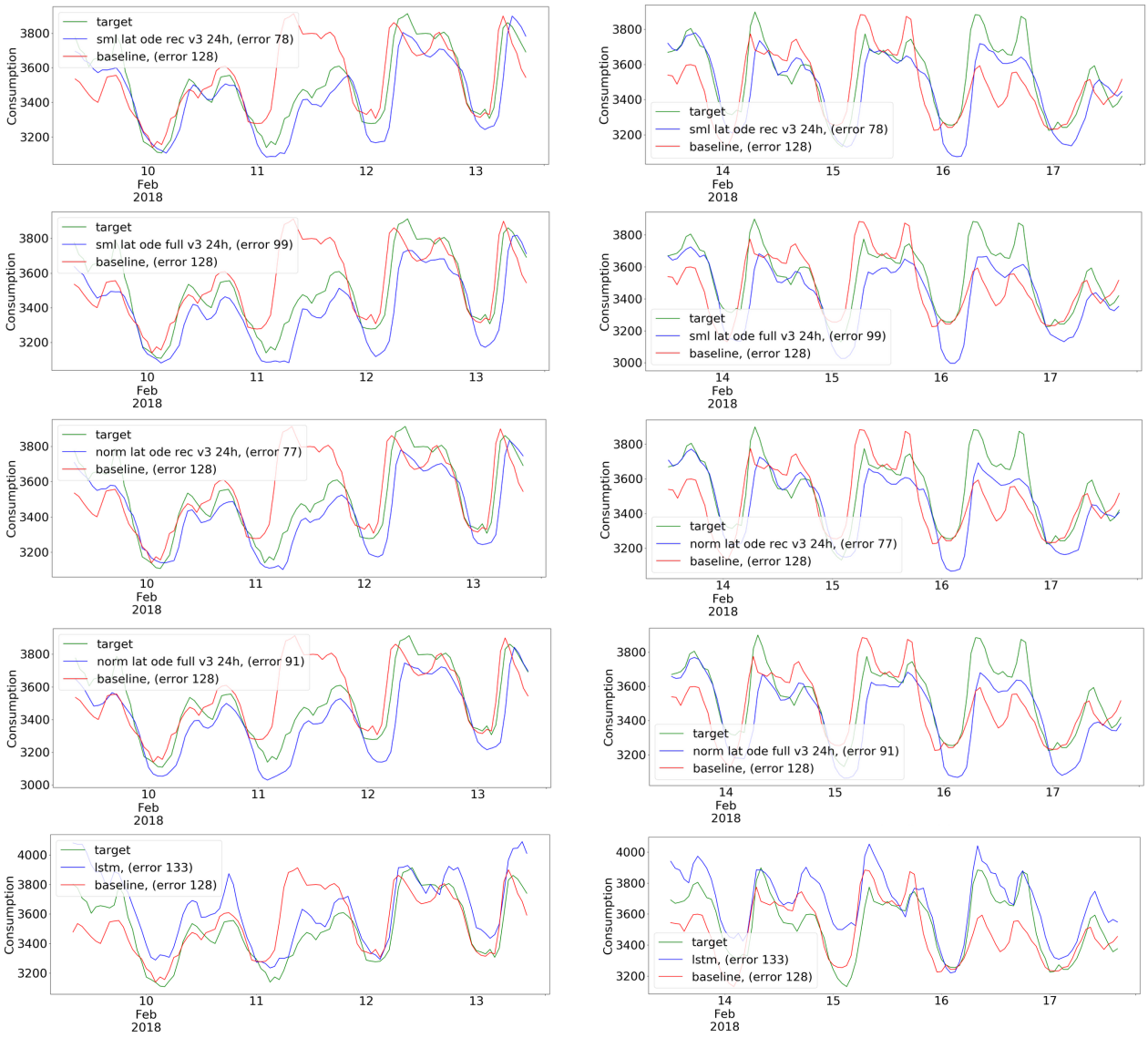
Experiment 3

1 Hour Ahead



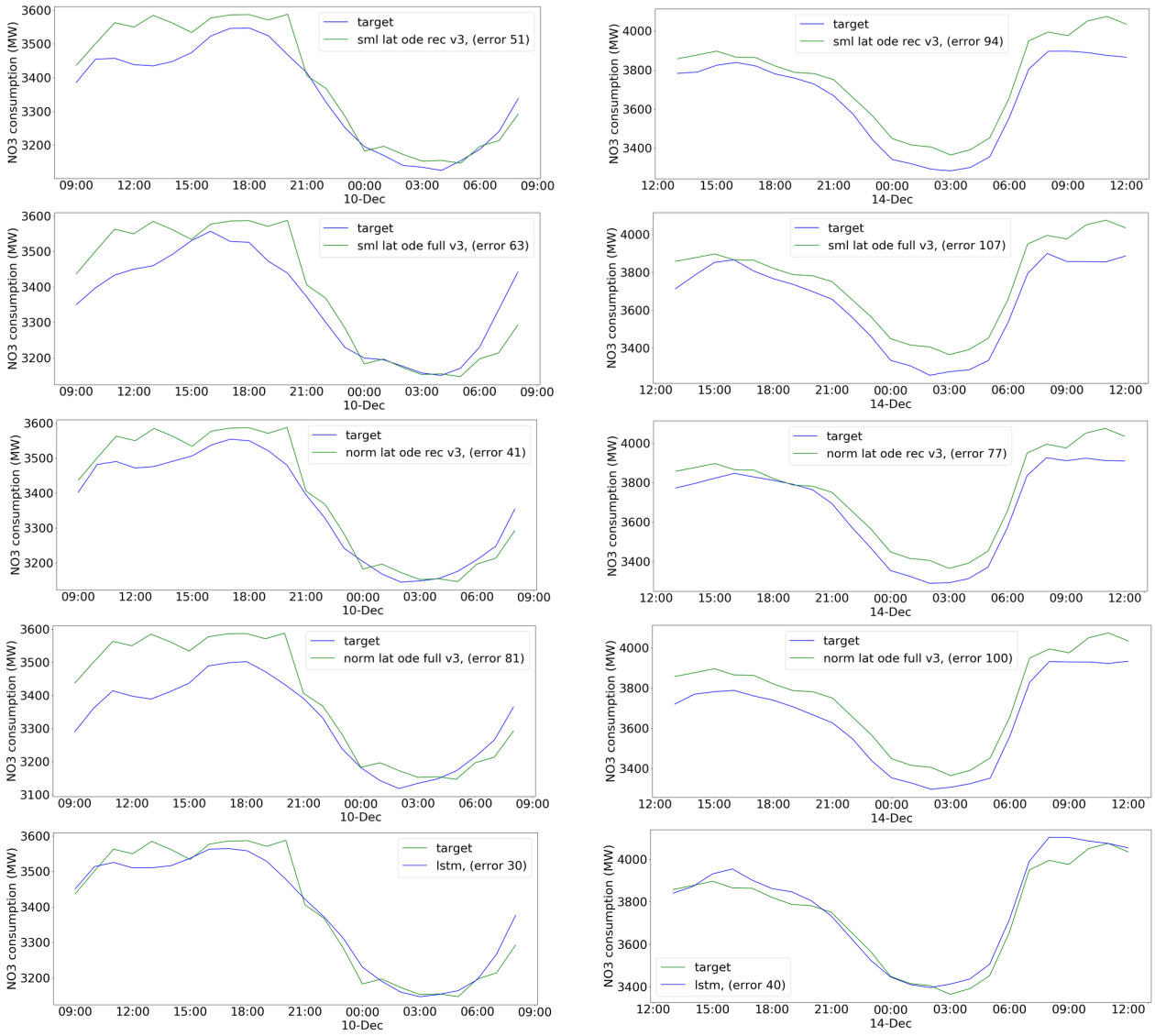
E3: Predicted one hour ahead trajectories from all models.

24 hours ahead

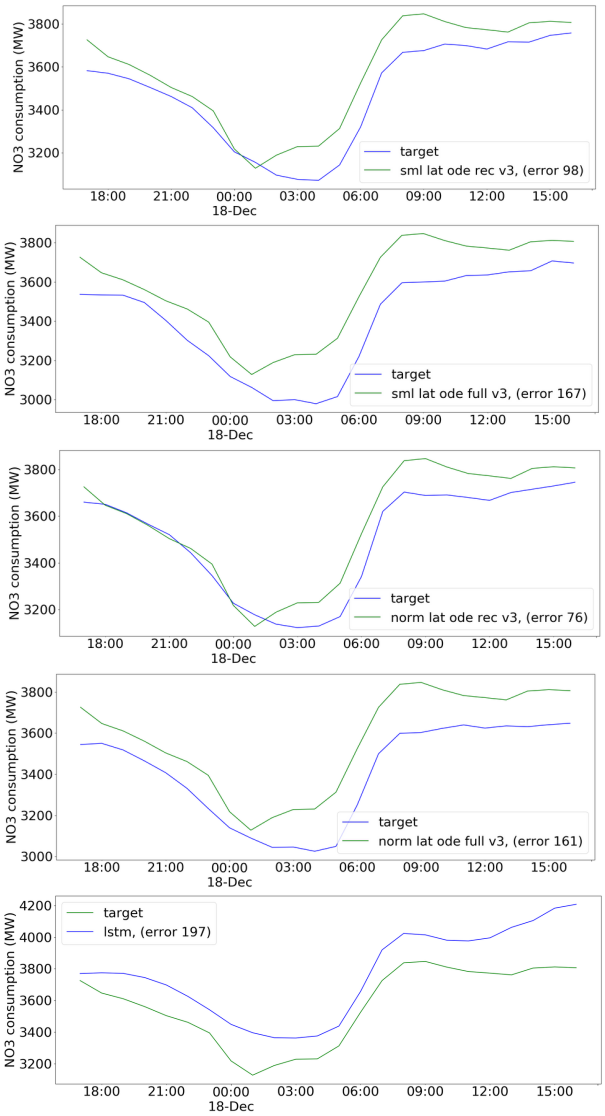


E3: Predicted 24 hour ahead trajectories from all models.

Next 24 hours



E3: Predicted next 24 hour trajectories from different models



E3: Predicted next 24 hour trajectories from different models