Morgan Feet Heggland
Patrik Kjærran

An exploration of sequence models using multi-task learning for multiphase flow rate estimation in oil and gas wells

Masteroppgave i Industriell Økonomi og Teknologiledelse
Veileder: Henrik Andersson
Medveileder: Bjarne Grimstad, Kristoffer Nesland
Juni 2022

Morgan Feet Heggland
Patrik Kjærran

An exploration of sequence models using multi-task learning for multiphase flow rate estimation in oil and gas wells

Masteroppgave i Industriell Økonomi og Teknologiledelse
Veileder: Henrik Andersson
Medveileder: Bjarne Grimstad, Kristoffer Nesland
Juni 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for økonomi
Institutt for industriell økonomi og teknologiledelse

Preface

The following thesis represents our master's thesis written during the final semester of our master's program in Industrial Economics and Technology Management at NTNU for the course *TIØ4905 - Managerial Economics and Operations Research, Master's Thesis.*

The thesis was written in collaboration with Solution Seeker AS, who generously provided productive and reflected supervision and counseling, a (very clean and extensive) dataset without which the results of this thesis would not be possible to obtain, and numerous exciting, challenging, and elaborate discussions ever sharpening our minds in tackling the problems at hand. Furthermore, they generously allowed us to visit their Brazilian office branch in Rio de Janeiro, where we could enjoy significant work progress on our thesis and much-needed relaxing sunny days on Ipanema Beach. Specifically, we would like to give our sincerest thanks to Bjarne Grimstad, Kristoffer Nesland, Vidar Gunnerud, and Knut Viljar Hilstad from Solution Seeker for their contributions to this thesis, for their generosity, and for their hospitality in accommodating us in Brazil.

Furthermore, the thesis was written under the supervision of Professor Henrik Andersson of the Department of Industrial Economics and Technology Management at NTNU. He has provided invaluable guidance in the formation of the thesis, aiding us in setting a course through the seemingly endless landscape of opportunities for exploration within machine learning and oil and gas production. We want to warmly thank him for his efforts in counseling us in this thesis.

Finally, we would like to emphasize that this thesis extends upon our work in Heggland and Kjærran [2021], our project report written in the course TIØ4500 during the fall semester of 2021. Consequently, several of the chapters of this thesis contains content inspired by or overlapping with the project report where this content is relevant for both works. However, all overlapping sections are revised, and new content is added where applicable to sufficiently inform the reader on the new topics explored in this thesis.

Trondheim, June 11, 2022
Morgan Feet Heggland & Patrik Kjærran

Abstract

Estimating the quantity of multiphase flow streaming through oil and gas wells is crucial to inform operational activities during offshore oil and gas production, such as production optimization. As a result, virtual flow meters (VFMs) have been developed because physically measuring the flow rates is often costly. However, while the underlying production system modeled is highly dynamic, most VFMs are steady-state VFMs that consider the instantaneous state of the system. Meanwhile, dynamic VFMs that account for past measurements may better model the underlying system but with increased model complexity.

This thesis conducts an empirical study of dynamic VFMs using state-of-the-art sequence model architectures and multi-task learning to examine the effects of considering past measurements when estimating flow rates. We present a methodology at the cutting edge of machine learning to leverage the information in the recent history of each well while simultaneously learning across wells. This allows the proposed models to model the temporal relationship between widely available sensor measurements and the resulting flow rates. The models are compared to a state-of-the-art steady-state baseline to determine their proficiency.

We test the presented methodology on field data from ten wells. The results indicate that sequence models demonstrate a more robust and flexible predictive behavior than their steady-state counterparts. However, compared with the state-of-the-art baseline only slightly improved overall predictive performance is obtained. Meanwhile, the positive model traits demonstrated by the sequence models are uncovered when examining the models in greater detail.

While the results give reason to believe that sequence models may improve the current state-of-the-art, they represent empirical indications based on a case study. Hence, we recommend that future research investigates different problems and model variations within the paradigm to strengthen further or weaken indications of its promise.

Sammendrag

Å estimere mengden flerfasestrøm som strømmer gjennom olje- og gassbrønner er avgjørende for å kunne ta informerte beslutninger angående operasjonelle aktiviteter relatert til offshore olje- og gassproduksjon, som for eksempel produksjonsoptimering. Som følge av dette har Virtual Flow Meters (VFM) som estimerer flytraten blitt utviklet, da det er kostbart å fysisk måle flerfasestrømmen. Mens det underliggende produksjonssystemet som modelleres er svært dynamisk, er de fleste VFM-er steady-state VFM-er som vurderer systemets øyeblikkelige tilstand. I mellomtiden kan dynamiske VFM-er som tar hensyn til tidligere målinger bedre modellere det underliggende systemet, men med økt modellkompleksitet.

Denne oppgaven gjennomfører en empirisk studie av dynamiske VFM-er ved bruk av toppmorderne sekvensmodellarkitekturer og multi-task learning for å undersøke effekten av å vurdere tidligere målinger når man estimerer strømningsratene. Vi presenterer en metodikk i forskningsfronten av maskinlæring for å utnytte informasjonen i historikken til hver brønn samtidig som vi lærer på tvers av brønner. Dette lar de foreslåtte modellene modellere det tidsmessige forholdet mellom mer tilgjengelige sensormålinger og de resulterende strømningshastighetene. Modellene sammenlignes med en avansert steady-state-modell for å vurdere ytelsen av de foreslåtte modellene.

Vi tester den presenterte metodikken på feltdata fra ti oljebrønner. Resultatene indikerer at sekvensmodeller demonstrerer en mer robust og fleksibel prediktiv atferd enn deres steady-state motstykker. Sammenlignet med den avanserte steady-state modellen oppnås imidlertid bare en svakt forbedret overordnet prediktiv ytelse. I mellomtiden blir de positive modelltrekkene demonstrert av sekvensmodellene tydeliggjort når man undersøker modellene mer i detalj.

Mens resultatene gir grunn til å tro at sekvensmodeller kan forbedre de nåværende beste modellene, er de empiriske indikasjoner basert på en casestudie. Derfor anbefaler vi at fremtidig forskning undersøker ulike problemer og modellvariasjoner innenfor paradigmet for å ytterligere styrke eller svekke indikasjoner på hvor lovende det er.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The quantity of oil and gas produced by each well during production is vital information. Such measurements are leveraged in critical operational activities and decisions, such as production optimization (maximizing output), diagnosing, shutting down, and drilling new wells, reservoir management, and forecasting oil field performance [Corneliussen et al., 2005, Bikmukhametov and Jäschke, 2020]. However, obtaining information about the outputs of wells is challenging.

The content extracted from a reservoir through a well is a flow of multiple phases: oil, gas, and water. The flow produced from a single well is commonly measured periodically using well tests or in close to real-time using multiphase flow meters (MPFM), both requiring expensive physical installations. Consequently, obtaining multiphase flow rate measurements is costly and time-consuming, resulting in a limited number of flow rate observations over the production time span of most wells [Zangl et al., 2014].

Continuous depletion of reservoirs and increasingly economically marginal fields demand more efficient allocation of resources and focus on rational operation [AL-Qutami et al., 2018]. Thus, more cost-effective approaches to estimating flow rates are sought after. Considerable research efforts have been made on this topic, resulting in the rise of virtual flow meters (VFM). VFMs are mathematical models that can cost-effectively estimate flow rates in real-time using different sensor measurements of well parameters, such as temperatures and pressures [Bikmukhametov and Jäschke, 2020]. While several successful applications exist, the need for improved performance in VFMs is still evident, motivating the topic of this thesis.

The underlying production system must be modeled to estimate the output flow of oil and gas from a well. Meanwhile, the production system is highly dynamic and complex, and different approaches to model such systems have arisen. This has led to the division between data-driven and physics-based VFMs and between dynamic and steady-state VFMs. Works relating to the different types of VFMs are outlined in Bikmukhametov and Jäschke [2020].

Physics-based VFMs mechanistically model the multiphase flow and the surrounding well system to estimate the flow rates. Meanwhile, the recent developments within computer science paradigms such as artificial intelligence (AI) and machine learning (ML) have given rise to data-driven approaches that model relationships between observable quantities in the well systems and the output flow rates. As the installation of sensors has become commonplace in hydrocarbon production systems, vast amounts of data measuring the operational state of oil wells (e.g. pressures, temperatures, valve openings) are available, in contrast to the sparse flow rate measurements. Increased computational resources and data access has enabled several successful VFMs using machine learning techniques.

Because modeling the dynamics of the production system over time is very complex, a divide between dynamic and steady-state VFMs has arisen. These differ in whether the model estimates flow rates based on the development of the system over time (dynamic) or only the instantaneous state (steady-state). Due to the increased complexity implied by a dynamic model, most applied VFMs are steady-state models. However, this comes with a performance cost, as historical information about the well's state is not considered when estimating flow rates. One approach commonly used in data-driven dynamic VFMs is sequential modeling. Sequence models are mathematical models that operate on sequences, either by inputting or outputting sequences, or both. Within AI, sequential modeling is widely studied with numerous impressive applications. This thesis explores dynamic data-driven VFMs, applying sequential modeling in combination with state-of-the-art ML methods to flow rate estimation.

This thesis is written in collaboration with Solution Seeker, a spin-off company from Engineering Cybernetics at NTNU, which is a leading provider of managed AI services to the petroleum industry. One of their key products is a steady-state VFM ML system that estimates flow rates through well sensor data. Their state-of-the-art model is presented in Sandnes et al. [2021], detailing a system capable of accurately predicting flow rates in oil and gas wells. The model utilizes the ML paradigm multi-task learning (MTL) to learn across data from multiple wells. This approach somewhat alleviates the issue of infrequent measurements of multiphase flow rates. However, such measurements are still scarce, while sensor measurements of e.g. temperature and pressure, are abundant. Additionally, in Grimstad et al. [2021] Solution Seeker expresses a need for future research to overcome the issues of low data volume in flow modeling. Thus, the question of whether this data can be leveraged to improve the flow rate estimates arises.

One attempt at exploiting the vast sensor data was made in Løvland [2021], a master's thesis written under the supervision of Solution Seeker. The thesis explores semi-supervised learning (SSL) methods. SSL is another machine learning paradigm combining supervised and unsupervised learning methods. These methods can exploit the widely available unlabeled data to create a different representation of the well system, aiming to improve flow rate estimates. However, the results of the thesis indicated negative effects of using SSL to estimate flow rates for a single well. In our project report, we built upon these results and Solution Seeker's system [Heggland and Kjærran, 2021]. We investigated whether state-of-the-art SSL methods can create better representations of the production system for flow rate estimation when used in MTL-based VFM. The results showed that learning state representations on unlabeled data considerably improved MTL-based flow rate estimates. Meanwhile, this thesis attempts to use additional sensor measurements in a different approach: dynamic VFMs. By considering the past history of the wells when estimating flow rates, a considerable amount of unlabeled data is leveraged. Thus, this thesis expands on the project report by replacing the SSL approach to producing informative well state representations with sequential models. In addition, it expands from considering instantaneous well states to sequences considering past states as well.

The overall objective of this thesis is to examine the promise of using sequence models in MTL-based flow rate estimation. The results from the project report indicated a benefit from using unlabeled data in a steady-state MTL model. This, together with the promise of modeling the production system over time, motivates the purpose of this thesis: to investigate whether dynamic MTL-based VFMs may perform better than steady-state variants. By using sequence models to estimate flow rates, more informative intermediate representations may be learned as historical data is also incorporated. The intention is to examine whether the produced representations may improve flow rate estimates when used in MTL-based flow rate estimation models, such as Solution Seeker's model. Hence, the findings of the thesis may support Solution Seeker in their research aiming to improve their flow rate estimation system.

The problem explored in this thesis is to use sequences of operational field data provided by Solution Seeker in MTL-based sequence models. The sequences represent the operational history leading up to the flow rate estimate, including unlabeled data. The sequential models are trained to extract the most relevant information in the production history to produce accurate rate estimates. These abstract representations of the well state are then used to estimate flow rates. Then we investigate whether estimates improve compared with steady-state models. This is summarized in the main research question explored in this thesis: *How does leveraging sequences of historical well state data affect the flow rate estimates of data-driven MTL-based virtual flow meters?* Thus, we emphasize that this thesis does not seek to maximize production output across wells. Concerning the problem of production optimization, the thesis aims to obtain more precise measurements of flow rates which represent information used in the said optimization problem.

The thesis represents an extension of Heggland and Kjærran [2021] by expanding the approach presented in the project report from steady-state to dynamic through sequential modeling. Furthermore, it builds on the results of Sandnes et al. [2021] and Løvland [2021] as it combines methods at the frontier of research within sequence models and multi-task learning. Hence, the thesis represents the following contributions. First, the thesis contributes to Solution Seeker's research by exploring dynamic VFMs. Second, it complements the existing VFM literature by simultaneously learning through unlabeled data and across data from multiple wells and accounting for historical data when producing estimates. Additionally, the thesis extends upon existing literature by examining the performance of TCNs and GRUs, two promising sequence model architectures that are not as well-studied as LSTMs. Finally, it also addresses the need for research attempting to overcome the issue of limited availability of data as indicated in Grimstad et al. [2021].

The thesis is structured as follows: we start by describing relevant background information and literature in Chapter 2, providing relevant context regarding the topic of the thesis. Then Chapter 3 presents fundamental machine learning theory relevant to understanding the approaches applied in exploring the presented research question. The problem at hand is formally described in Chapter 4 before we detail our methodology used to answer the research question in Chapter 5. Chapter 6 then describes the provided dataset on which the methodology is applied. The results obtained through the method are presented in Chapter 7 and discussed in Chapter 8. Finally, we provide concluding remarks and indicate needs for further research in Chapter 9.

Chapter 2

Background

This chapter provides relevant context regarding the problem of estimating flow rates in offshore oil and gas production. This motivates the subsequent problem statement and our approaches to solving it. We start by describing the components of the oil and gas production system in Section 2.1 before discussing production optimization in Section 2.2. Different ways of measuring the contents of the multiphase flow extracted from reservoirs are then detailed in Section 2.3. Next, Section 2.4 discusses flow modeling, describing different approaches towards estimating the flow rates. The partner company Solution Seeker and their relation to this thesis are then presented in Section 2.5. Finally, Section 2.6 highlights related literature to illustrate how the work of this thesis is positioned in and complements the current state of the field. Note that this chapter is a revised version of Chapter 2 of Heggland and Kjærran [2021], our project report.

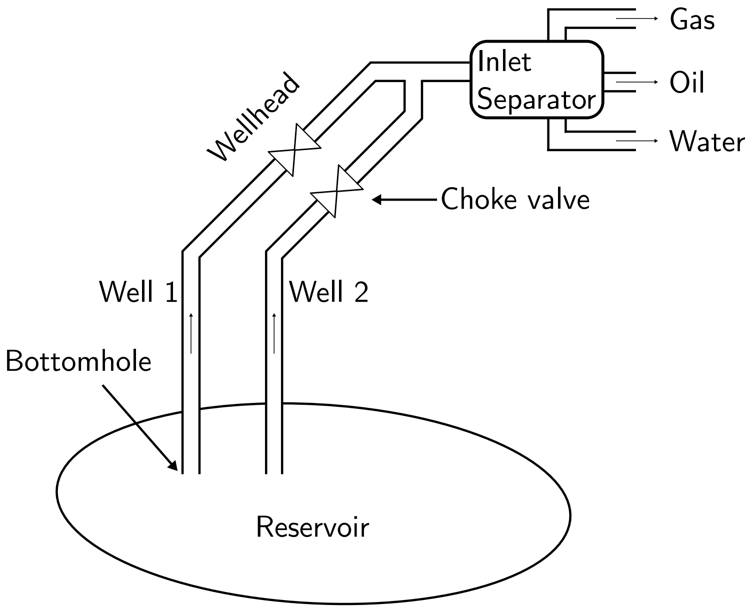


Figure 2.1: Illustration of an offshore oil and gas production system. Contents of the reservoir are extracted in the bottomhole of the two wells in the form of a multiphase flow. This flow can be regulated by operating the choke valve in the wellhead. Then, the stream is routed to an inlet separator that separates the contents into gas, oil, and water.

2.1 The offshore oil and gas production system

Figure 2.1 illustrates the main components of an offshore oil and gas production system. During production, hydrocarbons are extracted from *reservoirs* which are pockets of natural gas and oil located beneath the ocean floor. This is accomplished through the installment of *oil wells* which function as pressurized pipelines flowing the contents from the reservoir through the production system. A reservoir is commonly connected to several oil wells, which together may be referred to as an *asset*. The point of the well connecting to the reservoir is called the *bottomhole*. Here, a multiphase flow including hydrocarbons are extracted from the reservoir. This stream is a mixture of oil, gas, water, and gravel and can be regulated by turning a *choke valve* located on the *wellhead* at the top of the well. We denote the opening percentage of a choke valve u. Most wells have sensors installed measuring pressure and temperature at different points along the well system. The most common measurement locations are at the bottomhole and the wellhead, both upstream and downstream the choke valve [Bikmukhametov and Jäschke, 2020]. We denote the pressure and temperature upstream the choke p_1 and T_1 respectively, and the pressure and temperature downstream the choke p_2 and T_2 respectively. See Figure 2.2 for an illustration. Sensors measuring these quantities are typically sampled several times per minute, with frequencies such as 0.5 Hz being commonplace, according to Solution Seeker. From the wellhead, the flow is commonly merged with other well flows and routed to an inlet *separator* in a processing facility, as illustrated in Figure 2.3. Development of such production systems are standardized under ISO standard 13628-1, and a thorough specification can thus be found in International Organization for Standardization [2005].

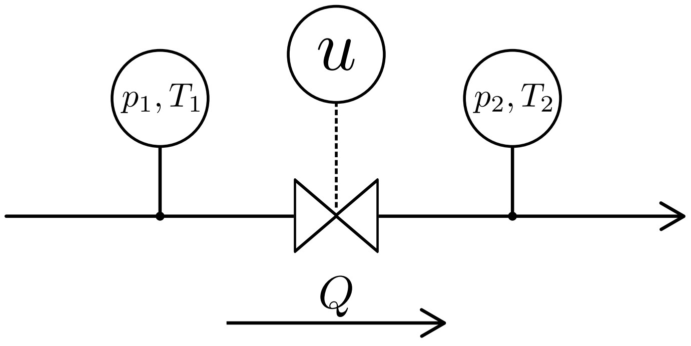


Figure 2.2: Illustration of a choke valve with relevant sensor measurements. p_1 and T_1 denote the pressure and temperature measured upstream the choke valve while p_2 and T_2 are the corresponding measurements downstream the choke valve. u is the choke valve opening, and Q is the multiphase flow rate.

2.2 Production optimization

When operating an oil field, operators aim to maximize the extraction of hydrocarbons throughout the field's lifespan, taking technical, economic, and other strategic factors into account. Planning of operations is commonly performed in different planning horizons to achieve this. The horizons include long-term strategic planning over several years, shorter horizons where operations are planned given the long-term strategic plans and goals, and short-term horizons, planning day-to-day production. See Wang [2003] for a review of different optimization problems and techniques applied in petroleum production.

In a short-term horizon, petroleum production engineers aim to maximize the total flow of oil and gas from the asset, given the known current state of the production system. This must be done while respecting any constraints on the production, such as capacity constraints and pipeline network constraints. During operation, the production engineers must thus decide how to regulate the multiphase flow from each well to maximize production output, given constraints and well flow rates. This short-term production optimization problem is called Real Time Production Optimization (RTPO). The operation of oil fields is subject to uncertainties causing unexpected changes in the system, requiring optimization models to be solved quickly, hence the term Real Time. A review of essential literature regarding the RTPO problem is given in Bieker et al. [2007]. One key issue in variants of the RTPO problem is that a measure or estimate of the multiphase flow rates from each well must be acquired to maximize production. See for instance Gunnerud and Foss [2010], where flow rates of different phases from wells are estimated using a well model and used in maximizing production. Acquiring precise estimates of the multiphase flow rates is challenging (see below), and this issue is the main focus of this thesis.

2.3 Measuring multiphase flow rates

The previous section illustrated how knowledge of the amount of oil and gas flowing through each well at any moment is vital to optimize production. However, physically measuring these quantities is challenging and costly [Hansen et al., 2019]. The quantity of oil flowing through a well at a given moment is commonly referred to as the *oil flow rate* denoted q_O, and likewise for the *gas flow rate* denoted q_G and *water flow rate* denoted q_W. These are typically measured in volumetric flow per day or hour at standard conditions [Society of Petroleum Engineers, 1994]. Together, these three measures define the *total flow rate* Q where $Q = q_O + q_G + q_W$. Solids like gravel are commonly neglected. Flow rates are also often represented as *fractions* of total flow. E.g. the *oil fraction* ϕ_O is defined as $\frac{q_O}{Q}$. Likewise holds for the *gas fraction* ϕ_G and *water fraction* ϕ_W. In addition to these rates, sometimes operators inject gas in the bottomhole to “reduce” the density of the oil, allowing it to flow more easily. We denote the injected gas lift flow rate q^{GL}.

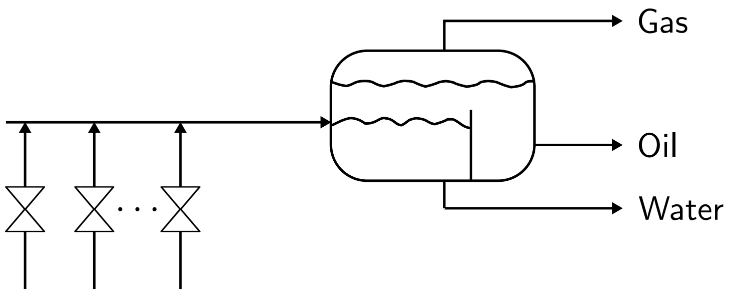


Figure 2.3: Illustration of a set of wells routed to the same inlet separator.

The traditional method for measuring a multiphase flow is by using separator tanks. From the wellheads, the flows from different wells are merged and routed to an inlet separator at a processing facility. A separator is a container that separates the multiphase content by gravity. As flow components have different densities the gas rises to the top of the tank, the oil separates in the middle, and the water sinks to the bottom over time. The output components can then be measured as single-phase flows. Single phase flows are much easier to measure and measurements reaching as low as 1% uncertainty can be obtained [Corneliussen et al., 2005]. See Figure 2.3 for an illustration of the separation process.

While this approach allows for multiphase flow measurement, it is impossible to measure the flow rates for specific wells if the input flow consists of streams merged from multiple wells. This issue can be handled in a few different ways: by use of *deduction well testing*, by utilizing a *test separator* or by installing *multiphase flow meters*.

Deduction well testing is the practice of measuring the flow rates of a specific well by first measuring the total flow rates as discussed above, then shutting in the given well (close the choke valve), and measuring the total flow rates after the system has stabilized. In this case, stability typically implies reaching downstream pressures in the other wells similar to before the shut-in. The difference in flow rates before and after the shut-in then approximates the output of the shut-in well. As this approach uses separators, it can yield relatively good estimates reaching uncertainties of down to 1% [Idsø et al., 2014]. However, it implies shutting in a well for the duration of the deduction test. The test can take up to several hours [Bikmukhametov and Jäschke, 2020], causing a reduction in production output and consequently lost revenues.

The multiphase flow from a specific well can be routed to a dedicated test separator to measure the flow rates. This method has the advantage of estimating flow rates with high accuracy using single-phase flow meters. However, while this approach does not require shutting in wells, it is still costly. For one, installing a test separator and the flowlines required is capital-intensive [Corneliussen et al., 2005]. Additionally, as mentioned, the separation process is time-consuming as it requires conditions to stabilize. Especially installations with longer flowlines take several hours to reach stable conditions, which also affects production output [Falcone et al., 2002]. Consequently, production engineers are limited in how often they are to schedule well tests for each well.

Multiphase flow meters (MPFMs) are measuring instruments that can be installed along the flowline of a well to measure multiphase flow rates. There exists a range of different MPFMs using different technologies, such as electromagnetic measurement principles, radioactive densitometry or spectroscopy, and ultrasonic measurement technology, to name a few. These instruments may yield measurements of relatively low uncertainty, however usually not as low as in well tests [Corneliussen et al., 2005]. A key advantage of an MPFM is the drastically increased availability of flow rate measurements, as measurements typically take minutes instead of hours. However, they require regular calibration to remain precise and are very costly to install. As a consequence, such multiphase flow meters are not installed on a large number of wells [Zangl et al., 2014]. A comprehensive overview of MPFMs can be found in Corneliussen et al. [2005].

2.4 Flow modeling

As illustrated in the previous section, physically measuring the multiphase flow rates can be costly and challenging. Consequently, considerable efforts have been made to model oil wells to estimate the flow rates. Such models may serve as a more accessible and less costly alternative to obtaining flow rate information with reasonable uncertainty. The problem of flow rate modeling can be described as follows: *Identify a model that produces an estimate $\hat{Q}$ of the measured flow rate Q at the given point in time for the given well, given measurements relating to the operational state of an oil and gas production system.* Such models are commonly referred to as *virtual flow meters* and can be described as mathematical models that estimate flow rates using different approaches and information. Two distinct modeling paradigms have emerged within the field of flow modeling. The first is the paradigm of physics-based VFMs leveraging physical principles to model an oil well system's behavior. The second paradigm uses the amounts of available data in data-driven approaches. See Figure 2.4 for an illustration. An extensive review of the paradigms is given in Bikmukhametov and Jäschke [2020], where paradigms are referred to as first principles VFMs and data-driven VFMs. A brief introduction is given below.

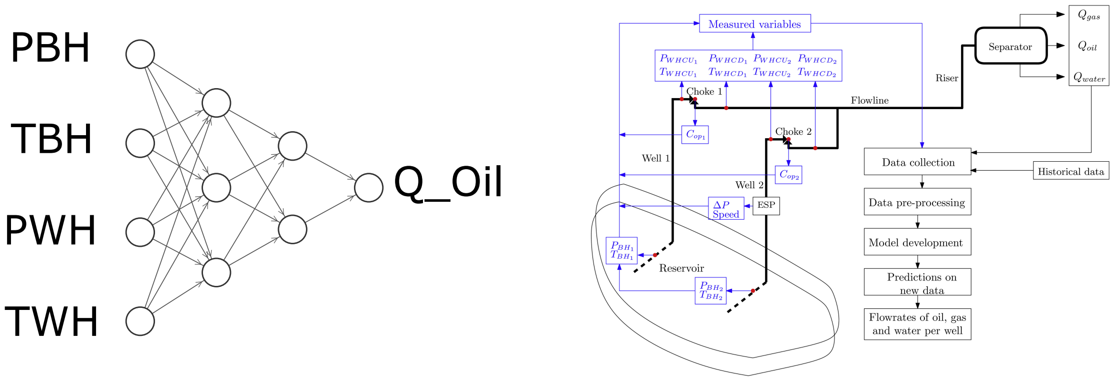


Figure 2.4: Illustration of the difference between data-driven and physics-based VFMs. **Left**: Data-driven VFMs are computer algorithms that learn relationships directly between the input measurements and the output flow rates. **Right**: Physics-based VFMs attempt to mechanistically model the production system with related physical parameters to estimate flow using physical properties and relationships. Right source: Bikmukhametov and Jäschke [2020]

As most industries have been transformed by the introduction of information technology and digitization, the oil and gas industry is no exception. Most wells are equipped with instruments that continuously measure pressure and temperature, among other quantities. Over time, the operations of oil and gas wells have generated large amounts of data from all recorded measurements. In combination with increased access to computation power, this has given rise to new digital, data-driven approaches for estimating flow rates. Data-driven models estimate flow rates by learning patterns in the available operational data instead of directly modeling wells [Bikmukhametov and Jäschke, 2020]. Such VFMs are commonly based on machine learning methods which have proven to be highly proficient at a wide range of different tasks. Fundamental machine learning theory is explained in Chapter 3. The applied methods range from simple models such linear regression [Zangl et al., 2014] to advanced models such as as feedforward neural networks [Al-Qutami et al., 2017], recurrent neural networks [Andrianov, 2018] or a combination of different neural networks [AL-Qutami et al., 2018]. A more detailed review of relevant literature within this field is provided in Section 2.6.

Physics-based flow models model the oil well mathematically to estimate the flow rates. This typically involves mechanistic modeling of the multiphase flow rate in the production system through the use of physical parameters of the well in addition to sensor measurements [Bikmukhametov and Jäschke, 2020]. The error between the modeled or predicted and actual flow is minimized by using algorithms that tune the model parameters. These VFMs commonly rely on physical relations within fluid dynamics, thermodynamics, and mechanics in modeling the production system. Despite data-driven VFMs increasing in popularity, physics-based VFMs remain the most prominent in the industry. Bikmukhametov and Jäschke [2020] present a thorough review of such VFMs.

VFMs can be considered either dynamic or steady-state, based on whether they consider the development of the system over time or only an instantaneous state when estimating

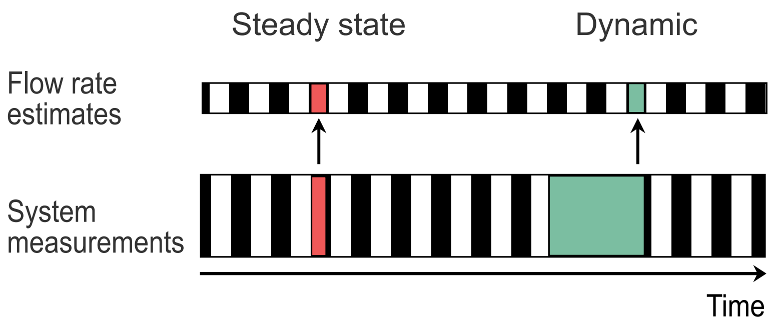


Figure 2.5: Illustration of the difference between dynamic and steady-state VFMs. Given a discretization of time into steps, steady-state VFMs use the system measurements for the current time step to produce the flow rate estimate for the current time step (illustrated in red). Meanwhile, dynamic VFMs also consider measurements from past time steps to estimate the current flow rate (shown in green).

flow rates. See Figure 2.5 for an illustration. A dynamic formulation enables the model to capture dependencies over time, yielding more informed flow rate estimates. However, due to the increased complexity of dynamic models, most commercial VFMs use a steady-state formulation.

Physics-based VFMs are often based on dynamic conservation relationships. However, the optimizers solve the problem in a steady-state manner and use solutions from the previous time steps as a starting point for estimates for the next time step [Bikmukhametov and Jäschke, 2020]. Both Kalman filter approaches and dynamic optimization can be used to make dynamic physics-based VFMs. However, computational complexity and expertise requirements have hindered the prevalence of dynamic physics-based VFMs in commercial applications [Bikmukhametov and Jäschke, 2020].

Dynamic data-driven VFMs are machine learning models that use measurements in the past to estimate the flow rate in the current time step. This differs from steady-state models, which only use measurements in the current time step to produce estimates for the current time step. Such models are often sequential models, i.e., models that consider the relationship between a sequence of observations over time and the quantity to be estimated. An example of this is recurrent neural networks. Sequence models in machine learning are discussed in Chapter 3. Dynamic data-driven VFMs are the primary focus of this thesis. Consequently, related works are discussed in Section 2.6.

It is also possible to combine principles from both VFM paradigms by, for instance, informing the machine learning process using physical properties. This practice is called *hybrid modeling*. An example can be found in Hotvedt et al. [2020] where a hybrid data-driven mechanistic VFM is proposed. Here, the choke valve is mechanistically modeled using first-principle equations where one of the parameters, the choke valve coefficient, is determined by using a fully connected feedforward neural network (described in Chapter 3).

2.5 Solution Seeker

The Norwegian company Solution Seeker – a spin-off from Engineering Cybernetics at NTNU – is a leading provider of managed AI services to the petroleum industry. Its services are used by several international oil companies to monitor thousands of oil and gas wells and to give operational support to engineers. The underlying technology is a machine learning system that extracts and presents valuable information about the performance of the wells from sensor data. This may aid engineers in work-intensive tasks, such as detecting anomalies in the behavior of the wells based on patterns in the data. Accuracy, low maintenance requirements, and quick adaptation to changing process conditions are key features of the system.

Recent research from Solution Seeker has demonstrated that neural network-based models can accurately predict flow rates in oil and gas wells. Multi-task learning, a learning paradigm that enables learning across data from multiple wells (described in Section 3.5.4), has been instrumental in achieving the robustness and accuracy required of an industrial application (see Section 2.6 for further details).

The company is also exploring other learning paradigms to battle the lacking availability of supervised data, i.e., multiphase flow measurements. Solution Seeker vocalizes a need for research on this topic in Grimstad et al. [2021]. Semi-supervised learning is a promising paradigm the company is currently examining to address this issue. SSL combines unsupervised and supervised learning techniques to solve a learning task (see Section 3.5.3). The promise of semi-supervised learning is that unlabeled production data, which vastly outnumbers labeled data on many tasks, may be utilized to improve models. While the paradigm may not improve upon supervised learning on all tasks, it is especially interesting to explore whether semi-supervised learning can improve performance on tasks with few labeled data points and many unlabeled data points.

By extending upon the the mentioned works relating to Solution Seeker, we may give further insights into which direction Solution Seeker's research on flow modeling should be directed. The thesis extends upon the two mentioned research areas through the introduction of sequence models in place of semi-supervised learning. Hence, it also represents an extension of our project report [Heggland and Kjærran, 2021]. We investigate the effects of accounting for historical data in a dynamic MTL-based VFM in a purely supervised context. Sequential models represent a different learning paradigm that addresses the issue of low data volume in VFMs by incorporating historic unlabeled production data in the predictions. The results may indicate whether such an approach can improve the performance of the Solution Seeker's current flow rate modeling system. In this manner, the thesis contributes to the advancement of Solution Seeker's machine learning system.

2.6 Related literature

This thesis examines the viability of estimating flow rates using sequences of operational history through machine learning models that operate on sequences. The sequence models learn representations of the operational state of oil wells based on the historical data points. These representations are then used as inputs in MTL-based VFMs to examine whether they improve flow rate estimates compared with steady-state VFMs. More specifically, the sequence models applied in this thesis are three variations of recurrent neural networks (RNN): a standard RNN, Long-short Term Memory (LSTM), and Gated Recurrent Unit (GRU), in addition to Temporal Convolutional Networks (TCN) (see Section 3.6). To illustrate how this thesis is positioned in the existing literature, we start by showcasing related literature by Solution Seeker before showcasing other related works within the problem domain of virtual flow metering. Finally, we highlight works relating to the applied methods within the machine learning literature across other domains.

Literature associated with Solution Seeker
A number of works associated with Solution Seeker are especially relevant for this thesis. First, Sandnes et al. [2021] present the state-of-the-art flow modeling architecture used in operations at Solution Seeker. To estimate oil and gas flow rates, Solution Seeker uses a homogeneous MTL approach (see Section 3.5.4). For a given time point, measurements from a given well are input to the neural network architecture along with well-specific trainable parameters. A flow rate estimate is produced and compared with the measured rate (from a separator test or MPFM) to calculate the error. Hence, this is a steady-state model. Second, Løvland [2021] explores the use of semi-supervised methods in flow modeling in detail. The author examines pre-training a variational autoencoder on unlabeled data for a single well (i.e., single-task, steady-state flow metering). Then, this autoencoder is used to encode and represent the state of the well for a set of labeled data before predicting flow rates for the same single well using a basic feedforward neural network. In applying this method, Løvland did not find that the pre-training improved the estimates. Based on this, Heggland and Kjærran [2021] examined combining the paradigms of SSL and MTL by training autoencoders to learn well state representations through unlabeled data from multiple wells and using this in combination with MTL-based steady-state flow models. The results showed that learning representations through unlabeled data improved flow rate estimates.

The need for further research towards remediating the issues of low data volume in VFMs is highlighted by Solution Seeker in Grimstad et al. [2021]. Here, a data-driven VFM using Bayesian neural networks is presented. The results show acceptable performance on field data. However, the authors argue that further research is required to tackle the lack of data concerning flow rate measurements. Their recommendations for future research include, among others, ”data-driven architectures that enable learning from more data, for instance, by sharing parameters between well models” and ”to focus on ways to overcome the challenges related to small data”. This thesis addresses these recommendations and examines cutting-edge machine learning models enabling learning both on a more significant amount of data and across wells through parameter sharing.

Existing flow modeling literature
As indicated earlier, the literature includes numerous works applying machine learning models to estimate flow rates. Some works are mentioned in Section 2.4, and Bikmukhametov and Jäschke [2020] give a detailed overview. In describing the literature, we start by providing a general overview of data-driven VFMs before discussing the most relevant works.

Before the prevalence of machine learning, attempts were made to model flow rates using linear regression. However, due to the highly complex and non-linear nature of the multiphase flow, such approaches were unable to obtain satisfactory performance [Mercante and Netto, 2022]. Consequently, as computational resources and data became increasingly available, research on data-driven VFMs focused more on approaches using artificial intelligence. Generally, studies differ in whether the model is steady-state or dynamic, which modeling approach is used, and what data the models are applied on. The data used is typically simulated or synthetic, from laboratory experiments or field data from producing wells. The literature includes several different machine learning methods such as feedforward neural networks, recurrent neural networks, convolutional neural networks, gradient boosting, support vector machines, and principal component analysis (PCA), to name a few. Bikmukhametov and Jäschke [2020] and [Mercante and Netto, 2022] provide overviews of works applying the different methods. Within steady-state VFMs, feedforward fully connected neural network architectures have been extensively applied in the literature, many of which are highlighted in the previously mentioned review papers. However, as this thesis investigates dynamic VFMs, we limit the following literature review accordingly.

Works relevant to this thesis in the literature study similar machine learning model architectures or examine dynamic data-driven VFMs. A large part of the literature concerning dynamic VFMs has explored the application of LSTM models to different production situations. For instance, Andrianov [2018] models a dynamic VFM using an LSTM architecture to estimate current or predict future gas and oil rates in two simulated operation scenarios that are dynamic and challenging to estimate. The authors show that an LSTM can achieve competitive performance compared with feedforward neural networks and hydrodynamical models. Similarly, Shoeibi Omrani et al. [2018] compare an fully connected neural network (FCN) with an LSTM network on field data exhibiting dynamic and transient operational behavior. They find that the LSTM is more capable of estimating rates under dynamic conditions than an FCN. Sun et al. [2018] estimate future multiphase flow rates using historical flow rates and wellhead pressure from shale wells. When trained on the production history of a single well, the model is able to predict flow rates, and when trained on several nearby wells, the models also show promising results when tested on a new well. Loh et al. [2018] study the performance of a deep LSTM architecture on a production dataset from mature wells with decreasing output. By combining this with an ensemble Kalman Filter (EnKF) that updates the rate estimates based on new observations, they are able to improve the model performance and create a more robust VFM. Finally, Mercante and Netto [2022] compare the performance of FCNs, LSTMs, and GRUs with VFMs that use a combination of either LSTM or GRU and an FCN. These models are trained on wells from publicly available datasets and compared. The GRU/FCN and LSTM/FCN combinations performed the best in five cases each of the ten wells tested. While the mentioned works examine dynamic VFMs using LSTM models and show promising performance results, However, they do not attempt to leverage multi-task learning and depend on flow rate measurements in previous time steps to estimate the next. Both of these points are addressed in this thesis as we study (among others) an LSTM architecture that uses MTL and is able to include unlabeled data in the input sequences. In this manner, our work extends upon and complements the existing literature on LSTM VFMs.

The literature also includes several works examining the use of Convolutional Neural Networks (CNN) and its temporal variant, Temporal Convolutional Networks. While not directly used in this thesis, conventional CNNs have been applied in flow metering problems and represent the main component of TCNs. Therefore, highlighting related works using this model type and how they relate to our thesis may also be insightful. First, Xu et al. [2020] uses a combination of CNNs and LSTM to estimate two-phase oil-gas flow rates in a laboratory experiment. Here, electrical capacitance tomography (ECT) sensors are used in combination with a venturi tube to describe the flow rate. Features relating to the flow composition are extracted from the ECT scans through deep CNNs, before being composed into sequences and used in an LSTM to create rate estimates. This model achieved the lowest relative error out of the models tested. Meanwhile, Zhang et al. [2020] proposes a dynamic VFM using conventional 1D CNNs in a laboratory experiment using a venturi tube, with varying success. In a similar venturi tube laboratory experiment, Wang et al. [2022] compares a CNN-LSTM architecture with a TCN. The TCN obtains considerably higher performance on both liquid and gas rate predictions. Of particular relevance is Gryzlov et al. [2021] who examine the performance of TCNs, LSTMs, and FCNs on two different synthetic datasets, one simulating a simple production scenario and the other describing noisy well test data, from Andrianov [2018]. The results show that the TCN outperforms LSTMs and that both models outperform FCN under dynamic conditions. Interestingly, the results show that the dynamic models are less accurate during steady phases. Additionally, the authors illustrate the considerable increase in computation time when training LSTMs and TCNs compared with FCNs. Note that none of the works in this chapter use field data or learn across wells. Additionally, only one work develops a model that can estimate flow rates using unlabeled data points. Thus, this thesis complements the literature on TCNs by examining the performance of TCNs on field data using MTL and sequences with unlabeled measurements.

Table 2.1 summarizes the related literature discussed in the previous paragraphs. The articles are classified in several ways in the different columns. First, the machine learning model types examined are listed, then whether the main contribution regards steady-state or dynamic VFMs. The next column indicates whether the model uses multi-task learning to learn rate estimation across multiple wells. Then, what sort of data is used to evaluate the models is shown, and finally, whether the model can produce using sequences including elements where the flow rate label is unknown. This final column thus considers the applicability of the proposed model in a situation where flow rate measurements are not known in all preceding time steps. By comparing rows in the table, we may identify contributions of the thesis to the VFM literature. In summary, this thesis represent the following contributions:

1. The thesis contributes to Solution Seeker's research by exploring dynamic VFMs
2. It complements the existing VFM literature by applying dynamic VFMs in an MTL setting with field production data while allowing for the use of unlabeled data points
3. The thesis extends upon existing literature by examining the performance of TCNs and GRUs, two promising sequence model architectures that are not as well-studied as LSTMs
4. It addresses the need for research attempting to overcome the issue of limited availability of data as indicated in Grimstad et al. [2021]

Table 2.1: Overview of related literature with classifications of relevant properties.

Article	Model type	Steady-state or dynamic	Multi-task learning?	Data	Unlabeled data?
This thesis	(RNN/LSTM/GRU/TCN) + FCN	Dynamic	Yes	Field	Yes
Sandnes et al. [2021]	FCN	Steady-state	Yes	Field	No
Løvland [2021]	VAE + FCN	Steady-state	No	Field	Yes
Heggland and Kjærran [2021]	AE + FCN	Steady-state	Yes	Field	Yes
Andrianov [2018]	LSTM/FCN	Dynamic	No	Synthetic	Yes
Loh et al. [2018]	LSTM + EnKF	Dynamic	No	Field	No
Shoeibi Omrani et al. [2018]	LSTM/FCN	Dynamic	No	Field/synthetic	No
Sun et al. [2018]	LSTM	Dynamic	No	Field	No
Mercante and Netto [2022]	(LSTM/GRU) + FCN	Dynamic	No	Field	No
Xu et al. [2020]	CNN + LSTM	Dynamic	No	Laboratory	No
Zhang et al. [2020]	CNN	Dynamic	No	Laboratory	No
Wang et al. [2022]	CNN + LSTM/TCN	Dynamic	No	Laboratory	No
Gryzlov et al. [2021]	TCN/LSTM/FCN	Dynamic	No	Synthetic	Yes

Machine learning literature

The machine learning models discussed in this thesis have proven very effective in different learning tasks within other domains. The recurrent architectures (RNN, LSTM, GRU) have been applied to numerous classical sequential modeling tasks, such as time series forecasting [Siami-Namini et al., 2018], language modeling [Sundermeyer et al., 2012], speech recognition [Graves et al., 2013] and handwriting generation [Graves, 2013] to name a few. Of particular relevance are works that apply sequence models to tasks of similar nature to the flow rate estimation problem. Such tasks often produce one output per sequence input, such as regression or classification. For instance, Li and Qian [2016] applies LSTMs to the problem of sentiment analysis, where emotions of comments from different websites are estimated. In this case, the input sequences come from sentences in the comments, and the output estimates the sentiment of the comment (positive, negative, neutral). Meanwhile, Li et al. [2020] estimate the health of lithium-ion batteries using a modified variant of LSTMs. Here, historical measurements of temperature, current, voltage, sampling time, and corresponding capacity are used to estimate the battery's state of health.

Temporal CNNs have also seen widespread adaptation in recent years. Applications include human action segmentation, detection and recognition [Lea et al., 2017, 2016, Sun et al., 2015, Kim and Reiter, 2017], time series anomaly detection [He and Zhao, 2019], different time series forecasting tasks such as traffic flow forecasting [Zhao et al., 2019], energy demand forecasting [Lara-Benítez et al., 2020] and wind speed forecasting for wind power production [Gan et al., 2021]. Another relevant work is Böck et al. [2019], who apply TCNs to a multi-task learning setting where both the beat and tempo of songs are learned together. In this context, the individual learning tasks greatly benefited from the multi-task learning approach by both learning a common representation and through learning from each other.

Perhaps the most well-known and ubiquitous work in the literature regarding TCNs is DeepMind's WaveNet [Oord et al., 2016], which illustrates the state-of-the-art nature of TCNs. WaveNet is a temporal CNN architecture for raw audio generation. The model may generate audio sampled thousands of times per second and reaches state-of-the-art performance on the text-to-speech problem, and produces more natural-sounding speech to human listeners than comparable models. Interestingly, the model fuels the speech generation in Google Assistant, which is Google's competitor to Apple's famous Siri voice assistant. Another important work relating to TCNs is Bai et al. [2018] which argues that a simple TCN may perform just as well or even outperform conventional RNN architectures on sequence modeling tasks. This is demonstrated by comparing how a simple TCN architecture performs against a vanilla RNN, LSTM, and GRU on common sequence modeling tasks. Here, TCN outperforms the other models in most of the tasks. These works motivate the choice of comparing a TCN to the conventional sequence model for the task of flow rate estimation and illustrate the cutting-edge nature of the methods applied in this thesis.

Chapter 3

Theory

This chapter presents theoretical concepts relevant to the thesis. The methods applied in exploring the research question in later chapters are based on these concepts. We start with a brief introduction presenting the core principles of machine learning in Section 3.1 before describing essential components more in-depth. First, Section 3.2 explains what neural networks are and how they are composed, followed by Section 3.3 showcasing how such networks learn and improve. Then, Section 3.4 discusses evaluation of machine learning models. Next, relevant machine learning paradigms and corresponding example methods are presented in Section 3.5. Finally, an overview of sequence modeling is provided in Section 3.6. In presenting relevant theory, we limit the discussion to only giving brief introductions emphasizing the relevant material for the thesis. Non-essential details are omitted, and we rather refer the interested reader to relevant sources. Furthermore, a substantial amount of the material below is inspired by two main sources: Goodfellow, Bengio, and Courville's book *Deep Learning* [Goodfellow et al., 2016] and the lecture notes from Massachusetts Institute of Technology course 6.036 - Introduction to Machine Learning [Kaelbling, 2019]. Additionally, as the relevant theory for this thesis considerably overlaps that of our project report, Sections 3.1–5 are revised versions from the project report [Heggland and Kjærran, 2021].

3.1 Introduction to machine learning

Machine learning is a subcategory of methods within Artificial Intelligence (AI). The core idea of machine learning is to make computers learn how to solve given tasks, commonly through experience. This notion is not new, and publications relating to the topic of making machines learn date back to the late 1950s and early 60s [Samuel, 1959]. Early research was inspired by the human brain and nervous system, where researchers attempted to replicate the functions of human neurons [McCulloch and Pitts, 1943]. However, many early attempts were constrained by very limited computing power. As computer technology improved, AI and machine learning methods became increasingly proficient at solving tasks. An early example is IBM's Deep Blue system, which used AI techniques to learn to play chess so proficiently that it beat the world champion, Gary Kasparov, in 1997 [Campbell et al., 2002]. Today, machine learning is a widely popular field with great research activity and numerous successful applications. This recent trend is enabled by the increased availability of data and computing power [Goodfellow et al., 2016].

On a very general level, machine learning can be described as follows. A model is designed with the intention of making it learn to perform some task. To induce learning, the model is exposed to some kind of experience. Using this experience, the model produces outputs. Some performance measure then evaluates the quality of the outputs. This process is orchestrated through a learning algorithm [Goodfellow et al., 2016]. We say that a computer program learns from experience with respect to some task and performance measure if the performance on the given task as measured by the given performance measure increases with the experience [Mitchell, 1997]. When the model performance is evaluated against the performance measure, the learning algorithm somehow corrects or alters the model's behavior based on the performance.

To give a more concrete example, consider a model that should learn to classify input images as pictures of either cats or dogs. The experience is, in this case, data in the form of images. First, the model attempts to classify an image, and then the performance is evaluated based on some performance measure related to whether it classified the image correctly. If it is incorrect, a learning algorithm updates the model such that its behavior is slightly altered in an intelligent manner, making it more likely to classify similar images correctly next time. As this process is repeated over many training examples (images), the model improves and learns to discern images of cats from images of dogs.

3.2 Neural networks

Neural networks are one of the most popular modeling architectures within machine learning and are especially relevant to this thesis. As the name suggests, neural networks are inspired by the networks of neurons in humans, in addition to the human brain [McCulloch and Pitts, 1943, Rosenblatt, 1958]. The following introduction of neural networks is based on Kaelbling [2019]. A neural network (in the machine learning sense) is composed of neurons. A single neuron is a function mapping from some input $x \in \mathbb{R}^m$ to an output $a \in \mathbb{R}$. This function is calculated as follows: each element x_j of the input vector is multiplied by some weight w_j and then summed together. In addition, a bias value $b \in \mathbb{R}$ is added to the sum. This produces the *pre-activation* z, which is then inputted to some *activation function* f to produce the output a. Let $w \in \mathbb{R}^m$ represent the weight vector. See Figure 3.1 for an illustration of the computation. This computation can then be written as follows:

$$a = f(z) = f(\sum_{j=1}^{m} w_j x_j + b) = f(w^T x + b)$$

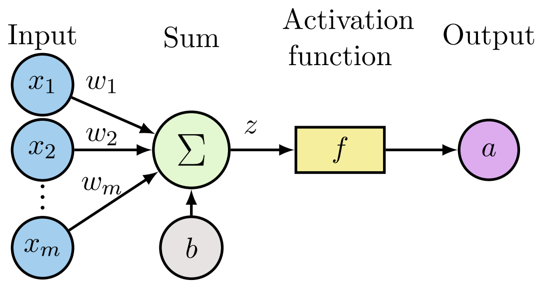


Figure 3.1: General composition of a single neuron in a neural network. Inputs x_1 to x_m are multiplied by weights w_1 to w_m to produce pre-activation z, which is input in activation function f to produce output a.

Neurons in a neural network are commonly organized in layers connected through weights. Here, any given neuron takes in the outputs from the previous layer multiplied by weight values for each pair of neurons. If a given layer has an input column vector $X \in \mathbb{R}^m$ and consists of n neurons or units, the layer produces an output column vector $A \in \mathbb{R}^n$. Each of the m input values is connected to each of the n output units through weights. Every input value to the layer is multiplied by the corresponding weight connecting to a given output unit to produce the pre-activation for that unit. Let $Z \in \mathbb{R}^n$ denote the n pre-activations. Let $W \in \mathbb{R}^{m \times n}$ denote the weight matrix representing n weight column vectors $w_j \in \mathbb{R}^m$. Let $B \in \mathbb{R}^n$ denote the bias column vector containing the bias values b_j for the n units. The output of a given layer can then be computed as

$$A = f(Z) = f(W^T X + B)$$

See Figure 3.2 for an illustration of the structure of a neuron layer in a neural network.

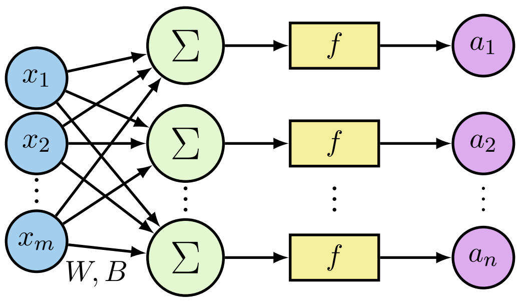


Figure 3.2: Composition of a layer in a neural network. m input values $x_1, \ldots, x_m$ are mapped to n output values $a_1, \ldots, a_n$ by multiplying the input column vector by the $m \times n$ weight matrix W and adding the $n \times 1$ bias column vector B to the product to compute n pre-activations. The final outputs are then computed by inputting the pre-activations into the activation function f.

Neural networks typically consist of several such layers, where the activation of one layer becomes the input of the next layer. Typically, neurons in a given layer are connected to neurons in the subsequent layer. This architecture is commonly referred to as a *feedforward neural network*. Furthermore, in *fully connected feedforward neural network* (FCN), each neuron in a given layer is connected to *all* neurons in the subsequent layers. This variant is used in this thesis, and an example illustration is shown in Figure 3.3. Furthermore, each neuron may use a different activation function. However, all neurons in a given layer typically use the same activation function. We commonly refer to the first layer taking in the original input as the *input layer* and the final layer producing the final output value(s) as the *output layer*. All the layers between are referred to as the *hidden layers*. The result of such a network is a computational structure able to approximate a broad range of functions. The expressive capabilities depend on the structure, based on the number of units, layers, and type of activation functions, among other factors. The models ability to fit different functions is often referred to as its *capacity* [Goodfellow et al., 2016].

Several popular activation functions exist. Note that if a neural network uses the identity function or other linear transformations as the activation function in all neurons, the entire network can be reduced to a single linear mapping of the input, giving a linear function. Therefore, non-linear activation functions must be included to approximate a wider range of functions. One popular activation function is the *rectified linear unit* or ReLU function, defined as

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} = \max(0, x)$$

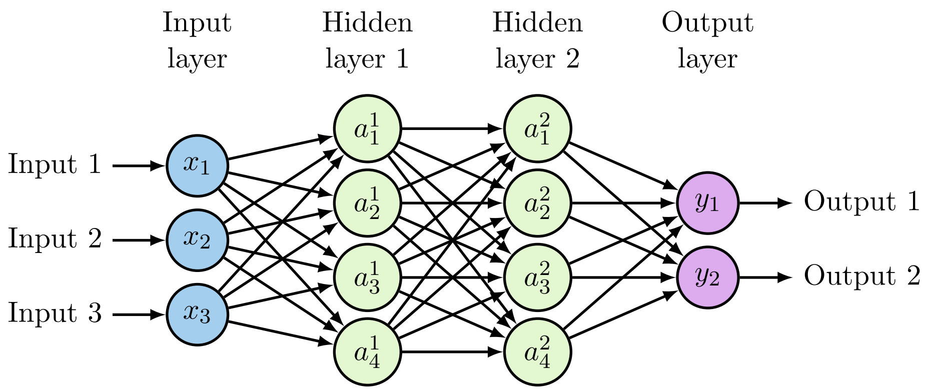


Figure 3.3: Example of a fully connected feedforward neural network. Each circle represents a neuron, and the arrows indicate connections between neurons through weights.

simply forcing negative inputs to zero. This activation function is very commonly used in hidden units, as it allows for fast and trivial differentiation, among other advantages [Goodfellow et al., 2016].

Another commonly used activation function is the *sigmoid function*, defined as

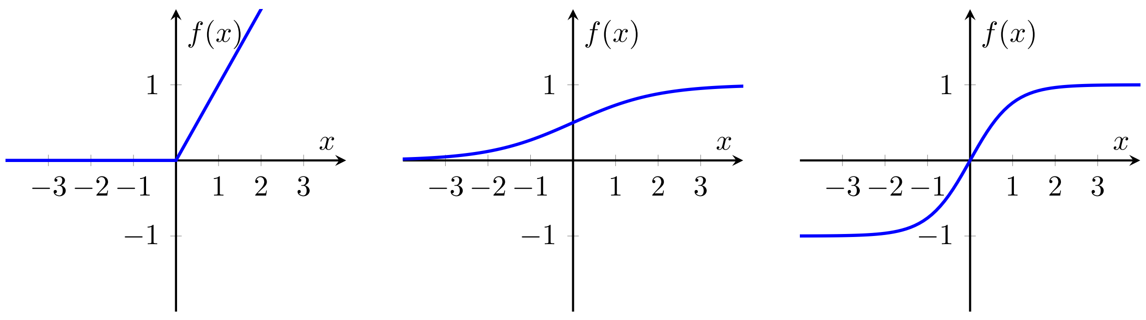
$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

mapping inputs to the interval $[0, 1]$. This is an example of what is sometimes called a *squashing non-linearity* as it maps the input to a finite interval. This activation function is commonly used in situations where a weighting between zero and one is needed, as in Long Short-Term Memory (LSTM) models (see Section 3.6), and when the output is to be interpreted as a probability.

Finally, the *tangent hyperbolic function* (tanh) is a third non-linear function commonly used as an activation function. Tanh is defined as

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}} = \frac{1 - \mathrm{e}^{-2x}}{1 + \mathrm{e}^{-2x}}.$$

Consequently, tanh maps inputs to the interval $[-1, 1]$. This is another key non-linearity in recurrent neural network cell structures (see Section 3.6). The three mentioned activation functions are visualized in Figure 3.4.



(a) Rectified Linear Unit **(b)** Sigmoid **(c)** Tangent hyperbolic

Figure 3.4: Visualization of ReLU, sigmoid and tanh activation functions.

3.3 Training process

As in the previous section, the following explanations are based on Kaelbling [2019]. To induce learning in machine learning models, their performance is measured and used to alter the models to perform better (with respect to the performance measure) in the future. To measure the performance of models, we use a *cost function* or *loss function* which compares the model's output with the desired output. In the context of neural networks, this may be denoted as follows. Let L be some loss function, x the input to the neural network NN parameterized by weights W and biases B, and y the desired output. The loss can then be calculated as

$$L(NN(x; W, B), y). \tag{3.1}$$

For neural networks, we may then induce learning by altering the parameters of the network based on the loss calculation. This is done using *gradient descent.* Gradient descent is an optimization method that updates the network's weights by computing the gradient of the loss with respect to each weight. The gradients indicate which direction the weight should be changed to increase loss. Thus, weights are updated in the negative direction of the gradient, descending towards lower loss. Given a step size η, the weight update for weight w_j using gradient descent can be written as

$$w_j = w_j - \eta \frac{\partial loss}{\partial w_j}$$

where we take a step of length η in the direction of decreasing loss.

To update the weights, we need to calculate the gradient of the loss with respect to each weight. As the loss is a function of the neural network's output, and the neural network's output is a composition of functions, the chain rule can be used to identify the gradients. First, the neural network produces the output by inputting x, calculating the output of the first layer, passing this forward to the next layer and so on until a final output is produced. This process is called a *forward pass*, represented by $NN(x; W, B)$ in Equation (3.1). Then, the loss is calculated to find the gradients. Let W^l denote the weights in layer l, and let L denote the final layer. Let Z^l and A^l denote the pre-activation and activation in layer l, respectively. The computation of the gradients can then be written somewhat informally as

$$\frac{\partial loss}{\partial W^l} = A^{l-1} \cdot \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \frac{\partial A^{l+1}}{\partial Z^{l+1}} \cdot \ldots W^{L-1} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial loss}{\partial A^L} \tag{3.2}$$

where $\frac{\partial loss}{\partial A^L}$ depends on the loss function. In short, this result comes from applying the chain rule to compute $\frac{\partial loss}{\partial W^l}$ and the fact that (informally)

$$\frac{\partial Z^l}{\partial W^l} = \frac{\partial W^{l^T} A^{l-1}}{\partial W^l} = A^{l-1} \quad \text{and} \quad \frac{\partial Z^l}{\partial A^{l-1}} = \frac{\partial W^{l^T} A^{l-1}}{\partial A^{l-1}} = W^l.$$

For a derivation and more detailed explanation of the above result, see Kaelbling [2019]. In this manner, after a forward pass, the gradients with respect to the weights of each layer are given by starting in the final layer, computing the gradient, and propagating the loss contribution backward. Hence, this method is commonly referred to as *back-propagation.*

When training a neural network, the following procedure is commonly conducted. First a *batch* of training examples $\{x^{(i)}\}_{i=1}^N$ is obtained, with corresponding true outputs or *labels*

$\{y^{(i)}\}_{i=1}^{N}$. Then, we iterate over this batch and input each example $x^{(i)}$ into the network to produce outputs $\hat{y}^{(i)}$ (this is commonly done in vectorized form, so all outputs in the batch are computed simultaneously). The loss is commonly calculated over the whole batch, using a loss function aggregating over multiple examples. An example of such a loss function is the Mean Squared Error (MSE) for regression problems where the network should estimate real-valued numbers (as opposed to, for instance, classification problems). The MSE loss over the batch can then be calculated as

$$L(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$$

or equivalently, as the mean of N individual squared error loss calculations.

When using batches, *batch gradient descent* is performed by summing the gradients of the loss over the batch when updating the weights. If the batch contains more than one example and fewer than all training examples, this process is commonly referred to as *minibatch gradient descent* [Goodfellow et al., 2016]. Given a per-example loss function L, the weight update is then specified as

$$W = W - \eta \sum_{i=1}^{N} \nabla_W L(NN(x^{(i)}; W, B), y^{(i)}). \tag{3.3}$$

Alternatively, we can randomly select single examples, calculate the loss with respect to the example and perform a weight update using this single example. This method is called *stochastic gradient descent* (SGD).

A final, prevalent variant is *minibatch stochastic gradient descent* which selects N training examples uniformly at random and performs the weight update based on these examples' contribution to the gradient. This gives a weight update scheme that is equivalent to Equation (3.3), with the only difference being that the training examples are randomly selected.

The different approaches towards performing the weight update are commonly referred to as *optimizers* as they attempt to optimize the neural network with respect to the loss function. One popular optimizer of significant relevance to this thesis is *Adam*, which combines the concept of minibatch stochastic gradient descent with the use of adaptive step sizes during the weight update [Kingma and Ba, 2014]. The step sizes are adjusted based on the topology of the space formed by the weight parameters and the loss function. Step sizes are calculated by estimating the first and second moments of the gradients and accounting for the magnitude of the gradient. The main idea is to avoid the weight updates from moving back and forth in a given direction (hence the estimation of momentum) and take larger steps when the space defined by the loss function is close to flat (as indicated by the magnitude of the gradient). Likewise, shorter steps should be taken when the topography is steep to avoid stepping over local minima. See Kingma and Ba [2014] for further details on Adam.

All the components required to understand the process of training a neural network have been presented. We can summarize this process in an algorithm, see Algorithm 1. We input a dataset $\mathcal{D}_N$ containing N training examples, an integer L indicating the number of network layers, an integer T specifying the number of weight updates to perform on each weight, a set $\{m^1, \ldots, m^L\}$ indicating the number of units in each layer, a set $\{f^1, \ldots, f^L\}$ specifying the activation function in each layer and a loss function *Loss*. For each weight update t, a step size $\eta(t)$ is calculated. Also note that the gradient calculations in the algorithm can be related to the gradient calculation presented in Equation (3.2) by the fact that $\frac{\partial Z^l}{\partial W^l} = A^{l-1}$ and $\frac{\partial Z^l}{\partial A^{l-1}} = W^l$.

Algorithm 1 Process of training a neural network

```
Train-NN($\mathcal{D}_N, L, T, \{m^1, \ldots, m^L\}, \{f^1, \ldots, f^L\}, Loss$)
  for $l = 1 \ldots L$ do                                  ▷ Randomly initialize network parameters
    $W^l \sim \mathcal{N}(0, \frac{1}{m^l})$                                    ▷ Normal distribution
    $B^l \sim \mathcal{N}(0, 1)$
  for $t = 1 \ldots T$ do
    $i$ = random sample from $\{1, \ldots, N\}$
    $x^{(i)}$ = corresponding random training example from $\mathcal{D}_N$
    $A^0 = x^{(i)}$
    for $l = 1 \ldots L$ do                 ▷ Forward pass to produce output of the network $A^L$
      $Z^l = W^{lT} A^{l-1} + B^l$
      $A^{l+1} = f^l(Z^l)$
    $loss = Loss(A^L, y^{(i)})$
    for $l = L, \ldots, 1$ do
      $\frac{\partial loss}{\partial A^l} = \frac{\partial loss}{\partial Z^{l+1}} \cdot \frac{\partial Z^{l+1}}{\partial A^l}$ if $l < L$ else $\frac{\partial loss}{\partial A^L}$          ▷ Error back-propagation
      $\frac{\partial loss}{\partial Z^l} = \frac{\partial loss}{\partial A^l} \cdot \frac{\partial A^l}{\partial Z^l}$
      $\frac{\partial loss}{\partial W^l} = \frac{\partial loss}{\partial Z^l} \cdot \frac{\partial Z^l}{\partial W^l}$          ▷ Calculate gradients with respect to parameters
      $\frac{\partial loss}{\partial B^l} = \frac{\partial loss}{\partial Z^l} \cdot \frac{\partial Z^l}{\partial B^l}$
      $W^l = W^l - \eta(t) \frac{\partial loss}{\partial W^l}$                    ▷ Gradient descent weight updates
      $B^l = B^l - \eta(t) \frac{\partial loss}{\partial B^l}$
  Return $\{W^l\}_{l=1}^L$, $\{B^l\}_{l=1}^L$                   ▷ Output learned network parameters
```

3.4 Model evaluation

The process described in Section 3.3 allows us to make a neural network learn and improve the model's performance through experience (data). This training process yields a model which performs well on the given learning task when inputting the data used to train the model, given that the model capacity is large enough. To evaluate the model's ability to generalize to different inputs, we test the model on a *holdout set* or *test set*. This is a set of input examples on which the model has not been trained. Ideally, these examples should be drawn from the same underlying distribution as the training examples. The model's ability to perform well on these examples represents the model's ability to generalize. In machine learning, a model that yields a high *training* loss is said to *underfit* the training data, while a model that yields a high gap between training loss and test loss is said to *overfit* the training data [Goodfellow et al., 2016]. An underfit model thus estimates the training examples poorly, typically resulting from insufficient representational capacity. That is, the model is typically too simple, having too few parameters. Conversely, an overfit model has been over-specialized to the training data to the extent that its ability to generalize has suffered, as indicated by a higher test loss than training loss.

Several approaches exist to counteract underfitting and overfitting, commonly referred to as *regularization* techniques. Here, we only mention techniques relevant to this thesis. A comprehensive overview of other regularization methods can be found in, for instance, Goodfellow et al. [2016]. One form of regularization is L_2 *regularization* or *weight de-*

cay. The technique limits the magnitude of the weights of a neural network by adding minimization of the sum of the L_2 norm of the weights in the loss function. Given an L_2 regularization factor λ, and a set of weights and biases W, a loss function with L_2 regularization can be written as

$$Loss(NN(x; W), y) + \lambda \sum_{w \in W} \|w\|_2^2.$$

The motivation behind L_2 regularization is the assumption that weights with smaller magnitudes lead to simpler models and that simpler models are less likely to over-specialize and overfit [Goodfellow et al., 2016].

Dropout is a different widespread regularization technique. When using dropout, a set of non-output units in the network are selected at random to be disabled in each training iteration for each training example. That is, the output of these units is set to zero for that iteration. This has the effect of preventing these units from contributing or participating in producing the output. Consequently, the remaining units of the network must share the responsibility of generating an ideal output. In this manner, the network becomes more robust to perturbations, as it prevents a smaller subset of the network from being responsible for most of the critical computation required to solve the learning task. Note that when the output of a unit is set to zero, the weights of that unit will not be updated in the backward pass as the contribution (gradient) will be zero. See Figure 3.5 for a visualization of dropout. In practice, dropout is commonly implemented by multiplying the activation of each unit during the forward pass with a randomly drawn binary indicator:

$$a^l = f(z^l) \cdot d^l$$

where a^l is the activation of each unit in layer l, $f(z^l)$ denotes the pre-activation of each unit in layer l being passed through an activation function f, and d^l represents a randomly drawn binary column vector which is element-wise multiplied with the activation. For each element in d^l corresponding to a unit in the layer, a value of zero is drawn with probability p, otherwise, a value of one is selected. d^l thus indicates whether each unit should be deactivated. This vector is then drawn again for each consecutive training step. During inference, instead of deactivating outputs, all weights are multiplied by p to achieve the same averaging effect on activations.

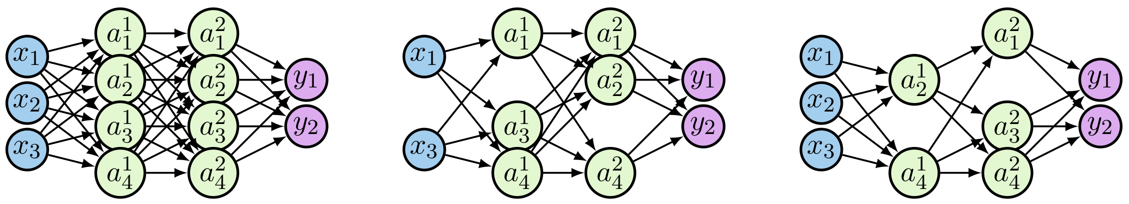


Figure 3.5: Example visualizations of dropout. The leftmost network is the original model with no disabled units. The other networks show examples of random selections of units whose outputs are set to zero during a training step, thus not contributing to the output or the gradient update.

Another regularization technique is *early stopping*. This method involves evaluating the model on a *validation set* after training the model on a number of training examples. Typically, the model is evaluated on the validation set after each run through the entire training set (after each *epoch*). The validation loss is then recorded and monitored over epochs. If the validation loss starts to increase systematically, the training procedure is terminated, and the weights corresponding to the lowest validation loss are reported [Goodfellow et al., 2016]. Note that the network does not train on the validation data directly, as we do not perform back-propagation using examples from this set. However, as the training is monitored by evaluating the model on the validation data, we implicitly use the dataset during training. Thus, the validation set should not be used to perform the final evaluation of the model. Instead, a different test set that has not been used during training should be used. This ensures an unbiased evaluation of the final model's ability to generalize.

A technique commonly used to identify high-performing neural networks is *hyperparameter tuning*. This is a form of optimization where different network configurations are trained and compared based on validation loss. Typically, a number of parameters such as the L_2 regularization factor or the number of layers are considered adjustable. Hyperparameter tuning is then performed as follows, using some optimization method. First, we decide values for each adjustable parameter using a sampling strategy, train the network and evaluate it on the validation data yielding a validation loss. The optimization method suggests new, carefully selected hyperparameter values based on this loss. Then, the training process is repeated using the new hyperparameters to produce a new validation loss. This process is repeated a number of times, where the optimization method directs the search towards lower validation loss by tuning the hyperparameters. In this way, we can say that hyperparameter tuning tends to generate models that overfit the validation data. Several methods exist for optimizing the loss functions. Perhaps the most simple approach is to perform a grid search where the objective function is evaluated for each combination of hyperparameter values in a grid of possible values. A more advanced optimization algorithm used in this thesis is the multivariate Tree-Parsen Estimator (TPE) sampler described in Falkner et al. [2018]. The sampler is used to guide the parameter search, which jointly considers the performance of previously sampled values across hyperparameter domains when suggesting new values.

3.5 Learning paradigms

The previous sections explained how we could train and evaluate a neural network. This forms the foundation for numerous variations of machine learning methods. This section presents different important learning paradigms discussed in the thesis. We start by describing supervised learning in Section 3.5.1, followed by unsupervised learning in Section 3.5.2, semi-supervised learning in Section 3.5.3 and finally multi-task learning in Section 3.5.4. We emphasize that we provide only a very brief introduction to each paradigm sufficient to comprehend discussions in the thesis. The following subsections are based on Goodfellow et al. [2016].

3.5.1 Supervised learning

Supervised learning is a form of learning where the training process is *supervised*, typically in the form of penalizing poor estimates as compared with some target or *label* measured by a loss function. Commonly, the labels are annotated by humans. Consequently, supervised learning datasets are, in many cases, time-consuming and costly to acquire. Classification and regression problems are typical learning problems within supervised learning. Here, training data includes some sort of labels, representing which class the input data corresponds to in the classification case and real-valued numbers corresponding to the input data in the case of regression. Such models are typically trained as described in the previous sections. Common methods used within this paradigm include different versions of neural networks, such as feedforward neural networks for regression tasks, convolutional neural networks for image classification and object detection tasks, and recurrent neural networks for natural language processing and other sequence-related tasks.

3.5.2 Unsupervised learning

In unsupervised learning, models learn to perform some learning task without supervision. Thus, such models train on data without any corresponding label (i.e., *unlabeled data*). Generally, methods where humans are not required to annotate target outputs are considered unsupervised. A typical example of an unsupervised learning problem is clustering. This could be, for instance, a clustering problem where data points are to be grouped such that the euclidean distance between points within the same cluster is minimized. K-means clustering is one such method, where a set of data points is to be clustered into k clusters. Another learning problem often considered an unsupervised learning problem is feature extraction or representation learning, where the intention is to learn the “best” representation of the data possible [Goodfellow et al., 2016]. Common methods used for this are Principal Component Analysis and autoencoders. An autoencoder is a model trained to reconstruct the input while being subjected to some form of restriction, making a perfect reconstruction impossible. Consequently, the model is forced to learn to extract informative features to reconstruct the input as correctly as possible. Thus, the encoder part of the model effectively becomes a feature extractor. This approach played a core role in our project thesis [Heggland and Kjærran, 2021]. See Figure 3.6 for an example of a feedforward neural net autoencoder structure.

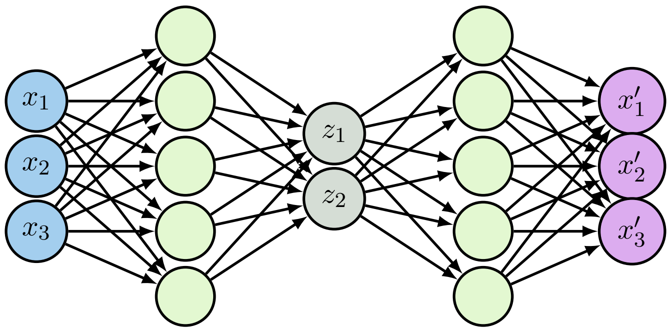


Figure 3.6: An example illustration of a feedforward autoencoder. The first three layers represent the encoding part of the autoencoder, whereas the last three layers represent the decoder. Note that the middle layer, whose activation z represents the latent encoding of the input x, is both the encoder module's output and the decoder's input. The decoder outputs a reconstruction x'.

3.5.3 Semi-supervised learning

Semi-supervised learning is a learning paradigm that combines the principles of supervised and unsupervised learning [Chapelle et al., 2006]. Such learning methods are commonly used when a combination of labeled and unlabeled data is used to perform a learning task. For instance, pre-training a model using unlabeled data in unsupervised methods before fine-tuning it to solve a supervised problem using labeled data is an example of a semi-supervised learning method. An example of this is given in Section 2.6. Thus, such methods can perform well in situations where large amounts of unlabeled data are available and only smaller amounts of labeled data are available. This was the topic of our project report, where we combined unsupervised autoencoders for feature extraction with supervised learning for flow rate regression using feedforward neural networks [Heggland and Kjærran, 2021]. Conversely, this thesis explores an alternative approach to leveraging the unlabeled data by use of sequence models (see Section 3.6). See Chapelle et al. [2006] for a thorough introduction to semi-supervised learning and Van Engelen and Hoos [2020] for a more recent extensive survey.

3.5.4 Multi-task learning

In multi-task learning, multiple learning tasks are performed with the same model. For instance, a model that learns to both identify road lanes in images and classify objects in the same image can be considered a multi-task learning model. The motivation behind such models is that the underlying information used to solve each task may be similar and that learning patterns and relationships within one task may be useful in the other tasks as well [Caruana, 1997]. In this way, learning the tasks together could prove more successful than producing two independent models. In addition, multi-task learning has the advantage of parameter sharing, where the learning tasks share trainable parameters when combined into one model. Thus less trainable parameters are required to solve the learning tasks.

A wide range of architectural variations exists concerning how to combine the tasks and how to condition on them. For instance, a neural network designed for Multi-task learning (MTL) could share a common "backbone" of layers among all tasks (common layers) before separating into different independent heads for each task (separate layers), see Figure 3.7. Numerous far more advanced approaches to conditioning on the tasks exist. See for instance Ruder [2017] for an overview. It is not immediately obvious how loss functions should be designed and how to train such multi-task learning architectures. Several approaches exist, and sensible approaches will vary based on the nature of the learning tasks. Perhaps the simplest alternative is to minimize a weighted sum of the cost functions of each task and train the network in a conventional fashion. See Stanford University [2021] for an introduction to this topic.

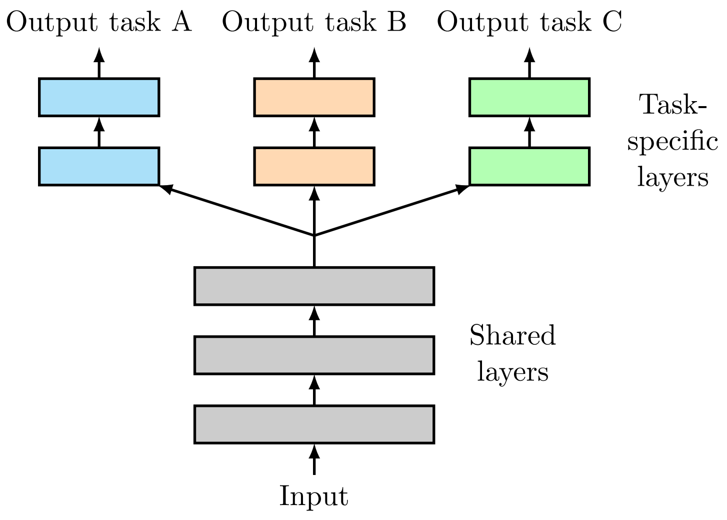


Figure 3.7: An example MTL architecture, where the first neural network layers are shared before separating into task-specific layers. This architecture performs all tasks when given input.

The example architecture described above performs all specified tasks when given an input. Another alternative is to use a neural network architecture that performs one of the tasks when given an input. In this case, the input to the network includes a task-specific indicator specifying which task to perform. See Figure 3.8 for an illustration. This architecture can be useful in situations where the tasks are somewhat homogeneous or similar in nature, as it allows for a high degree of parameter sharing between tasks. Consequently, this design is suitable for multiphase flow rate estimation, where each task represents the estimation of flow rates of a given well. In this case, each learning task is very similar, albeit not identical, as wells do not behave identically. Learning across wells can then be achieved through parameter sharing. The part of the input identifying which task to perform (which well to estimate flow rates for) is used by the network to infer well-specific patterns. Meanwhile, the experience from one well can be used to perform more informed inferences on other wells. Training such a model is rather simple, as we can simply calculate the loss with respect to the given task and update the weights accordingly, like in single-task learning described in earlier sections. This approach is applied in Sandnes et al. [2021] and is also used in this thesis.

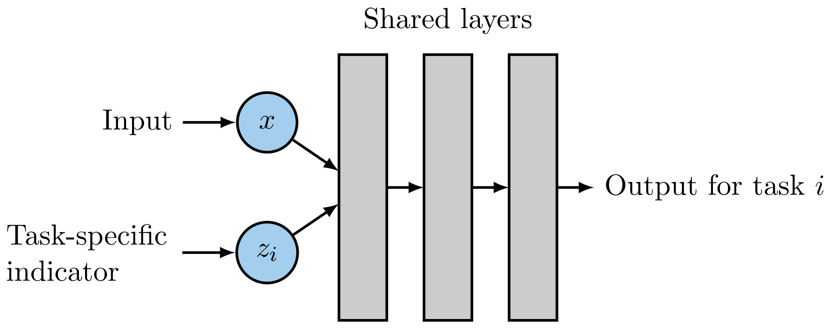


Figure 3.8: An alternative MTL architecture, where all layers are shared between tasks, and the task to perform is specified by a task-specific indicator z_i in the input to the model.

3.6 Sequence modeling

So far, we have mainly considered feedforward networks with fully connected layers. However, several other types of networks exist. This section describes network variants that are especially relevant for processing sequences. We start by giving an introduction to the core concepts of sequence modeling in Section 3.6.1 before describing recurrent neural networks in Section 3.6.2

3.6.1 Introduction to sequence models

In general, sequence models refer to models that operate on sequential data. A sequence represents ordered information, such as data points ordered in time (time series) or a string of words forming a sentence. Thus, sequence models are relevant in problems where such information is present either as input, output, or both. A sequence model can therefore be described as a function mapping from a fixed-size vector or a sequence of vectors to either a fixed-size vector or a sequence of vectors [Goodfellow et al., 2016].

Formally, we may denote the input as a vector x or a sequence of vectors $x_1, x_2, \ldots, x_n$. Likewise, we denote the output as a vector y or a sequence of vectors $y_1, y_2, \ldots, y_m$. The sequence model is denoted f. This gives a natural classification of sequence models:

One to many:	$f(x) = y_1, y_2, \ldots, y_m$	Mapping from vector to a sequence
Many to one:	$f(x_1, x_2, \ldots, x_n) = y$	Mapping from sequence to vector
Many to many:	$f(x_1, x_2, \ldots, x_n) = y_1, y_2, \ldots, y_m$	Mapping from sequence to sequence

In the many-to-many case, some models may allow $n \neq m$ while others may only support $n = m$.

Crucially, any output associated with a given time step is computed using only inputs that are already observed, i.e. inputs from the given time step and past steps. Hence, given an input sequence $x_1, x_2, \ldots, x_n$, a sequence model may only use input elements $x_1, x_2, \ldots, x_t$ to compute the output y_t at time t.

Furthermore, sequence models may compute the output in a single operation or in a sequential manner, where for each step, a new input is processed, a new output is generated, or both. Such models are often stateful and update an internal state every processing step. Recurrent neural network (RNN) models operate in this manner. In such models, one may differ between many-to-many models that process a single element of the input sequence before the first output is produced (second from right in Figure 3.9) and those that process the entire input sequence first (rightmost variant in Figure 3.9). See Figure 3.9 for an illustration of the different variants.

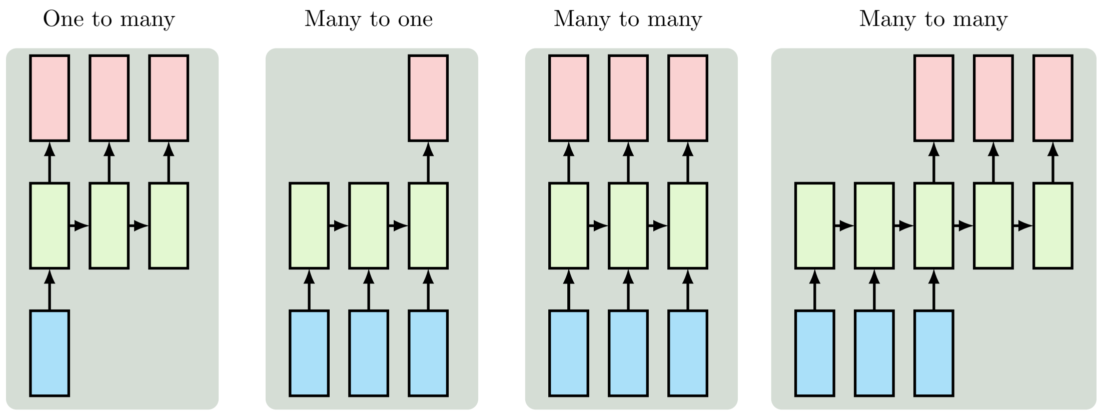


Figure 3.9: Visualization of different sequence model variants. The input is either a single vector or a sequence of vectors which are then sequentially processed by the model to produce an output vector or sequence of vectors. Figure inspired by Karpathy [2015].

The problem dictates the desired inputs and outputs. For instance, sentiment analysis and time series classification are examples of problems where a many-to-one sequence model would be appropriate. These are problems where an input sequence is to be mapped to a vector of values (sentiment scores or classification probabilities). Meanwhile, image captioning is an example of a problem where one-to-many sequence models are relevant. Here, an input image represented as a fixed-size vector is to be mapped to a sequence of words describing the image. Finally, many-to-many sequence models are often used in, for instance, time series forecasting and language modeling, where an input sequence is used to produce an output sequence.

There exist many different sequence models. Historically, time series analysis has been one of the most prominent application areas, containing numerous sequence models operating on time series. For instance, a wide array of statistical models have been implemented within this field. Famous statistical models include Autoregressive Integrated Moving

Average (ARIMA) and its numerous extensions, least squares regression models, and state space models. See, for instance, Shumway et al. [2000] for an extensive introduction to classical time series models.

The recent success of machine learning gave rise to several sequence model variants based on machine learning architectures. One of the first widely adopted architectures was recurrent neural networks (RNNs), a family of neural networks processing sequential data. Recently, two different network architectures have become prominent: temporal CNNs (TCNs) and attention-based models. The latter is similar to RNNs but differs in how they model dependencies between information from different time steps in input sequences. Where RNNs sequentially update an internal state (see details in Section 3.6.2), attention-based models simultaneously observe the entire input, allowing them to exploit dependencies between elements in all time steps more effectively. Consequently, attention-based models can be considerably more complex and require large amounts of computation to perform well. Transformer models are examples of attention-based models and have recently reached state-of-the-art performance within language modeling tasks. See for instance OpenAI's GPT-3 Brown et al. [2020]. While attention-based models have seen the most widespread application within language modeling, temporal CNNs are a fairly recent innovation that has also seen success within other sequence modeling tasks of a more temporal nature. RNNs and TCNs are studied in this thesis and are therefore especially relevant. The models are described in detail in the following subsections.

3.6.2 Recurrent neural networks

Recurrent neural networks (RNNs) are a family of neural networks that process sequential data [Goodfellow et al., 2016]. This subsection covers relevant theory on RNNs, from the basic concepts to the advanced architectures relevant to this thesis. The explanations are inspired by Olah [2015].

The core idea behind RNNs differentiating them from traditional feedforward networks is the principle of memory. Intuitively, for a model to capture dependencies in a sequence over time, the model must somehow remember information that has been input in earlier time steps (and from earlier sequences). Memory can be accomplished by storing a state in the network. This state may then be updated every time step when a new element of the sequence is passed through the network. Furthermore, the output generated in each step depends on this state. In this manner, RNNs can be considered state machines. The main goal of RNNs is to learn network parameters that allow the network to capture task-critical information in the internal state, and that optimally transform the internal state to the ideal output vectors. The internal state is often called the *hidden state.* In practice, the persistence of information is accomplished through loops in the network structure. The hidden state from a given time step is part of the input used to compute the hidden state in the next time step, hence the name "recurrent" networks. An illustration of this is provided in Figure 3.10.

To elucidate the core principles of RNNs, consider the example structure illustrated in Figure 3.10, resembling an RNN in its simplest form. For a given time step t, a recurrent network A considers the corresponding input vector x_t from the input sequence. The network computes a new hidden state h_t using x_t and the hidden state from the previous time step h_{t-1}. h_t is then output from the network and looped back into A for the next time step. h_t is computed using a function f of the input x_t, the previous hidden state

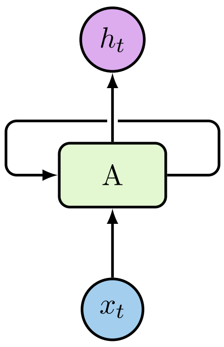


Figure 3.10: A simple RNN structure. An input vector x_t and the previous hidden state is processed by the RNN A to compute the new hidden state h_t, which is output and fed back for the next input. Figure inspired by Olah [2015].

h_{t-1} and the network parameters W:

$$h_t = f(x_t, h_{t-1}; W).$$

In RNNs, the network typically includes trainable parameters associated with the input W_x, the hidden state W_h and a bias b. Given an activation function f, the hidden state is then computed as

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

This architecture is often called a "vanilla" or "standard" RNN as it represents an RNN in its simplest form, and the activation function f is then commonly chosen to be $\tanh(x)$.

The feedback loop in the RNN allows information to persist between time steps. Thus, for each time step t, the network considers a new input x_t, outputs a new hidden state h_t, and propagates the hidden state through the loop to the next time step. A more intuitive way of visualizing this structure can be obtained by unfolding A in time, as seen in Figure 3.11. When viewing the structure in this way, it is important to remember that A represents the same network in all time steps, with the same parameters. Meanwhile, it allows us to see how information is propagated across time, as the hidden state is fed back into A for the next time step.

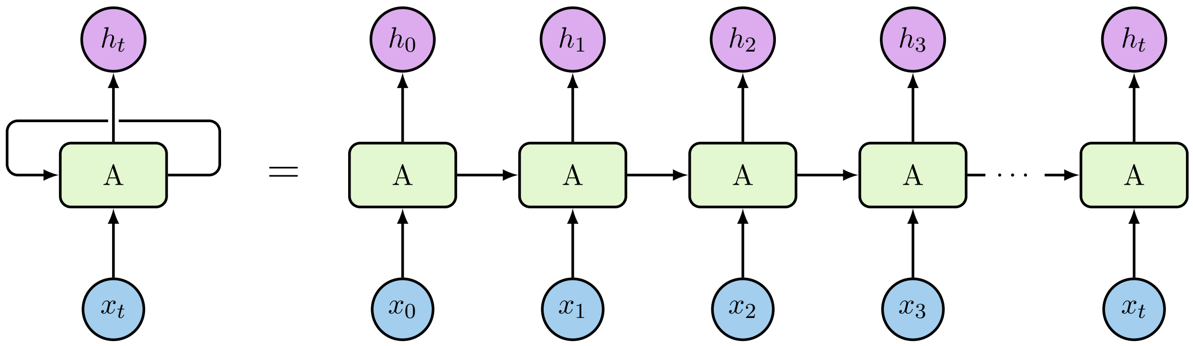


Figure 3.11: Unfolding an RNN across time. An input sequence $x_0, x_1, \ldots, x_t$ is input in the RNN A to produce an output sequence $h_0, h_1, \ldots, h_t$. Note that the same parameters defining A is used to produce the output in every step. Figure inspired by Olah [2015].

Note that in the simple example discussed here, the hidden state is the output in each iteration. In practice, another neural network layer may be used to learn a transformation from the hidden state to the desired output. Additionally, as the RNN discussed here produces an output for each time-step, it is considered a many-to-many sequence model.

However, RNNs can be designed to fit any of the different sequence model variants described in Section 3.6.1 by altering the design of inputs and outputs in the model. A many-to-one RNN can be designed by only outputting a vector after the final element of the input sequence is processed but still updating the hidden state when processing each input element. Likewise, a one-to-many RNN can be designed by computing a hidden state based on a single input vector. Then, for each output element, a hidden state is computed based on the current hidden state and is output and fed back to produce the subsequent output.

The view of the RNN network unfolded in time also makes it easier to conceptualize how sucha network may be trained. The training process can be regarded in much the same way as in feedforward networks. However, one essential difference is that the RNN uses the same parameters to compute each output. In the following, we consider the many to many RNN from Figure 3.10, but a similar procedure is followed for the other RNN variants. Given a training example consisting of an input sequence $x_1, x_2, \ldots, x_n$, a corresponding label sequence $y_1, y_2, \ldots, y_n$, and a per-element loss function $L(h_t, y_t)$, we can train the RNN as follows. First, the input sequence is forward passed through the RNN to produce outputs $h_1, h_2, \ldots, h_n$. Based on this, we want to identify the contribution of each parameter to the loss of the entire predicted sequence. That is, we want to identify $\frac{\partial L_{seq}}{\partial W_x}$, $\frac{\partial L_{seq}}{\partial W_h}$ and $\frac{\partial L_{seq}}{\partial b}$ where $L_{seq} = \sum_{t=1}^{n} L(h_t, y_t)$ denotes the total sequence loss. This will then form the basis for the weight updates for the given training example. Now, as the model uses the same parameters to compute every output, we can identify the gradients by summing up the parameter contributions to the per-element losses:

$$\frac{\partial L_{seq}}{\partial W} = \sum_{t=1}^{n} \frac{\partial L(h_t, y_t)}{\partial W}$$

where W is one of W_x, W_h or b. However, another important property of RNNs is that any given element in the output is a function of the previous outputs. Thus, the output at a given time step affects the output of all subsequent outputs. Consequently, to identify the per-element losses, we must calculate them backward in time, going from the final step to the first. In this manner, we can accumulate the loss with respect to each parameter to identify the total gradients used in the weight update. This process is called *back-propagation through time.* We omit the details of the actual gradient calculations for brevity, but they depend on the concept of total derivative and the chain rule. For a complete derivation, see for instance Williams and Zipser [1995] or Kaelbling [2019].

The process described above enables RNNs to process sequential data, extract information and learn to use this information to solve the given learning task. In theory, this structure allows the network to associate information from earlier time steps and earlier input sequences with the current time step. This concept is compelling because the model can theoretically learn temporal dependencies, something a conventional stateless neural network cannot. However, in practice, a simple RNN, as described above, struggles to connect information between earlier time steps and the current time step when the gap grows [Olah, 2015]. That is, the model is unable to capture long-term dependencies. For instance, consider a language modeling task where the RNN is trained to predict the next word in a sentence. If the input sequence is “the fish is in the”, it is easy for a human to figure out that the next word should be “sea” or similar nouns. Here, the gap between the necessary information and the step where it should be recalled is small, and RNNs succeed at learning the connection and solving the task. Meanwhile, consider a different example where the task-critical information comes much earlier in the input sequence: “Yesterday I went for a run, and ... you could say I really love” Here, it is clear that the next word should be a something the subject enjoys, but to determine that the answer is “running”, we would need to remember back to the earlier content, which could be far in the past. When this gap increases, the RNN becomes unable to learn the information dependency and solve the task.

The problem with long-term dependencies for RNNs arises from the nature of back-propagation through time. Recall that to compute the gradients with respect to the loss of the outputs at early time steps in the RNN, we have to work our way back from the final time step. Additionally, recall that the outputs at a given time step depend on the outputs from earlier steps. Consequently, in order to compute the gradient at, for instance, step n, we must multiply the per-element loss at step n with the derivatives of each hidden state with respect to the previous hidden state $\frac{\partial h_t}{\partial h_{t-1}}$, for t from 2 to n [Kaelbling, 2019]. This product will either explode or shrink as the length of the sequence increases. All weight updates therefore either consistently increase or decrease the parameters such that their values explode or vanish. Consequently, the model struggles to learn as the weights are not appropriately updated. Thus, the RNN struggles to capture long-term dependencies during training.

The issue outlined above is addressed by the concept of *gate mechanisms*. This thesis studies two RNN architectures using gate mechanisms: The Long Short Term Memory (LSTM) network and the Gated Recurrent Unit (GRU). When describing these architectures, it is beneficial to think of the RNN structure as a chain of neural network layer modules (as A is represented in Figure 3.11). In the standard RNN case, the module or “cell” has a simple composition, such as a single tanh layer, as in the vanilla RNN studied in this thesis. This structure is visualized in Figure 3.12a. We can conceptualize RNNs as a chain of such cells, where the cell contains all neural network layers used to compute the output and the next hidden state. The cell structures of LSTMs and GRUs are also illustrated in Figure 3.12, and their components are described below.

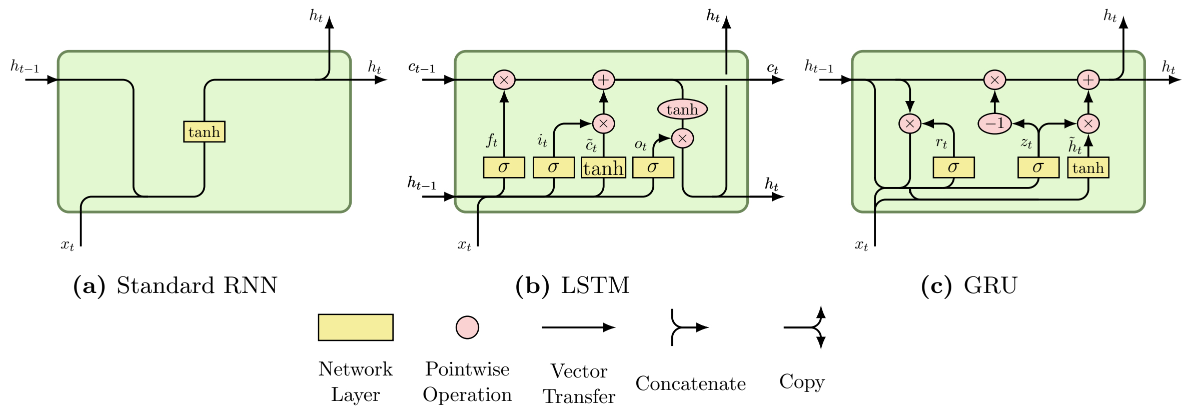


Figure 3.12: Visualization of standard RNN, LSTM and GRU cells. As described by the legend, the yellow rectangles represent neural network layers, the pink circles are pointwise operations (e.g. vector addition or multiplication), the lines indicate the transportation of vectors, and two lines merging represents a vector concatenation while two lines splitting specify the given vector being copied and transported in both directions. Figure inspired by Olah [2015].

Long Short Term Memory networks
The Long Short Term Memory (LSTM) network is an RNN based on gate mechanisms that was designed to avoid the problem of long-term dependencies. Since their introduction by Hochreiter and Schmidhuber [1997], they have seen considerable success and are capable of learning long-term dependencies. The key to this success lies in the structure of the LSTM cell, which is illustrated in Figure 3.12b. Note that numerous variations of the LSTM cell

exist and that this explanation is based on the variant described in Olah [2015]. While cell compositions differ, the core concepts remain the same. Compared with the standard RNN cell with a single tanh layer, the LSTM cell contains four layers interacting in a specific manner to administrate the flow of information in and out of the cell. When following the explanation of the LSTM cell below, we recommend referring to Figure 3.12b for a visual reference.

In the LSTM cell, information is stored in the cell state c_t. The cell state is first passed from the previous time step. Then, old information is removed from the cell state, and new information is added before it is propagated to the next step. Additionally, it is used to compute the output (hidden state) at step t, h_t. Gate mechanisms are used to remove and add information to the cell state. A gate consists of a sigmoid layer and a pointwise multiplication. Recall from Section 3.2 that the sigmoid function maps the input to the interval $[0, 1]$. Therefore, the sigmoid layer controls how much information should "pass through" the gate. When the sigmoid output is pointwise multiplied with the associated vector, low sigmoid values will let less information through and vice versa. In this manner, the flow of information along the cell state may be controlled. The LSTM uses three such gates: a *forget gate*, an *input gate* and an *output gate*.

The first gate the cell state passes is the forget gate. This gate considers the previous output h_{t-1} and the current input x_t to determine how much information in the previous cell state c_{t-1} to forget. Based on this, an output vector f_t is computed, containing values between 0 and 1 for each component of the cell state. The forget values are computed as

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where $[\cdot, \cdot]$ represents row-wise vector concatenation. When multiplied with c_{t-1}, components of f_t close to 0 remove a majority of the information in the corresponding component in the cell state, thus "forgetting" it. Conversely, values in f_t close to 1 hardly affect the cell state. Thus, if the previous hidden state and the current input make the LSTM decide that some information is not relevant anymore and should be replaced, the forget gate allows it to forget the currently stored information.

Next, new information is added to the cell state. Two neural network layers are used to accomplish this. First, the input gate computes i_t, specifying to which extent components of the cell state should be updated. Second, a tanh layer produces a vector of candidate values $\tilde{c}_t$ that could be added to the current cell state. Recall from Section 3.2 that tanh outputs values between -1 and 1. Then, the two vectors are multiplied, producing the values that should be added to the cell state. Thus, which components to update and how to update them is learned separately. The two vectors are computed as

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{c}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c). \end{aligned}$$

Thus, in this step, the network looks at the current input and the previous hidden state to decide whether to store new information. Consequently, if the network decides that information should be replaced, this step can ensure that new information is added to the cell state after the old is forgotten.

After the network has decided upon the information to forget and add, the actual cell state update can be performed. Given the old cell state c_{t-1}, the forget vector f_t, the input vector i_t and the candidate values $\tilde{c}_t$, the new cell state c_t is computed as

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t.$$

Thus, by multiplying the old state by f_t, the information to be removed is forgotten, and by adding $i_t * \tilde{c}_t$, the new candidate values are added, scaled based on how much each component of the cell state should be updated. Consequently, this step realizes the replacement of old information with new information based on x_t and h_{t-1}.

Finally, the output for the current time step h_t is decided based on the updated cell state c_t, the current input x_t and the previous output h_{t-1} in two steps. First, the output gate computes an output vector o_t that controls the magnitude of the final output for each component of the cell state. Second, the cell state is passed through a tanh function to scale it between -1 and 1. h_t is then produced by multiplying the two components together:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(c_t).$$

Hence, the LSTM controls which components of the re-scaled cell state should be output by multiplying by the output vector o_t. The output gate can then learn relationships between the previous output, the current input, and what values to output. This allows the network to condition the output on the context of past inputs encoded in the cell state and the most recent input.

By introducing the gate mechanism described above to control the information flow, LSTM networks can successfully capture long-term dependencies. Consequently, they outperform standard RNNs on most sequence modeling tasks, especially as the length between information dependencies increases.

Gated Recurrent Units
A variation of the LSTM cell that has gained recognition is the Gated Recurrent Unit (GRU), introduced in Cho et al. [2014]. The GRU cell can be considered a simplification of the LSTM cell. First, the cell state and hidden state (output) are combined, denoted h_t. Second, the input and forget gates are merged into a single gate called the *update gate.* Consequently, the GRU cell contains only three neural network layers and two gates. The second gate is called the *reset gate* which impacts the generation of new hidden state candidate values. Finally, the GRU cell contains no output gate. See Figure 3.12c for a visualization of the cell.

Given an input x_t and the previous state h_{t-1} the GRU cell computes a reset vector r_t in a similar manner to that of f_t in LSTMs:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

The reset vector is then multiplied by h_{t-1} to determine to what extent each component of the previous hidden state should contribute to the production of candidate values in the new state. Alternatively, as Cho et al. [2014] describe it, how much "the previous hidden state is ignored". This product is then used to compute candidate values for the new hidden state $\tilde{h}_t$:

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b_h)$$

Next, the update vector z_t is computed in a similar fashion to i_t in LSTMs:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

Finally, the hidden state update is computed using the update vector z_t and the candidate values $\tilde{h}_t$:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

The state update step illustrates how the input and forget gates are combined into the update gate through z_t. The components of the hidden state that the network decided to assign high values in z_t forget a considerable part of the previous hidden state and add more information from the candidate values. Meanwhile, components with low values in z_t correspond to less information being replaced by new information.

3.6.3 Temporal convolutional networks

Temporal convolutional neural networks are another neural network variant with recent success that processes sequential data. They are essentially a particular configuration of convolutional neural networks (CNNs). Consequently, we start with a brief introduction to CNNs. Note that this introduction is limited to the concepts regarding CNNs required to provide sufficient theoretical background to understand temporal CNNs. Thus, we refer the interested reader to, for instance, Goodfellow et al. [2016] for a more comprehensive review of CNNs. Moreover, the following introduction is inspired by said source.

Convolutional neural networks
At its core, a CNN [LeCun et al., 1989] is a neural network that processes grid-like data [Goodfellow et al., 2016]. Examples of grid-like data include temporal or spatial signals. For example, time series are temporal signals that can be considered grid-like, where values in the series correspond to elements in a one-dimensional array or grid. Images are examples of a spatial signal, where each pixel value corresponds to an element in a two-dimensional grid. By applying a *convolution operator*, CNNs can learn to identify patterns and extract features from such data more efficiently compared with a feedforward network. We begin by describing this operation before detailing the core advantages it implies for learning tasks involving grid-like data.

For practical purposes, the application of the convolutions in CNNs can be described as "sliding" a *kernel* or *filter* over the grid-like data to compute an output. The kernel is an array, matrix, or tensor (multi-dimensional matrix) of parameter values depending on the dimension of the data. It is applied to a given input element by aligning the center element of the kernel over this input element and computing the dot-product or matrix-product over the overlapping elements of the kernel and input data. The convolution operation involves applying the kernel to a single input data element. Then, a convolutional layer in a CNN performs this operation across the entire input data. The kernel is used to identify patterns or extract features in the input data and can be of any size, typically considerably smaller than the input. An illustration of the convolution operation is shown in Figure 3.13.

Formally, the convolution operation is a linear mathematical operation on two real-valued functions [Goodfellow et al., 2016]. The convolution is defined as

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da,$$

where $s(t)$ is the resulting convolution output when applied at real value t, x is a real-valued function mapping to input elements and w denotes the kernel. For discrete input data, the discrete convolution is defined as

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

The discrete definition can be rewritten to illustrate its connections to the practical explanation of convolutions above. Consider a multi-dimensional input, such as a 2-dimensional

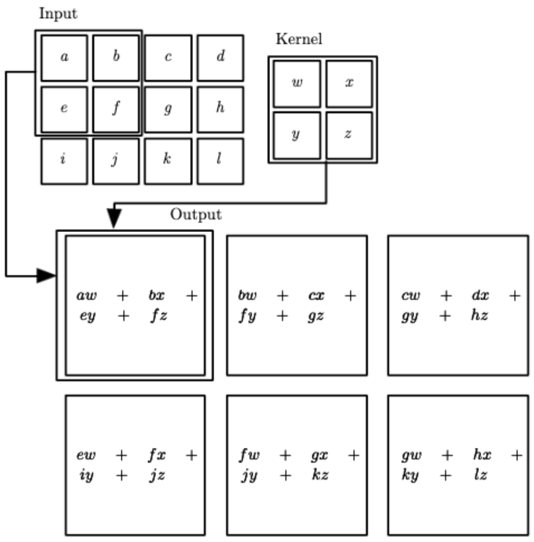


Figure 3.13: Illustration of the convolution operation. As the kernel slides over the data, the product between overlapping elements in the kernel and data is computed. Figure from Goodfellow et al. [2016].

image I and a 2-dimensional kernel K. If we assume that all input values to I and K that do not map to pixel values or kernel parameters map to 0, we can rewrite the discrete convolution to a finite sum over the two dimensions. Thus, the output of applying the convolution operation to the element (i, j) of the image can be written as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Hence, the output is the product of overlapping values between the kernel and the image, where the kernel is centered over the pixel in location (i, j).

Padding is commonly applied to the input data to enable convolutions on the edges of the input. Padding simply refers to appending a sufficient number of elements to the edges of the input such that it is possible to apply the filter on the original edges. The padded values are commonly set to zero to not affect outputs, but other values are possible. Note that padding allows the output of the convolution operation to be of the same shape as the input.

Convolution layers can also be applied using *stride*. A convolution layer with stride simply skips a specified number of input elements when applying the kernel to the next element. Using stride effectively reduces the output size. Meanwhile, stride may have the effect of unevenly distributing the contribution of each input element to the output of the layer.

The output of a convolution layer is commonly coupled with non-linear activation functions to ensure the network can model non-linear relationships. Furthermore, other operations such as *pooling* are commonly applied after convolution layers to manipulate the output. Pooling layers typically serve to summarize the input to the pooling layer in some way. The summary can for instance be aggregations such as *max pooling* or *average pooling*. See Goodfellow et al. [2016] for more details.

The kernel parameter values dictate the outputs of a CNN. For instance, to solve an object classification task, it would be preferable for a network to learn to detect patterns and features relating to the different classes. By carefully selecting kernel parameter values, the kernel can be designed to detect different features. See Figure 3.14 for an example.

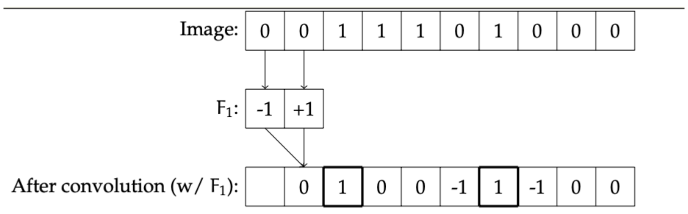


Image: 0 0 1 1 1 0 1 0 0 0
F1: -1 +1
After convolution (w/ F1): 0 1 0 0 -1 1 -1 0 0

Figure 3.14: An illustration of applying a 1D convolution. Given a binary input sequence, by carefully selecting the values of the kernel, it becomes a detector of a change from 0 to 1 (indicated by 1 in the output), which could, for instance, have the interpretation of left edge detection. Figure from Kaelbling [2019].

When applying several different kernels to the input with carefully selected parameter values and stacking several such convolution layers, CNNs can become powerful feature extractors. Naturally, hand-tuning each kernel weight quickly becomes infeasible with increasing numbers of parameters. Instead, parameters are commonly tuned through supervised learning using labeled data, a loss function, and gradient descent with back-propagation as described in Section 3.3.

In particular, CNNs perform well on learning tasks involving grid-like data due to several key advantages. First, if one were to process a 2D image in a fully connected network, one would require weights associated with each pair of pixel values and units in the next layer. With increasing input sizes, the number of weights required explodes. A CNN solves this problem through *parameter sharing*, where the weights are reused. Thus, a CNN requires considerably fewer weights than FCNs to perform equivalent operations. The second advantage relates to the *spatiality* of information. A CNN can extract meaning from spatial relations between input elements by design. For example, a CNN can quickly learn to account for relations between pixels close to each other using kernels, something an FCN would struggle to accomplish. The third advantage relates to the *locality* of information in the input. A CNN can learn to detect patterns regardless of where they appear in the topology of the input. For instance, a CNN can learn to identify a human nose using filters, regardless of its location in an image. These key advantages motivate the design of the CNN structure and explain its success in learning tasks related to grid-like data.

Temporal CNNs
Temporal convolutional neural networks (TCNs) are a specific variant of CNNs designed to process temporal data. The following explanation of TCNs is inspired by Bai et al. [2018] and focuses on one-dimensional inputs, as this is the relevant use case for this thesis.

A TCN is a CNN that satisfies two fundamental properties: given an input sequence, the TCN outputs a sequence of equal length, and when applying convolutions to a given time point, no information from later time points is considered [Bai et al., 2018]. In other words, no information from the future is allowed to "leak" into the past. The first property can be ensured using a fully-connected 1D CNN with zero-valued padding of $k - 1$ elements in each layer, where k is the kernel width. The number of output elements in each layer then remains constant. The second property is enabled by *causal convolutions*. Causal convolutions ensure that convolutions applied on a given element in a given layer only considers elements corresponding to past time points. In practice, this can be accomplished by simply applying the $k - 1$ padding to the left side of each layer, assuming the past direction is to the left. Thus, when applying the convolutional kernel to any given element in the layer, the output value is associated with the time point of the rightmost element overlapping with the kernel. All elements included in the convolution represent the past relative to this rightmost element. Thus, when sliding the kernel over elements in the layer, the rightmost output element considers the k last elements of the input. The bottom layer in Figure 3.15 illustrates this.

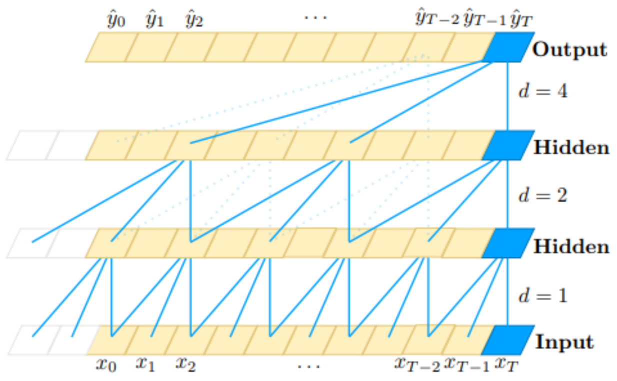

$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{T-2}$ $\hat{y}_{T-1}$ $\hat{y}_T$
Output
$d = 4$
Hidden
$d = 2$
Hidden
$d = 1$
Input
x_0 x_1 x_2 ... x_{T-2} x_{T-1} x_T

Figure 3.15: Illustration of a TCN with three layers of dilated causal convolutions. The convolutions are causal as the output is computed without future time points. Furthermore, a dilation scheme of $d = \mathcal{O}(2^i)$ is used, where the dilation factor doubles for each layer, thus doubling the size of gaps between elements involved in convolutions. Figure from Bai et al. [2018].

By stacking several causal convolution layers, the *receptive field* increases. The receptive field is the effective length of input history considered in the network outputs. By inputting a sequence to a causal convolution layer, then applying additional causal convolution layers on top of the previous, the effective number of elements from the input sequence considered by each output unit increases linearly with the number of layers [Bai et al., 2018]. However, by applying *dilated* causal convolutions [Oord et al., 2016], the receptive field can be increased exponentially with the number of layers. A dilated convolution with dilation factor d applies the convolution filter over the input, but with each kernel component spread out with "gaps" of length $d-1$. In other words, instead of applying the filter to k consecutive input elements, the filter is applied to k input elements where $d-1$ consecutive elements are skipped for each element included in the calculation. Formally, given a one-dimensional input sequence $x \in \mathrm{R}^n$, a k-sized kernel $f : \{0, 1, \ldots, k-1\} \to \mathrm{R}$, a dilated convolution F with dilation factor d on element s of the sequence is defined as

$$F(s) = (x *_d f)(s) = \sum_{i=0}^{k-1} f(i) x_{s-d \cdot i}.$$

Here, $s - d \cdot i$ where i ranges from 0 to $k-1$ correspond to the time steps of past elements in the sequence involved in the dilated convolution. By combining dilated and causal convolutions exponentially increasing receptive fields can be designed by increasing the dilation factor with each concurrent layer. For example, setting $d = \mathcal{O}(2^i)$ at layer i of the network is a common dilation scheme. See Figure 3.15 for a visualization of dilated convolutions.

The TCN architecture implies several benefits compared with RNNs regarding sequence modeling. For one, while RNNs must wait for the previous time steps to be calculated before the next, TCNs can compute outputs in parallel. This parallelism is enabled by the computational structure of the TCN using kernels. Furthermore, said structure allows gradients to be back-propagated through a considerably shorter path than in RNNs, as the path is not in the time direction. This allows for more stable differentiation and ensures TCN avoids the problem of vanishing or exploding gradients described in Section 3.6.2.

Furthermore, another advantage of TCNs is that the receptive field is highly flexible and can easily be adjusted by tuning the number of layers, kernel widths, and dilation rates to adapt to the input in different ways.

This thesis studies a TCN architecture heavily inspired by the TCN model proposed in Bai et al. [2018]. This model is constructed as explained above, with the additional introduction of *residual connections*. In essence, a residual connection allows neural network layers to learn modifications of the identity function instead of learning complete transformations. In a residual block, the input x is branched out, and a series of transformations $\mathcal{F}$ are performed before the output is calculated by adding the result of the transformations back to x:

$$o = Activation(x + \mathcal{F}(x))$$

This approach has proven beneficial for deeper neural networks due to the simplification of the transformations. Each layer in the proposed TCN is composed of a residual block containing two dilated causal convolution layers, as illustrated in Figure 3.16. More details can be found in [Bai et al., 2018].

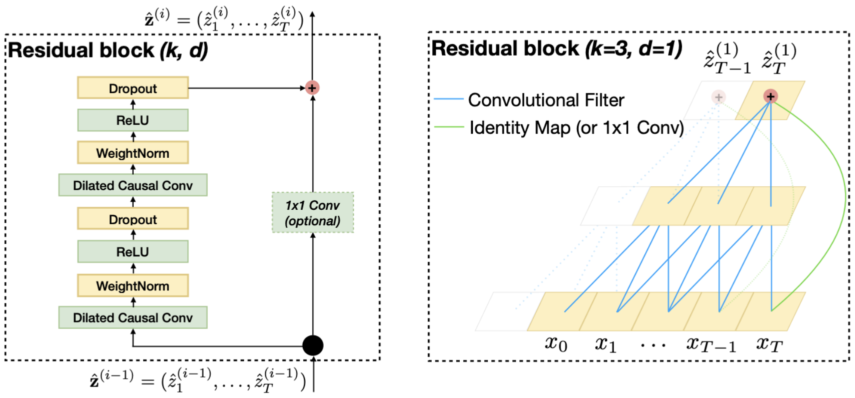

$\hat{\mathbf{z}}^{(i)} = (\hat{z}_1^{(i)}, \dots, \hat{z}_T^{(i)})$
Residual block (k, d)
Dropout
ReLU
WeightNorm
Dilated Causal Conv
Dropout
ReLU
WeightNorm
Dilated Causal Conv
1x1 Conv (optional)
$\hat{\mathbf{z}}^{(i-1)} = (\hat{z}_1^{(i-1)}, \dots, \hat{z}_T^{(i-1)})$
Residual block (k=3, d=1)
$\hat{z}_{T-1}^{(1)}$ $\hat{z}_T^{(1)}$
Convolutional Filter
Identity Map (or 1x1 Conv)
x_0 x_1 $\cdots$ x_{T-1} x_T

Figure 3.16: Illustration of the residual block in the TCN proposed in Bai et al. [2018]. The output from applying two dilated causal convolutions with the same kernel size k and dilation factor d is added back to the input. If the input and output are of different depths, an optional 1x1 1D convolution is applied before the addition to math dimensions. To avoid overfitting, dropout and weight normalization are performed after each convolution. ReLu is used to ensure non-linearity. Figure from Bai et al. [2018].

Chapter 4

Problem statement

Considerable time and effort were invested into defining the problem to explore in this thesis. In deciding what to study, several factors deemed important were identified. First, the thesis must contribute to Solution Seeker's research on improving their machine learning system. Consequently, they provided an outline for the thesis, specifying the work should extend upon our project report and their current research. Second, we wanted to explore a problem where sufficient data was available to ensure that we could evaluate our proposed approaches and make more time for other work besides collecting and processing data. This ensures our time is most efficiently spent aiding Solution Seeker in their research. Perhaps most importantly, we wanted to study a topic related to our research interests. Consequently, we decided early on that our thesis should consider flow rate estimation as it fits all the identified criteria. Furthermore, as flow rate estimation regards a dynamic system with time dependencies, we were eager to explore models that consider the time dimension of the problem. Moreover, we were inspired by the widespread success of sequence models in other similar problems (see Section 2.6). Consequently, we wanted to explore the application of such models in flow rate estimation. Thus, substantial effort was invested into evaluating the technical viability of different sequence models and approaches. Eventually, this elaborate process resulted in the following problem statement and corresponding research question that this thesis explores.

This chapter describes the problem studied in this thesis and the main research question explored. First, Section 4.1 presents the primary research question before Section 4.2 outlines formal details regarding the problem studied in the thesis.

4.1 Research question

Solution Seeker's current machine learning system relies on labeled data and only considers sensor measurements for the corresponding time period of the label (see Section 2.5). Furthermore, we have seen in Section 2.3 that obtaining labels (flow rate measurements) is costly. In contrast, the input sensor measurements are widely available (see Section 2.1). Solution Seeker has expressed the need for research on ways to overcome the challenges of limited data availability regarding flow rates, as described in Section 2.6. Furthermore, as illustrated in Section 2.4, flow rate estimation is inherently a dynamic problem considering a complex, dynamic system. Additionally, related research both regarding flow rate estimation and other machine learning literature has shown positive results when applying machine learning sequence models (see Section 2.6). Sequence models also enable the utilization of larger volumes of data as more (possibly unlabeled) information is included when the input is a sequence of past well state history. This motivates the following primary research question, which is the focus of this thesis:

Primary research question: *How does leveraging sequences of historical well state data affect the flow rate estimates of data-driven MTL-based virtual flow meters?*

The methods used to investigate the research question are presented in Chapter 5. Different sequential models (standard RNN, LSTM, GRU, and TCN) consider historical sequences of well state data in an MTL architecture to estimate flow rates. Then, different input sequence lengths and time horizons are compared to study how performance is affected by temporal granularity and history range. This methodology is designed to test our hypotheses relating to the research question. We hypothesize the following:

Hypotheses

1. Models trained on historical sequences of well state data will improve performance in estimating flow rates compared with steady-state models
2. Including well history from transient periods leading up to stable periods will improve performance on rate estimation in said stable periods
3. For a given input sequence length, models that consider shorter time horizons (higher temporal granularity) will estimate well flow rates more accurately than models considering longer horizons (lower granularity)
4. For a given input time horizon, models that consider a longer input sequence (higher temporal granularity) will estimate flow rates more accurately than models considering a shorter input sequence (lower granularity)

The hypotheses' primary motivation is that the underlying system dictating the flow rates is dynamic, and considering its development over time should allow better modeling of relationships between the system and output rates. Furthermore, the hypotheses stating a higher temporal granularity in the input sequence will yield higher performance are motivated by the high-frequent changes observed in the sensor measurements (see Chapter 6). We hypothesize that such high-frequent dynamics influence the output rates to a greater extent than slower, more long-term changes in the system. Furthermore, by providing models with sequences of higher granularity, they are given more information, which they can then determine how to best leverage to solve the problem.

By testing the hypotheses, we gain insights into the primary research question. To test the hypotheses, we formulate supplementary research questions. Based on these, we can design a methodology that, when followed, allows us to answer the supplementary research questions and thus test our hypotheses. The strengthening or weakening of the hypotheses thus informs the primary research question. Consequently, the following supplementary research questions are formulated:

Research questions

1. How do models trained on historical sequences of well state data affect model performance in estimating flow rates compared with steady-state models?
2. How does the inclusion of well history from transient periods affect performance on flow estimation in stable periods?
3. How does the time horizon considered in input sequences of dynamic VFMs affect well flow rate estimation performance?
4. How does the temporal granularity of the input sequences of dynamic VFMs affect well flow rate estimation performance?

Hence, we emphasize that the primary purpose of the thesis is to investigate whether an extended representation of the production system that considers past well state history can be more informative than an instantaneous steady-state representation. The hypothesis is that such an extended representation more accurately describes the state of the production system and thus can be used to estimate the current flow rates more accurately. However, while more accurate estimates may lead to more informed operational decision-making and increased revenues, this is not necessarily the case and is not the focus of the thesis. Hence, the focus is not to maximize production output or revenues but rather to improve the quality and availability of the flow rate information that is used in such optimization problems.

4.2 Formal problem details

The problem studied in this thesis is based on the flow modeling problem outlined in Section 2.4. Formally, we may describe the problem as follows.
Given:

- x_{jt}*, the raw sensor measurements for well* j *at time* t,
- SP_{ij}*, the set of uniformly distributed time points of same frequency as* x_{jt} *corresponding to the stable period* i *for well* j,
- Q_{jt}*, the measured total flow rate measured for well* j *at time* t,

identify a model that uses measurements x_{jt} *to produce an estimate of flow rate measurements* Q_{jt} *during stable period* i *as defined by* SP_{ij}*, with minimal estimation error.* By comparing the performance of models considering different representations the measurements x_{jt}, we may then answer the primary research question.

The system of interest considered in this problem is the offshore oil and gas production system described in Section 2.1. More specifically, the problem considers the choke valve of the given well due to its relevance to the flow rates and the availability of related sensor measurements. Moreover, relevant literature this thesis builds upon, such as Sandnes et al. [2021], Løvland [2021] and Heggland and Kjærran [2021] follows the same approach. This system, with related measurements involved in the problem, is specified in Figure 4.1.

Hence, we define the raw sensor measurements for well j at time t x_{jt} as:

$$x_{jt} = [p_{jt,1}, T_{jt,1}, u_{jt}, p_{jt,2}, q_{jt}^{GL}, \phi_{jt,O}, \phi_{jt,G}]^T$$

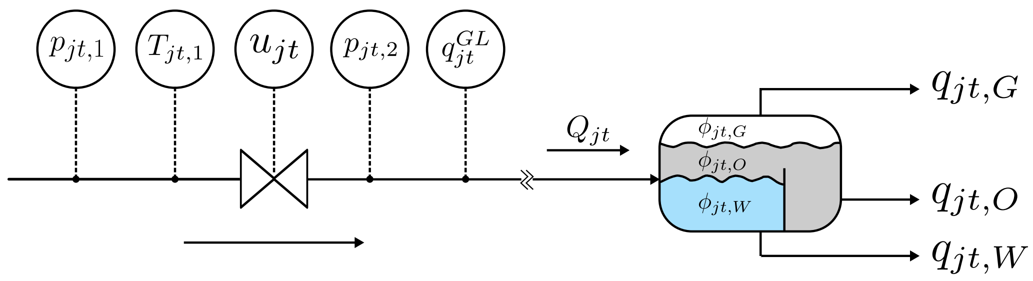


Figure 4.1: Illustration of the system of interest with relevant measurements. All indices represent the measurement at a point in time t for well j. $p_{jt,1}$ and $T_{jt,1}$ are the pressure and temperature upstream the choke valve, respectively. Likewise, $p_{jt,2}$ is the pressure downstream the choke valve. u_{jt} is the choke valve opening percentage. q_{jt}^{GL} is the measured injected gas lift. Where available from MPFMs or well tests, $q_{jt,O}$, $q_{jt,G}$ and $q_{jt,W}$ represent the oil, gas, and water flow rates, respectively. Likewise, $\phi_{jt,O}$, $\phi_{jt,W}$ and $\phi_{jt,G}$ represent the phase fractions for the given point in time and well. Finally, Q_{jt} is the total multiphase flow rate at time point t in well j.

The inputs considered in the proposed models are created from measurements x_{jt} and represent either a vector of mean values of sensor measurements recorded during a given stable period, a sequence of such vectors for subsequent stable periods leading up to the stable period to estimate for, or a uniformly resampled sequence of higher resolution aggregations of x_{jt} leading up to and including the period. The alternatives are used in different models to compare estimation performance when the operational history leading up to the stable period is described in varied manners. The different representations are described in detail in Section 5.1.

Stable periods are defined as time periods during production where all control variables (gas lift and choke valve) remain constant and sufficient time has passed since the last control variable changed. Furthermore, longer periods that satisfy these conditions are divided into shorter stable periods to account for long-term changes in the system. Thus, the well's default state is to operate under stable conditions by this definition. When control variables are changed, the system becomes transient and eventually stabilizes in a new stable state.

Hence, the thesis focuses on estimating flow rates during stable periods. This allows us to compare steady-state models to dynamic models directly. While the inputs vary, the target output, and thus the loss function, is the same. Furthermore, as the system is generally under stable conditions during operations (as defined above), flow rate estimates during stable periods may inform operational decisions to a greater extent than estimates during transient periods.

We could have modeled a more comprehensive part of the system. For instance, the model could be extended to consider the production system from the reservoir up to and including the choke valve. This can be accomplished by including bottomhole temperature and pressure measurements. Such a model would give a more detailed overview of the current state of the well as more information is included. However, according to Solution Seeker, available sensor measurements at the bottomhole are commonly unavailable, limiting the applicability of such an approach. Taking this into account and considering the arguments in the previous paragraph, we limit the thesis to model the choke valve.

Chapter 5

Method

In this chapter, we detail the methods used to test the hypotheses and answer the research question presented in Chapter 4. First, Section 5.1 gives a high-level overview of the approaches used. Next, the state-of-the-art model architectures are described in detail in Section 5.2. Finally, the process of training and identifying best-performing models is explained in Section 5.3. Where the methodology sufficiently overlaps, explanations include revised passages from our project report [Heggland and Kjærran, 2021].

5.1 Overall approach

Recall the outline of the relevant problem described in Section 4.2. The following approach is followed to examine whether flow rate estimation performance improves by modeling the relationship between well state history over time and flow rates. Three classes of models are developed, classified by the nature of the input data $x_j^{(i)}$, where $x_j^{(i)}$ is a representation of the operational state of well j associated with the stable period i:

1. **Steady-state baseline**: $x_j^{(i)}$ is a vector of means of the measured values during steady-state period i for well j
2. **Steady-state mean sequence models**: $x_j^{(i)}$ is a sequence of T steady-state mean vectors, from consecutive periods up to and including period i for well j
3. **Dynamic sequence models**: $x_j^{(i)}$ is a sequence of T vectors of consecutive uniformly re-sampled measurements such that the sequence history ranges D days. The final entry corresponds to the final time point of stable period i for well j

Hence, each model category represents the well's history to different extents and varying forms. See Figure 5.1 for an illustration. All models are designed to model the relationship between the given input representation and $Q_j^{(i)}$, the corresponding mean total flow rate measured during the respective stable period for the given well. That is, given an input representation $x_j^{(i)}$, the model should output an estimate $\hat{Q}_j^{(i)}$. This approach allows us to study how the different representations of the well's state affect flow rate performance, thus enlightening the research question.

First, the steady-state baseline model serves as a benchmark. The performance of sequence models can be compared with the baseline's performance to determine their capabilities.

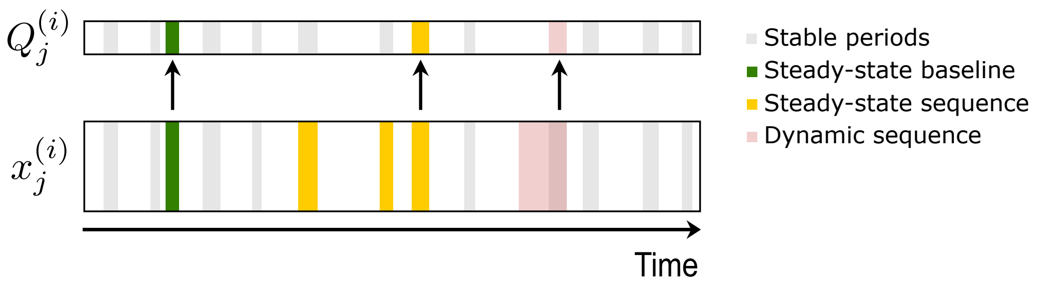


Figure 5.1: Illustration of the different well history representations $x_j^{(i)}$ and their relationship to flow rates $Q_j^{(i)}$ during corresponding stable periods.

Next, the steady-state mean sequence models represent a simple extension of the input representation, where only data from stable periods are considered. Hence, the input sequence is not uniformly distributed in time. These models serve to demonstrate whether more complex sequence models are redundant. Different sequence lengths T are tested to gain a broader understanding of their viability. Finally, the dynamic sequence models examine the performance of state-of-the-art sequence models, using a uniformly resampled input sequence of both transient and stable measurements to estimate the flow rate. We study the performance of such models using different sequence lengths T and time horizons D. The quality of the models' output estimates allows us to investigate the effect of leveraging the well history in different ways.

We now introduce notation used to formally describe the overall model architecture. Recall the relevant system measurements presented in Section 4.2, and the raw sensor measurements for well j at time t that we defined as

$$x_{jt} = [p_{jt,1}, T_{jt,1}, u_{jt}, p_{jt,2}, q_{jt}^{GL}, \phi_{jt,O}, \phi_{jt,G}]^T$$

x_{jt} represents the measurements re-sampled uniformly at the frequency of the sensor measuring most infrequently. Furthermore, let SP_{ij} be the set of uniformly distributed time points of same frequency as x_{jt} corresponding to the stable period i for well j. See Section 4.1 for the used definition of stable periods. The steady-state mean vector $\bar{x}_j^{(i)}$ used in the first two model classes is then the mean over measured data in SP_{ij}:

$$\bar{x}_j^{(i)} = \frac{1}{|SP_{ij}|} \sum_{t \in SP_{ij}} x_{jt}$$

Likewise, the labels $Q_j^{(i)}$ are given by the equivalent calulation over the measured total flow rates:

$$Q_j^{(i)} = \frac{1}{|SP_{ij}|} \sum_{t \in SP_{ij}} Q_{jt}$$

Finally, the dynamic sequence model input sequences are re-sampled to given frequencies such that T time points span D days of history. Thus, given values for T and D, the raw measurements are aggregated such that there is one measurement every $\frac{D \cdot 24 \cdot 60}{T}$ seconds. If $\mathcal{T}_t$ is the set of time points uniformly distributed at the same frequency as x_{jt} spanning one such interval ending at time t, the corresponding re-sampled measurement vector is given by

$$x_{jt}^{DT} = \frac{1}{|\mathcal{T}_t|} \sum_{\tau \in \mathcal{T}_t} x_{j\tau}$$

This resampling gives a new data set for each choice of pairs of T and D. Furthermore, the resampling is performed such that the final element of each sequence corresponds to the final time point of the corresponding stable period. This approach ensures that each generated sequence overlaps with the stable period and avoids the generation of identical input sequences for stable periods close in time.

Thus, the input representation is given by

$$x_j^{(i)} = \begin{cases} \bar{x}_j^{(i)} & \text{Steady-state baseline} \\ \bar{x}_j^{(i-T+1)}, \ldots, \bar{x}_j^{(i-1)}, \bar{x}_j^{(i)} & \text{Steady-state mean sequence models} \\ x_{j,(t-T+1)}^{DT}, \ldots, x_{j,(t-1)}^{DT}, x_{jt}^{DT} & \text{Dynamic sequence models} \end{cases}$$

This allows us to describe the overall model architecture: First, each model uses an embedding layer to produce an encoding w_j of the well number j for a given input representation $x_j^{(i)}$. Hence, w_j is a vector of trainable parameters, and we choose a four-dimensional embedding. Consequently, we may write $w_j = \mathit{Embedding}(j)$. Thus, given an input representation $x_j^{(i)}$, a well embedding vector w_j is generated, and the two components are input to a model f. The model then produces a flow rate estimate $\hat{Q}_j^{(i)}$ based on the input information. That is, $f(x_j^{(i)}, j) = \hat{Q}_j^{(i)}$. w_j serves to inform the model of which well the input originates from, such that the model may learn to adapt to each well, and learn across wells using MTL. This architecture is illustrated in Figure 5.2.

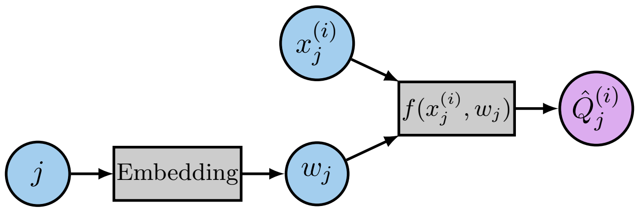


Figure 5.2: High-level illustration of the proposed model architecture. First, a well-specific indicator w_j is generated by an embedding layer. Then, a representation $x_j^{(i)}$ of the well state history and w_j generates an estimate of the flow rate $\hat{Q}_j^{(i)}$ using the flow rate estimation model f.

The attentive reader may question the decision to include $\phi_{jt,O}$ and $\phi_{jt,G}$ in the state vector x_{jt} as these variables represent the percentage of the total flow consisting of oil and gas, respectively. This information is naturally not present in unlabeled data. Furthermore, in the labeled data, fractions can be derived from labels, leading to “leakage” if used in inputs. However, the fractions are assumed to remain somewhat stable over time and are determined by the measurements of the latest well test performed on the given well. Furthermore, the water fraction $\phi_{ij,W}$ is omitted as it is given by $1-\phi_{ij,O}-\phi_{ij,G}$, leaving it redundant. Additionally, it may not be immediately intuitive how estimating the total flow $Q_j^{(i)}$ also estimates the individual oil, gas, and water flow rates. This can be explained using the same assumption: each phase flow rate can be calculated using the assumed fraction. That is, for instance $q_{j,O}^{(i)} = Q_j^{(i)} \phi_{j,O}^{(i)}$ where all quantities represent corresponding means over the stable period. Consequently, estimating the total flow rate instead of the three individual rates enables us to simplify the complexity of the model.

The use of state-of-the-art sequence models in this approach is highly motivated by the compelling benefits of modeling temporal dependencies in the well system. Furthermore, the utilization of unlabeled data in the sequences is another compelling factor. As seen in Section 2.6, such approaches have proven highly successful in related literature both within flow modeling and other fields. Additionally, as discussed in Section 2.6, the approach is motivated by the preceding works of Solution Seeker, highlighting the need for future research within methods utilizing the available data to a greater extent.

5.2 Model architectures

The following subsections provide a detailed description of the architectures of the three model categories presented in Section 5.1.

5.2.1 Steady-state baseline

The steady-state baseline is inspired by Solution Seeker's current machine learning system, as described in Sandnes et al. [2021]. The model is composed of a feedforward fully connected neural network (see Section 3.2). The network inputs the row-wise vector concatenation of $\bar{x}_j^{(i)}$ and the well embedding w_j and outputs $\hat{Q}_j^{(i)}$. That is, given a steady-state baseline model f_B composed of a network NN defined by parameters W, flow rate estimates are computed as

$$\hat{Q}_j^{(i)} = f_B(\bar{x}_j^{(i)}, w_j) = NN([\bar{x}_j^{(i)}, w_j]; W).$$

Recall from Section 5.1 that $x_{jt} \in \mathbb{R}^7$ and thus $\bar{x}_j^{(i)} \in \mathbb{R}^7$. Given a four-dimensional well embedding such that $w_j \in \mathbb{R}^4$, the input layer of the network lies in $\mathbb{R}^{11}$. Next, the network contains a given number of hidden layers, each with the same number of hidden units. Both of these parameters are subject to a hyperparameter search, as described in Section 5.3. That is, we do not fixate the network structure but allow a hyperparameter search to sample from specified domains to identify an ideal depth and width. Finally, the last hidden layer maps to an output layer in $\mathbb{R}$, representing $\hat{Q}_j^{(i)}$. See Figure 5.3 for an illustration of the architecture.

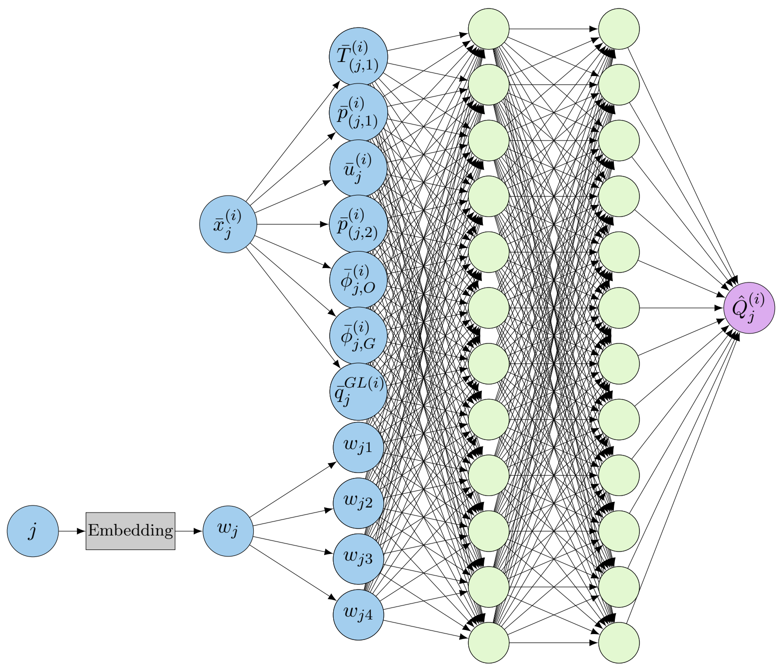


Figure 5.3: Illustration of the steady-state baseline model architecture. The number of hidden layers and units are adjustable, and two hidden layers with 12 units each are shown here for visualization.

All hidden units in the network use ReLU activation functions. Meanwhile, the output unit does not use an activation function. This is because the domain of activation functions in the output layer must coincide with possible values of the labels. We scale labels to values near zero and want to avoid imposing restrictions on the possible output values of the networks. Not using an activation function in the final layer accomplishes this.

The primary motivation behind the proposed model architecture is its simplicity and documented proficiency in flow rate estimation. In addition, it is a well-studied architecture that represents a reasonable steady-state VFM alternative and is the design applied in Solution Seeker's machine learning system. Thus, it is well-suited as the baseline model to compare dynamic VFMs with and examine the research question.

The decision to allow a hyperparameter search to determine the network structure is motivated by the fact that we have no prior knowledge of what high-performing transformations of the input look like. Meanwhile, we want to ensure that the neural network's capacity does not become a constraining factor. Therefore, the hyperparameter domains are designed to allow for as much flexibility in the networks as possible. By allowing the hyperparameter search to adjust the regularization of the network, a wide range of models with varying complexity can be identified and evaluated. For example, a deeper and wider network gives a model with high capacity and can be regularized to adjust the model complexity. Meanwhile, a more shallow and narrow model can obtain reasonable capacity with limited regularization.

5.2.2 Steady-state mean sequence models

The steady-state mean sequence models represent a simplistic extension of the steady-state baseline model to a sequential input. The architecture of these models differs from the baseline only in the input. Whereas the baseline inputs the steady-state mean vector corresponding to the stable period associated with the label $\bar{x}_j^{(i)}$, these models input a sequence of such vectors $\bar{x}_j^{(i-T+1)}, \dots, \bar{x}_j^{(i-1)}, \bar{x}_j^{(i)}$. See Figure 5.4 for an illustration of the architecture. We develop three such models, each with a different number of vectors in the sequence (different sequence lengths T). See Table 5.1.

The network inputs the row-wise vector concatenation of the T vectors in the sequence and the well embedding w_j and outputs $\hat{Q}_j^{(i)}$. That is, given a steady-state mean sequence model f_{SS} composed of a network NN defined by parameters W, flow rate estimates are computed as

$$\hat{Q}_j^{(i)} = f_{SS}(\{\bar{x}_j^{(i-T+1)}, \dots, \bar{x}_j^{(i-1)}, \bar{x}_j^{(i)}\}, w_j) = NN([\bar{x}_j^{(i-T+1)}, \dots, \bar{x}_j^{(i-1)}, \bar{x}_j^{(i)}, w_j]; W).$$

Hence, given a well embedding $w_j \in \mathbb{R}^4$ and a sequence length T, the input dimension of the network lies in $\mathbb{R}^{7T+4}$. Otherwise, the network structure is designed identically to that of the baseline model described in Section 5.2.1.

The motivation behind this model design is to investigate the performance of a somewhat naive extension of the steady-state baseline model. The intention is that these models may learn more complex relationships than a steady-state model by considering measurements from consecutive steady-state periods. They thus serve as a bridge between the baseline and dynamic sequence models in terms of complexity and information considered. This allows us to examine whether the more complex sequence models are justified or if simpler architectures may suffice.

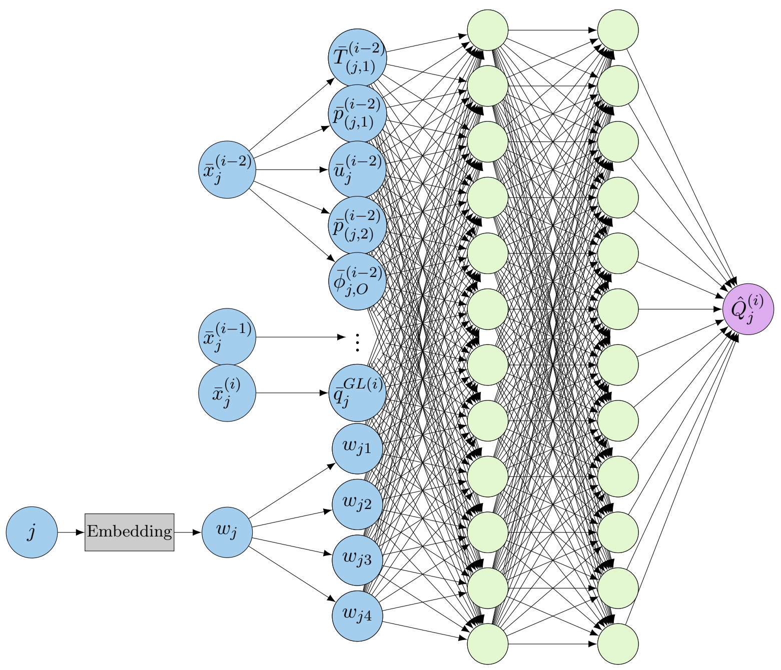


Figure 5.4: Illustration of the steady-state mean sequence model architecture. The number of hidden layers and units are adjustable, and two hidden layers with 12 units each are shown here for visualization.

5.2.3 Dynamic sequence models

The dynamic sequence models represent the most complex models studied, and the models considering the most extensive well history. Their main component is one of the sequence model architectures described in Section 3.6. That is, we test models using standard RNNs, LSTMs, GRUs, and temporal CNNs. For each architecture, we test nine pairs of time horizon D and sequence length T (see Table 5.1).

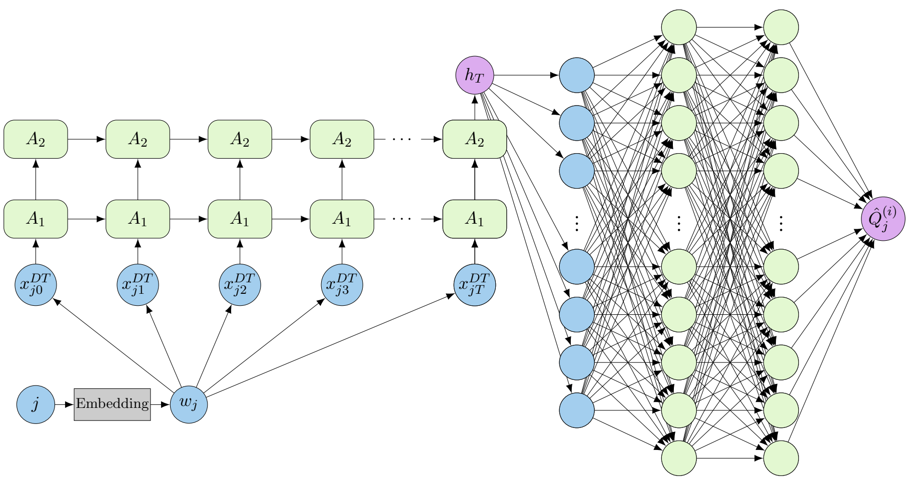
As described in Section 5.1, these models input a sequence of T measurements $x^{DT}_{j,(t-T+1)}$, $\dots, x^{DT}_{j,(t-1)}, x^{DT}_{jt}$ uniformly re-sampled such that each sequence spans D days. The sequences are processed by the sequence models together with w_j, before the output is passed through a fully connected layer to produce the final flow estimate. That is, given a dynamic sequence model f_{SEQ} composed of a network NN defined by parameters W, flow rate estimates are computed as

$$\begin{aligned}\hat{Q}^{(i)}_j &= f_{SEQ}(\{x^{DT}_{j,(t-T+1)}, \dots, x^{DT}_{j,(t-1)}, x^{DT}_{jt}\}, w_j) \\ &= NN([x^{DT}_{j,(t-T+1)}, w_j], \dots, [x^{DT}_{j,(t-1)}, w_j], [x^{DT}_{jt}, w_j]; W)\end{aligned}$$

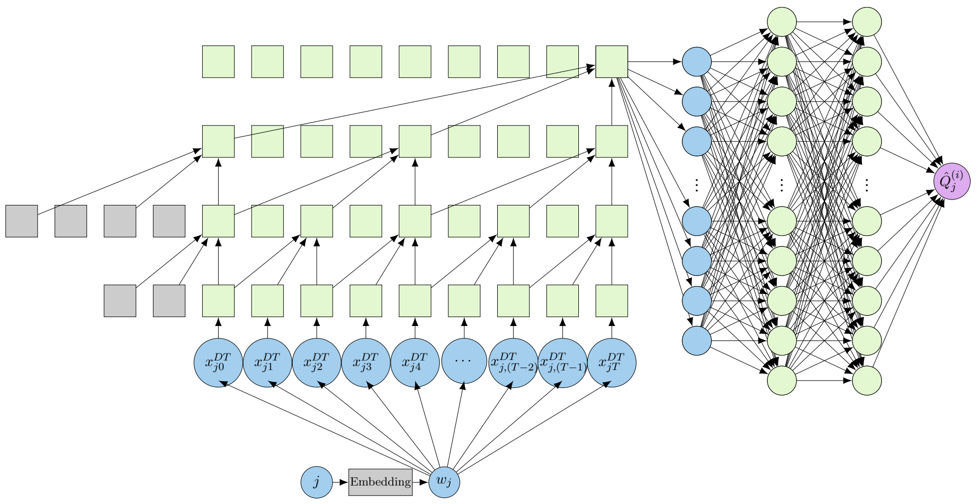
Here, the well embedding is repeated and concatenated to each sequence element before processing the sequence. See Figure 5.5a for an illustration of the architecture.

In the models using recurrent networks, the input elements are processed sequentially. The new computed cell states are passed to the next step and the subsequent recurrent layers for each step. Then, when the entire sequence is processed, the output of the final layer in the final time step is extracted and passed through an FCN. The number of RNN layers and dimensionality of the hidden state is subject to hyperparameter tuning. Likewise, the structure of the subsequent FCN is also adjustable in a hyperparameter search.

The models using TCNs function in a similar manner. Dilated causal convolutions are



(a) Recurrent architecture. The recurrent network can have anumber of layers, and is here visualized using two layers. The dimensions of the hidden state h_t



(b) Temporal CNN architecture. Here, the TCN is visualized with a kernel width k of 3 and 3 dilated causal convolution layers. The dilation scheme $d = \mathcal{O}(2^i)$ is used. The gray elements are padded values, and $d(k-1)$ elements are added to the left in each layer.

Figure 5.5: Illustration of the dynamic sequence model architectures. For illustration, the input elements are time indexed based on their position in the sequence.

applied to the input sequence. A specified number of multiple filters are applied in each layer, determining the depth of all subsequent next layers. The dilation scheme $d = \mathcal{O}(2^i)$ is used, as discussed in Section 3.6. The number of layers in the TCN is chosen such that the rightmost element of the output sequence is generated based on the entire input sequence. That is, we select the number of layers such that the receptive field is at least as large as the input sequence length. The required number of layers n to achieve this depends on the chosen kernel width k and sequence length T:

$$n = \left\lceil \log_2 \left(\frac{T-1}{2(k-1)} + 1 \right) \right\rceil$$

See for instance Lässig [2020] for a derivation of this result.

Finally, the rightmost element of the final layer of the TCN is extracted and passed through an FCN to generate the flow rate estimate, similar to the RNN architecture. The number of filters to use in every layer and the kernel width and the structure of the subsequent FCN are subject to hyperparameter tuning.

The proposed TCN structure is motivated by the factors discussed in Section 3.6. It is inspired by Bai et al. [2018] and only differs in the residual block. Due to experiencing gradient instabilities during initial testing, we omit the weight normalization layer in the residual blocks and regularize the network using weight decay and dropout.

5.3 Training process

The chosen model architectures allow for a range of different neural network configurations. To answer the research question, we identify high-performing models by training, evaluating, and comparing models with different network configurations. In doing so, we apply the following methodology. First, for each of the presented model architectures, we train networks using different values of the hyperparameters, constituting different network configurations. Then, the trained networks' performances are evaluated to investigate the proficiency of the given network structures. To find the best-performing neural network configuration for a given architecture, we use automatic hyperparameter tuning. Below, we describe the training process for a given network configuration before explaining how hyperparameter searches are conducted to search for the best-performing network configurations.

During the training of a given network configuration, training examples $x_j^{(i)}$ from the training set are input into the network. This produces a flow estimate $\hat{Q}_j^{(i)}$. To induce learning, the quality of the network's flow rate estimates is evaluated using a loss function L, comparing the rate estimate to the actual rate measurement $Q_j^{(i)}$. We use Mean Squared Error (MSE) as our loss function. In addition, we use L_2 regularization by penalizing large weights in the network to avoid overfitting. Let W denote the weights of the network and λ the L_2 regularization factor. This gives the following loss calculation for a set of N training examples with labels Q producing estimates $\hat{Q}$:

$$L(Q,\hat{Q}) = \frac{1}{N}\sum_{i,j}(Q_j^{(i)} - \hat{Q}_j^{(i)})^2 + \lambda \sum_{w\in W} \|w\|_2^2$$

Upon calculating this loss, we update the weights of the network by calculating the gradient of the loss with respect to the model weights using back-propagation and gradient descent with the Adam optimizer as described in Section 3.3 and Section 3.6. Applying this optimization process, we iterate over the training set in batches, gradually improving the network weights with respect to the training loss. After iterating through the entire training set, we evaluate the network by calculating the validation loss. One such run through the training set and subsequent run through the validation set constitute one epoch. We then repeat this process for a given number of epochs. The training and validation loss is calculated and reported as the epochs progress. During this process, we use early stopping and save the configurations yielding the lowest validation losses (see Section 3.4).

The above explanation describes the process of training and evaluating a given neural network. To find the highest performing neural network configuration within a given model architecture, we perform a hyperparameter search. The search involves tuning the model-specific hyperparameters described in Section 5.2. Additionally, the learning rate of the Adam optimizer and the L_2 regularization factor λ are tuned for all models. We use the hyperparameter optimization framework presented in Akiba et al. [2019] to perform this hyperparameter search.

The hyperparameter search is performed as follows. The search consists of a specified number of trials. Each trial represents a complete training process for a given network configuration with continuous monitoring of the validation loss after each epoch. For each trial, a value is sampled for each hyperparameter within specified domains using corresponding sampling strategies. This gives a neural network configuration to be trained over a specified number of epochs. Based on the validation losses obtained during the training over the trials, the optimization algorithm directs the search towards hyperparameter values expected to improve the validation loss. Thus, as the trials progress, the search trains different neural networks and gradually identifies configurations with lower validation losses with no guarantee of optimality. In theory, the networks that obtain the lowest validation losses should be the most proficient at estimating flow rates.

A multivariate Tree-Parsen Estimator (TPE) sampler is used to guide the parameter search, which jointly considers the performance of previously sampled values across hyperparameter domains when suggesting new values. See Falkner et al. [2018] for details.

As described in Section 3.3, test sets are put aside for the duration of the training. To measure a model's ability to generalize, it is evaluated on the test sets once the model parameters are finalized. That is, we calculate the test loss for the network configuration, obtaining the lowest validation loss in a given search. The procedure followed to identify the final flow rate estimation models is outlined in Algorithm 2. Furthermore, see Table 5.1 for an overview of all the models that hyperparameter searches is performed on in the thesis.

Algorithm 2 Procedure followed to identify final flow rate estimation models

```
IDENTIFY-MODELS($\mathcal{D}$, n_trials, n_epochs)
    for each model architecture $m$ do
        Process and split dataset $\mathcal{D}$ into $\{\mathcal{D}^m_{train}, \mathcal{D}^m_{val}, \mathcal{D}^m_{test}\}$   ▷ Model-specific datasets
        Randomly initialize $\Theta^*_m$   ▷ Parameters of best model in search
        $L^*_m = \infty$   ▷ Lowest validation loss obtained in search
        for each trial $t = 1, \dots,$ n_trials do   ▷ One search iteration
            Sample values for hyperparameters using optimizer based
            on performance of earlier trials
            Randomly initialize parameters of network, $\Theta_{mt0}$
            $L^*_{mt} = \infty$   ▷ Lowest validation loss obtained in trial
            $\Theta^*_{mt} = \Theta_{mt0}$   ▷ Parameters of best model in trial
            for each training epoch $i = 1, \dots,$ n_epochs do
                Train network on $\mathcal{D}^m_{train}$ using Adam, producing $\Theta_{mti}$
                Calculate validation loss $L_{mti}$ using $\mathcal{D}^m_{val}$ and $\Theta_{mti}$
                if $L_{mti} < L^*_{mt}$ then   ▷ Record best model in trial
                    $L^*_{mt} = L_{mti}$
                    $\Theta^*_{mt} = \Theta_{mti}$
                else if validation loss early stopping criteria met then
                    Break for-loop   ▷ Start next trial
            if $L^*_{mt} < L^*_m$ then   ▷ Record best model in search
                $L^*_m = L^*_{mt}$
                $\Theta^*_m = \Theta^*_{mt}$
        Calculate test loss $L_{test,m}$ using $\mathcal{D}^m_{test}$ and final identified network $\Theta^*_m$
    Return $\{\Theta^*_m, L_{test,m}\}$   $\forall m$
```

Table 5.1: Overview of all models trained in the thesis.

Model category	Architecture	Time horizon	Sequence length	Hyperparameters
Steady-state baseline model	FCN baseline	-	1	FCN hidden layers & units, Adam learning rate, weight decay
Steady-state mean sequence models	FCN steady-state sequence	-	3, 10, 30	FCN hidden layers & units, Adam learning rate, weight decay
Dynamic sequence models	RNN	1 day	32, 64, 128	FCN hidden layers & units, hidden state size, recurrent layers, Adam learning rate, weight decay
		10 days	32, 64, 128	
		30 days	32, 64, 128	
	LSTM	1 day	32, 64, 128	FCN hidden layers & units, hidden state size, recurrent layers, Adam learning rate, weight decay
		10 days	32, 64, 128	
		30 days	32, 64, 128	
	GRU	1 day	32, 64, 128	FCN hidden layers & units, hidden state size, recurrent layers, Adam learning rate, weight decay
		10 days	32, 64, 128	
		30 days	32, 64, 128	
	TCN	1 day	32, 64, 128	FCN hidden layers & units, number of filters, kernel width, Adam learning rate, weight decay
		10 days	32, 64, 128	
		30 days	32, 64, 128	

Chapter 6

Dataset

To explore the research question, Solution Seeker has provided a dataset of the readily available measurements presented in Chapter 4. This chapter describes the provided data in Section 6.1. Then, Section 6.2 details the process of transforming the raw dataset to a format applicable to the methodology presented in Chapter 5. Finally, Section 6.3 discusses the properties of the processed dataset. The chapter contains some revised explanations from our project report [Heggland and Kjærran, 2021] as the provided dataset and required processing steps somewhat overlap.

In the figures and tables presented in this chapter and Chapter 7, abbreviations are used to refer to specific measures or features relevant to the rate estimation problem. Table 6.1 shows an overview of the abbreviations used for each feature.

Table 6.1: Mapping from abbreviations used in figures to corresponding features in the state representation. The last three features are inferred and are not present in the provided dataset.

Abbreviation	Symbol	Feature description
CHK	u_{jt}	Choke valve opening percentage
PWH	$p_{jt,1}$	Pressure upstream the choke valve
PDC	$p_{jt,2}$	Pressure downstream the choke valve
TWH	$T_{jt,1}$	Temperature upstream the choke valve
QOIL	$q_{jt,O}$	Oil flow rate (MPFM)
QWAT	$q_{jt,W}$	Water flow rate (MPFM)
QGAS	$q_{jt,G}$	Gas flow rate (MPFM)
QGL	q_{jt}^{GL}	Injected gas flow rate (gas lift)
FOIL	$\phi_{jt,O}$	Oil fraction from reservoir (generated)
FGAS	$\phi_{jt,G}$	Gas fraction from reservoir (generated)
QTOT	Q_{jt}	Total multi-phase flow rate (generated)

6.1 Dataset overview

The provided dataset contains sensor measurements and control variable readings from ten wells originating from a single field. The raw measurements span a three-year period from 01/01/2019 to 31/12/2021, with full data coverage for all wells. These measurements are based on high-frequency sensor readings, which are interpolated using spline interpolation and uniformly resampled to a 10-second frequency (0.1 Hz). All flow rate measurements are measured by dedicated multiphase flow meters (MPFM) installed in each well. The resampling gives a dataset of approx. 94.7 million records, of which ≈ 9.47 million originate from each of the ten wells. Each record consist of the first eight features listed in Table 6.1. See Table 6.2 for a summary of the provided dataset. Furthermore, an illustration of all sensor measurements for well 0 plotted over the available time frame can be seen in Figure 6.1.

Table 6.2: Overview of measurements in the provided dataset.

Feature	Description	Unit	Frequency	Measurements/well	Start date	End date
CHK	Choke opening	%	0.1 Hz	9 469 440	01/01/2019	31/12/2021
PWH	Upstream pressure	bar				
PDC	Downstream pressure	bar				
TWH	Upstream temperature	°C				
QOIL	Oil flow rate	Sm^3/h				
QWAT	Water flow rate	Sm^3/h				
QGAS	Gas flow rate	Sm^3/h				
QGL	Gas lift flow rate	Sm^3/h				

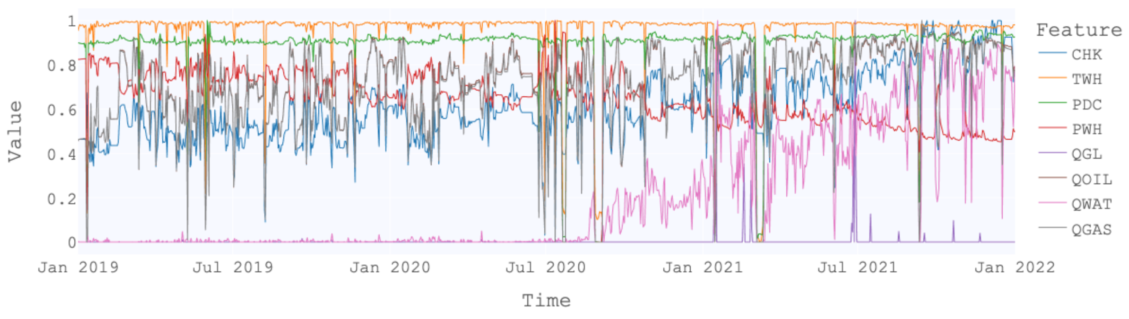


Figure 6.1: Example sensor measurements from well 0 for each non-label feature in the dataset. Note that the measured values are scaled to the interval $[0, 1]$ for visualization.

As specified in Chapters 4 and 5, the steady-state model inputs and target flow rates are mean compressions of the underlying time series over stable periods. Following the definition of stable periods outlined in Section 4.2, an algorithm provided by Solution Seeker identifies periods that are considered stable using the provided dataset. In short, the algorithm identifies time windows where the well state is considered stable based on the trend and variability of the control variables (choke valve and gas lift) and rule-based criteria applied to sensor measurements. The periods are represented as non-overlapping pairs of start and end timestamps for each well. The periods can then be used to generate the steady-state compressions by calculating the mean over all sensor measurements and control variables as specified in Section 5.1. The compression reduces the series of measurements for each stable period to a single data point. In total, the algorithm identified 18 475 stable periods in the provided dataset. An overview of the volume and distribution of the stable periods can be seen in Table 6.3 and Section 6.1.

Table 6.3: Overview of identified stable periods per well over the three-year period in the provided data set when using Solution Seeker's algorithm.

Well	Start date	End date	Span [days]	Num. periods	Sparsity [n/day]
Well0				2026	1.849
Well1				1792	1.635
Well2				1876	1.712
Well3				1846	1.684
Well4	01/01/2019	31/12/2021	1096	1868	1.704
Well5				1912	1.745
Well6				1629	1.486
Well7				2072	1.891
Well8				1758	1.604
Well9				1696	1.547

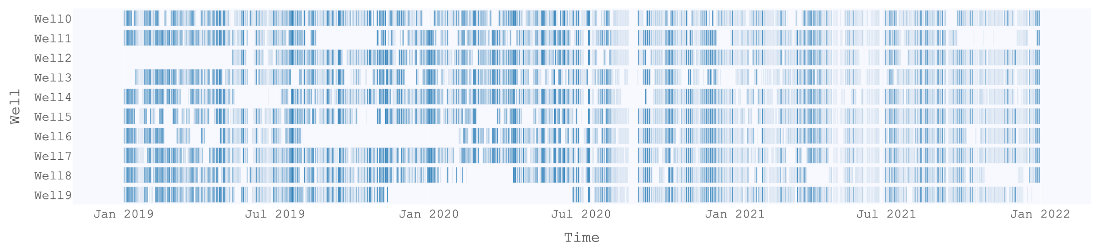


Figure 6.2: Overview of the distribution of identified stable periods over the three-year period in the provided dataset when using Solution Seeker's algorithm. Each vertical line represents the span of a stable period for the given well.

6.2 Data processing

This section describes the processing steps performed to transform the provided data into the datasets defined by the specifications outlined in Chapter 5. We begin by outlining how the derived features in the problem are generated in Section 6.2.1, before detailing the steps performed to filter the stable periods in Section 6.2.2. Then, the applied dataset splitting strategy is described in Section 6.2.3. Section 6.2.4 explains further processing steps performed after the dataset split. Finally, Section 6.2.5 illustrates the process of generating the final datasets used in experiments.

6.2.1 Feature generation

As indicated in Table 6.1, FOIL, FGAS, and QTOT are calculated features. These quantities are derived from the MPFM measurements in the provided data. To generate the features, the following transformations are performed:

1. All flow rates ($q_{jt,O}$, $q_{jt,G}$, $q_{jt,W}$ and q_{jt}^{GL}) are converted from volumetric flow per time (Sm^3/h) to an estimated mass per time (kg/s) using approximate phase densities. This makes the magnitudes of the measurements more comparable.

2. Using the flow rates converted to kg/s, the total multiphase flow is calculated as $Q_{jt} = q_{jt,O} + q_{jt,G} + q_{jt,W}$, as described in Section 2.3. As $q_{jt,G}$ denotes the measured gas flow rate at the wellhead, it also measures previously injected gas lift. Hence,

Q_{jt} is the total multiphase flow rate passing the choke valve, while $Q_{jt} - q_{jt}^{GL}$ is an estimate of the flow originating from the reservoir.

3. Given the comments in the previous point, reservoir rate fractions are computed as $\phi_{jt,O} = \frac{q_{jt,O}}{Q_{jt}-q_{jt}^{GL}}$, $\phi_{jt,G} = \frac{\max\{q_{jt,G}-q_{jt}^{GL}, 0\}}{Q_{jt}-q_{jt}^{GL}}$, where the denominator is an estimate of the total multiphase flow rate originating from the reservoir. A value of zero is assigned in cases where the fraction is not well-defined (division by zero) or where the denominator is negative. A negative denominator implies a greater amount of injected gas lift than measured total multiphase flow ($q_{jt}^{GL} > Q_{jt}$), which is erroneous. Likewise, $q_{jt,G} - q_{jt}^{GL} < 0$ implies a greater amount of injected gas lift than the measured gas rate, which is also erroneous.

6.2.2 Filtering stable periods

As the stable periods define the labels, it is important to ensure the identified stable periods that inaccurately reflect realistic steady-state scenarios. In theory, the flow rate should remain relatively constant over a stable period. However, a preliminary analysis of the provided data identified a number of issues to address to ensure the quality of the final datasets. For example, some periods contained irregularly low mean total flow rates. In other periods, the flow rate variability was considerably high. Additionally, some periods contained choke valve changes during the period. To address these issues, the per-feature standard deviation of the raw data over each stable period was calculated. Furthermore, the fraction of non-positive values of total flow rate measurements throughout the period was calculated. A stable period is then discarded if:

1. The period consists of more than 20 % non-positive total flow rate measurements. This corresponds to 119 periods, of which 88 periods do not contain a single positive rate measurement. From inspecting the raw data, most of the cases can be explained by flow-meter malfunction, well shut-ins, or a reported total flow rate (incl. gas lift) exceeding the injected gas rate.
2. The standard deviation of the reported flow rate over the stable period exceeds 1. This applies to 127 periods, which violates the assumption that the flow rate should be constant over the stable period.
3. The choke valve opening changes throughout the stable period. This filter applies to 24 periods and violates the assumption that control variables should be unchanged during and preceding a stable period.

Applying the above filters discarded 246 stable periods (some periods satisfy multiple conditions), leaving 18 229 remaining periods before dataset splitting.

6.2.3 Dataset splitting

Following the practices described in Section 3.4, the dataset is split into a training, a validation, and a test set. Note that as the labels are associated with stable periods, dataset splitting corresponds to partitioning the stable periods into three sets. Worth noting when defining a split strategy is the fact that measurements from different wells are correlated as they are connected to the same reservoir. For instance, the reservoir pressure decreases over time. Other effects are that FOIL and FGAS typically decrease over time for the entire reservoir. In addition, all wells are connected to the same production separator and are thus affected by the pressure in the separator. However, in contrast to the reservoir pressure, PDC is an input to the model and would therefore not cause information leakage of significance. For these reasons, it is important to prevent the time span of the test set from overlapping in time with the training data for correlated wells, introducing data leakage into the training step.

Furthermore, if extracting a randomly selected fraction from all data points as testing data, the models may learn to interpolate the training data rather than generalize, yielding poor predictions on future, unseen data. This would also apply to the validation set, as randomly sampled validation data would potentially guide the hyperparameter search towards model configurations able to interpolate the training data, rather than learning a general representation of the system capable of extrapolating into the future.

To simulate how the models would be trained and used in practice (evaluated on the most recent available data), we select the two last months (61 days) of available data as test data and the two months prior to that as validation data (61 days). This splitting strategy is sometimes referred to as a *fixed partitioning* split or combined in the form of a *forward chaining* split. An overview of the resulting split decomposed per well can be seen in Table 6.4, and a visualization of the partitioning of the stable periods are shown in Figure 6.3

Table 6.4: Overview of dataset splits, indicated by the number of stable periods per split for a given well. Note that an additional 95 periods are discarded from the training set in a filtering step described in Section 6.2.4.

Well	Training	Validation	Test	Total
Well0	1752	97	148	1997
Well1	1619	51	41	1711
Well2	1584	128	159	1871
Well3	1547	127	148	1822
Well4	1583	125	136	1844
Well5	1614	129	158	1901
Well6	1361	101	154	1616
Well7	1763	128	159	2050
Well8	1593	99	45	1737
Well9	1439	132	109	1680
Total	15855	1117	1257	18229

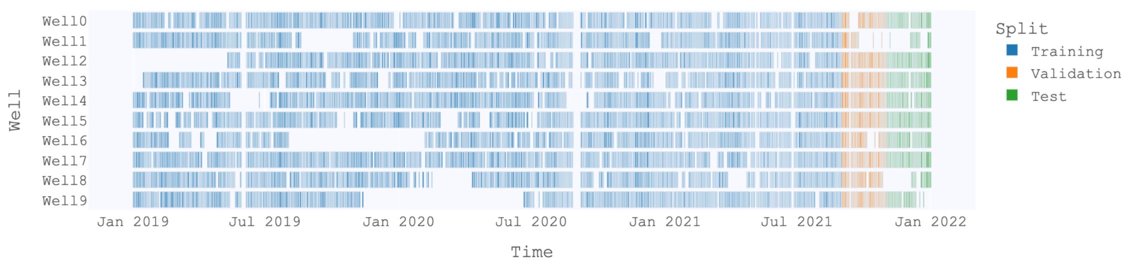


Figure 6.3: Illustration of the splitting of the dataset into training, validation, and test sets. Each vertical line represents the span of a stable period for the given well, with colors indicating which of the three sets the stable periods are included in.

6.2.4 Filtering and transformations after splitting

After the splitting, three different processing steps are performed on the data to produce the final datasets for model training. The data is transformed by a normalization step, filtered to eliminate extreme values, and categorical variables are transformed to a numeric representation, interpretable by the neural network architectures.

Per-well normalization
Input features and flow rate measurements are *normalized*, centering the data around zero on a per-well basis. The per-feature mean for a given well is calculated using all *raw* measurements in the time span of the *training* data. Given a feature z_{jt} and the set $\mathcal{T}$ of all uniformly sampled time points in the raw dataset spanning the duration of the training set, the normalized feature $\hat{z}_{jt}$ is calculated as

$$\hat{z}_{jt} = z_{jt} - \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} z_{j\tau} = z_{jt} - \bar{z}_j$$

This transformation is then applied to the validation and test data during evaluation. It should also be applied to future records during inference before being input into the trained models.

As it is desirable to keep information about differences in variance across wells, no per-well *standardization* is performed. Dataset features and the label are on a reasonable scale, such that no rescaling is necessary to ensure numerical stability during training.

Filtering of extreme measurements
From an initial exploratory data analysis of the dataset, it became evident that a minuscule number of feature values in the provided data take on extreme values. For example, some readings deviate by more than 30 standard deviations. Such extreme values are often a result of sensor faulty and not representative measurements. Therefore, such records were filtered by discarding all stable periods containing pressure or temperature readings that deviate from the per-well mean by more than five standard deviations. The filter discarded 95 stable periods, corresponding to 0.60 % of the training dataset. This filtering is applied to the training data only to avoid altering the validation and test data distributions. The primary motivation behind filtering these values is to increase stability during training - extreme values may result in very high squared error losses during early training iterations, causing large weight fluctuations.

Ordinal encoding of well identifiers
To indicate which well a given input state representation originates from, we want to include a representation of each well in the input data to the neural networks. An ordinal encoding is used to convert the well identifiers to integers. The encoding scheme is very simple. A new integer is assigned for each unique well in the dataset, starting at zero (Well0 $\rightarrow$ 0, Well1 $\rightarrow$ 1, ...). This encoding scheme is selected as it is a suitable input format for an embedding layer, which all the proposed models use. The encoded value can be interpreted as an index used for lookup in the embedding matrix, reducing the embedding operation from a matrix to a vector product.

6.2.5 Dataset generation

Using the raw dataset along with the processed stable periods, we can build final datasets compatible with the input specifications of the different model architectures outlined in

Chapter 5. Recall from Chapter 5 that the steady-state baseline model inputs the mean of each input feature over a stable period. Furthermore, the steady-state mean sequence models input a sequence of such vectors, where lengths of three, ten, and 30 are tested. Finally, the dynamic sequence models input sequences of resampled measurements. Here, both the time horizon covered by the sequence and the number of elements used in the sequence varies. Nine pairs of (time horizon, sequence length) are tested, using horizons of one, ten, and 30 days, and sequence lengths of $32, 64$, and 128. In total, this gives $1 + 3 + 9 = 13$ datasets. An overview of all the datasets can be seen in Table 6.5.

All the sequential datasets are produced by generating sequences associated with each label. Section 5.1 formally describes the methodology followed to generate the sequences. In short, we start by calculating steady-state mean vectors for each stable period. This serves as the input in training examples for the baseline model, where the label is the mean total flow rate for the corresponding stable period.

The datasets for the steady-state mean sequence models are produced by generating sequences of consecutive steady-state mean vectors of the specified length. Each training example thus consists of such a sequence as input. The label of the training example is the mean total flow rate for the stable period corresponding to the final vector of the sequence.

The input sequences for the dynamic sequence models are organized similarly, but the sequence elements differ considerably. Sequences are generated as follows. First, the raw uniformly sampled data is resampled to a resolution such that the given sequence length for the dataset spans the given time horizon. The sequence is then generated by the consecutive, uniformly resampled data points leading up to and during the given stable period. The ending time point of the corresponding stable period is used as the starting point for the resampling of the given sequence. Hence, each sequence uses a different starting point for the resampling. Furthermore, the resampled data points of different sequences are computed based on different parts of the original data. This approach ensures that two stable periods close in time do not generate identical input sequences, even if the resampling frequency is low.

Table 6.5: Overview of generated datasets for each model category.

Model category	Time horizon [days]	Sequence length [n]	Resampling interval [s]
Steady-state baseline model	-	1	-
Steady-state mean sequence models	-	3	-
		10	
		30	
Dynamic sequence models	1	32	2700
		64	1350
		128	675
	10	32	27 000
		64	13 500
		128	6750
	30	32	81 000
		64	40 500
		128	20 250

6.3 Properties of the dataset

Per-well variability
Figure 6.4 shows the standard deviation σ_{fj}, for feature f on well j calculated on the entire dataset. This decomposition shows that the variance for a given signal varies greatly across wells. For instance, $\sigma_{CHK,4}$ is over three times as large as $\sigma_{CHK,3}$, indicating that the choke control variable is more frequently adjusted for well 4. Similar patterns can be observed for the other features. Hence, the operating conditions vary for the different wells. This may imply that some flow rates can be more easily modeled for some wells than others.

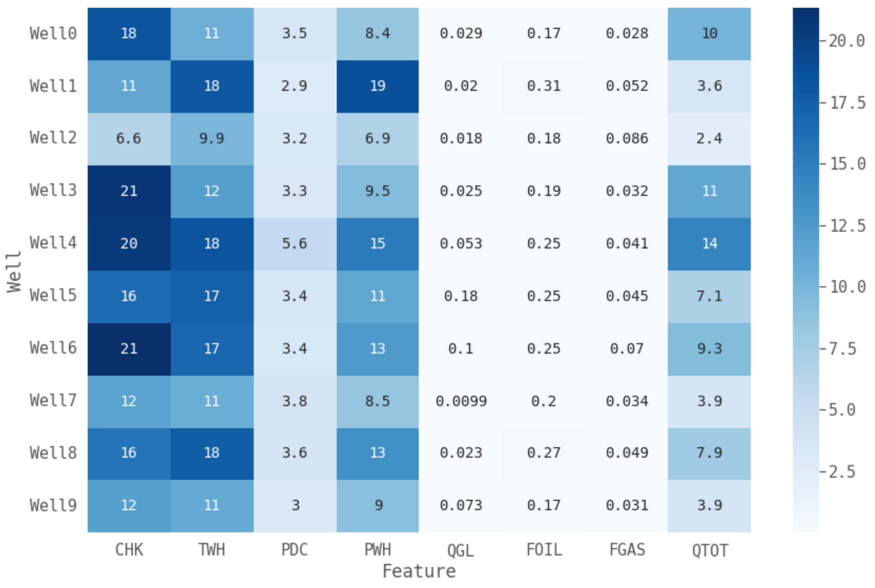


Figure 6.4: Standard deviations per feature-well pair, calculated on the entire dataset.

Correlations and physical properties
When studying the pairwise correlations between input features, as shown in Figure 6.5, we can observe some physical properties of the production system as captured by the sensor measurements. For one, we observe that the choke valve opening percentage and total flow rates are highly positively correlated ($\rho_{CHK,QTOT} = 0.86$) as is to be expected. A greater opening of the choke valve allows a greater volume of multiphase flow to stream past the MPFM sensor, thus increasing the total measured flow rate. Considering this correlation in combination with the standard deviations per feature-well pair in Figure 6.4, we can gain some insights into the behavior of flow rates in the data. Given the high standard deviation for CHK on different wells in the data, we expect QTOT to demonstrate high variability for the same wells. Considering the physical implications of varying the choke valve opening, this implication would be reasonable. Furthermore, this may imply that wells that have high standard deviations for CHK and QTOT represent more challenging wells to model flow rates for.

Furthermore, the heat map displays other correlations which are to be expected, such as the negative correlation between choke opening and upstream pressure ($\rho_{PWH,CHK} = -0.3$). If the choke valve opening increases, a higher volume of multiphase flow is let through the choke valve, and the upstream pressure decreases. Equivalently, we observe a slightly positive correlation between the downstream pressure and the choke valve opening ($\rho_{PDC,CHK} = 0.19$), which is also reasonable as when the choke increases, the pressure builds up downstream the choke valve as a greater volume flows. However, this pressure is systematically controlled by operators to maintain a constant level, as indicated in Figure 6.4 by the consistent standard deviations of a lower scale than the upstream pressure across wells. This may, in part, explain the somewhat lower correlation coefficient. These physical properties can be later used to evaluate whether the model's response to changes in input signals is in line with the expected response dictated by the physical relationships.

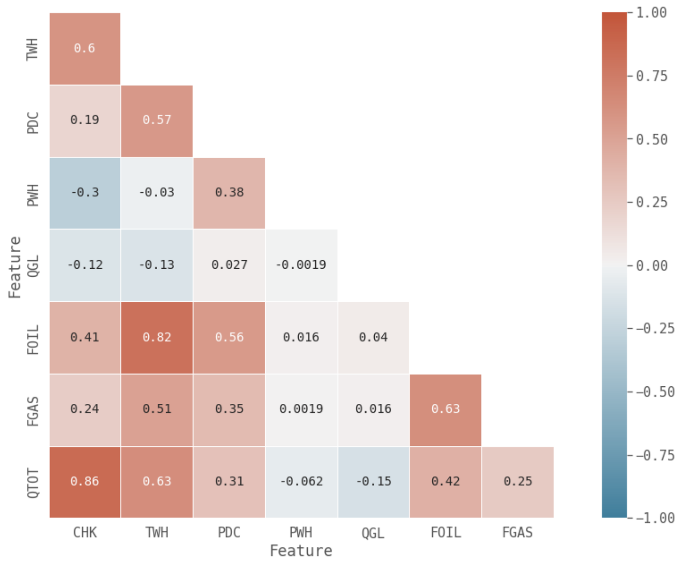


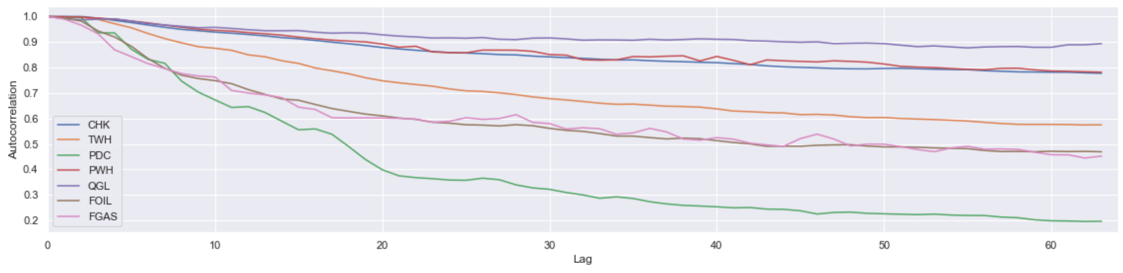
Figure 6.5: Visualization of the Pearson Correlation Coefficient (PCC) between all pairs of features in the dataset.

Autocorrelation

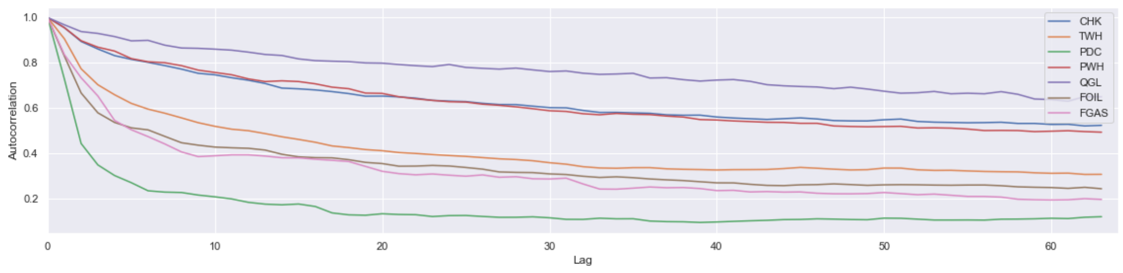
To better understand the dependencies in the data over time, lagged autocorrelation is analyzed per feature per well. For a sequence of a given length, lagged autocorrelation describes the correlation between the final element in the sequence and a given element from earlier time steps. By calculating this autocorrelation between the final and each of the elements of the sequence, we obtain an understanding of how current measurements for each feature depend on past measurements. See Figure 6.6 for a visualization of the mean per-feature lagged autocorrelation across all wells when using a sequence length of 64 elements spanning one, ten, and 30 days back in time.

From studying the lagged autocorrelations in Figure 6.6, it is evident that there is a relationship between a feature's current and recent values. This indicates, as expected, that there are temporal dependencies in the production data well beyond the maximal duration of stable periods. Hence, this observation motivates the proposed dynamic sequence models.

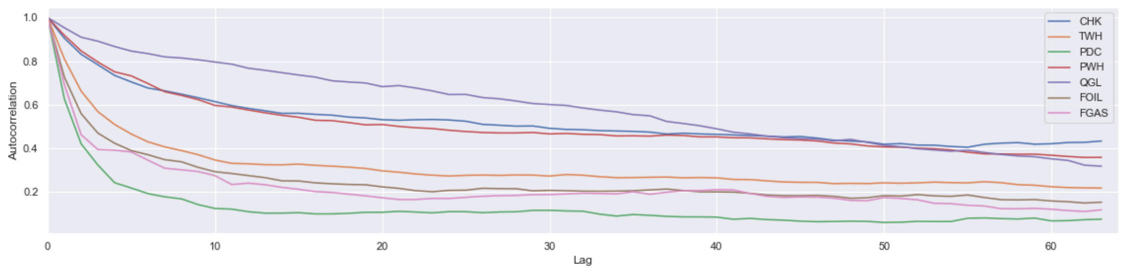
In addition, as showcased by Figure 1 in Appendix A, the per-feature autocorrelation varies considerably across wells. This suggests that the behavior of measurements preceding stable periods are vastly different across wells. Hence, knowing which well a given record originates from may be relevant for models to learn such dynamics. By informing the model of which well a sequence originates from, it may learn to adapt to the distinctive behaviors of each well accordingly.



(a) Lagged autocorrelation for sequences with a time horizon of 1 day.



(b) Lagged autocorrelation for sequences with a time horizon of 10 days.



(c) Lagged autocorrelation for sequences with a time horizon of 30 days.

Figure 6.6: Mean per-feature autocorrelation across all wells using a sequence length of 64 time steps for three different time horizons spanning a varying number of days back in time.

Chapter 7

Results

Using the processed versions of the provided dataset outlined in Chapter 6 we train models following the methodology described in Chapter 5. In total, we train 40 model variants subject for comparison. This forms the basis for the discussion of the research questions of the thesis. In this chapter, we present the results from training the different models. First, Section 7.1 describes the setup used to run the experiments to obtain the results. Then, the overall predictive performance of the models is showcased in Section 7.2. The remaining sections study selected high-performing models in greater detail. First, the model performance decomposed per well is shown in Section 7.3. Next, Section 7.4 illustrates the effect of perturbing the information in the earliest sequence elements, before Section 7.5 demonstrates the effects of decreasing the temporal granularity of input sequences. The trained models' ability to estimate more dynamic flow rates is evaluated in Section 7.6. Then, loss gradients with respect to the input data for different models are presented in Section 7.7. Finally, the results of re-training high-performing models on a lower data volume are shown in Section 7.8. The results are discussed in Chapter 8.

7.1 Experiment setup

Before presenting the results, we give a detailed overview of the experiments. First, Section 7.1.1 describes relevant details regarding the implementation of the experiments. Then, Section 7.1.2 presents specific details about the experiments run to produce the results.

7.1.1 Implementation details

The presented methodology is implemented in Python. The models and training logic are implemented using the machine learning library PyTorch version 1.11. This allows us sufficient freedom to tailor the involved components to our specifications while gaining the advantages of optimized data structures and abstracted logic. The hyperparameter tuning is performed using Optuna version 2.10, a hyperparameter optimization framework [Akiba et al., 2019]. We chose this library primarily due to its ease of integration with PyTorch training loops and because it implements state-of-the-art hyperparameter optimization algorithms. The hyperparameter tuning experiments were tracked using Neptune.ai, an experiment tracking framework that enables easy logging, storing, organization, introspection, and comparison of model metadata across runs.

Experiments were performed using four identical virtual machine instances running on Google Cloud Compute Engine. Each machine is equipped with an NVIDIA Tesla T4 GPU. This allows us to leverage GPU hardware acceleration using the CUDA framework (version 11.3) with PyTorch. By containerizing our code using Docker, this cloud computing setup allows us to easily scale the computation resources as required. Furthermore, the GPU acceleration considerably speeds up training as GPUs allow for substantial parallelization of vectorized operations. Considering the scale of the experiments, this speed-up considerably shortened the required running time to perform all experiments, despite requiring a longer implementation time. One experiment of 40 hyperparameter searches took approximately 80 GPU hours to run. Thus, it could be completed in 20 hours using four GPU accelerated machines.

7.1.2 Experiments

By applying the methodology described in Chapter 5, the 40 models outlined in Table 7.1 have been studied. Note that the models correspond to those presented in Table 5.1, but are here shown with the associated model names used in figures and tables in the following sections.

Table 7.1: Overview of model architectures tested in the thesis. Each model is identified by an alias, indicating the architecture, associated time horizon and sequence length, where applicable.

Architecture	Time horizon	Sequence length	Alias
FCN baseline	-	1	MTL-ONLY
FCN steady-state sequence	-	3, 10, 30	FCN-SEQ-`[sequence_length]`
	1 day	32, 64, 128	RNN-1D-`[sequence_length]`
RNN	10 days	32, 64, 128	RNN-10D-`[sequence_length]`
	30 days	32, 64, 128	RNN-30D-`[sequence_length]`
	1 day	32, 64, 128	LSTM-1D-`[sequence_length]`
LSTM	10 days	32, 64, 128	LSTM-10D-`[sequence_length]`
	30 days	32, 64, 128	LSTM-30D-`[sequence_length]`
	1 day	32, 64, 128	GRU-1D-`[sequence_length]`
GRU	10 days	32, 64, 128	GRU-10D-`[sequence_length]`
	30 days	32, 64, 128	GRU-30D-`[sequence_length]`
	1 day	32, 64, 128	TCN-1D-`[sequence_length]`
TCN	10 days	32, 64, 128	TCN-10D-`[sequence_length]`
	30 days	32, 64, 128	TCN-30D-`[sequence_length]`

First, an initial run of all experiments considering the 40 models presented above was conducted. When analyzing the results from the experiments it was evident that the GRU models performed very similarly to the LSTM models. The average validation and test MSE for each model architecture in this run can be seen in Appendix B. The mean validation and test MSE for the GRU model was 2.6 % and 9.5 % lower than for LSTMs, respectively. However, the best-performing LSTM model obtained a test MSE that was 24.1 % lower than the best-performing GRU model (see Appendix C). Thus, the GRU models with different input sequences performed similarly to LSTMs on average, but the best-performing LSTM performed marginally better than its GRU counterpart. In addition, the GRU models took approximately equal time to train compared with

LSTMs (see Appendix B). As explained in Section 3.6, GRUs and LSTMs are very similar. Furthermore, the literature contains extensive arguments stating that LSTM variants (such as the GRU) generally do not perform significantly better than the standard LSTM which is used in this thesis [Bai et al., 2018]. Hence, we decided to disregard GRU models in further experiments, as they provided limited additional insights beyond the other recurrent models. Furthermore, this decision considerably reduced the number of GPU hours required to run all experiments. Thus, the total number of models tested was reduced to 31.

For each of the resulting 31 model architectures presented above, a hyperparameter search over 300 trials was conducted where each model is trained for 100 epochs, following the procedure outlined in Algorithm 2. Each trial samples a value for each of the hyperparameters of the given model, using a given sampling strategy within a specified domain. An overview of the hyperparameters for the different model experiments is presented in Table 7.2. Early stopping with checkpoints was implemented to stop model training if the validation loss has not improved over the preceding 30 epochs. Checkpointing enables us to restore the model state at the point where validation loss was at its lowest. Furthermore, a median pruner was applied to avoid spending computational resources on unpromising trials. Trials are interrupted if the best intermediate result in the trial is worse than the median of intermediate results of previous trials at the same epoch, measured by validation loss. A warm-up period of 20 epochs was used, meaning that no pruning is allowed to occur for the first 20 epochs. In addition, pruning is disabled for the first ten trials to obtain a sufficient number of candidate optimization histories to compare with when pruning.

Table 7.2: Overview of hyperparameters used to identify models in the different categories.

(a) Overview of hyperparameters used to identify the steady-state baseline model and the steady-state sequence models.

Hyperparameter	Min	Max	Sampling strategy	Applies to	Description
`learning_rate`	1E-3	1E-2	log-uniform	All	Optimizer weight update step size
`l2_reg`	1E-4	1	log-uniform	All	L_2 regularization factor
`fcn_hidden_layers`	1	3	linear (step=1)	All	Number of hidden layers
`fcn_hidden_units`	32	128	linear (step=16)	All	Number of hidden units in each layer

(b) Overview of hyperparameters used to identify dynamic sequence models.

Hyperparameter	Min	Max	Sampling strategy	Applies to	Description
`learning_rate`	1E-3	1E-2	log-uniform	All	Optimizer weight update step size
`l2_reg`	1E-4	1	log-uniform	All	L_2 regularization factor
`fcn_hidden_layers`	1	2	linear (step=1)	All	Hidden layers in the FCN
`fcn_hidden_units`	32	64	linear (step=16)	All	Hidden units in each layer in FCN
`recurrent_layers`	1	3	linear (step=1)	RNN, LSTM, GRU	Number of recurrent layers
`hidden_size`	32	64	linear (step=16)	RNN, LSTM, GRU	Hidden state dimension
`kernel_size`	3	7	linear (step=1)	TCN	Width of each kernel
`num_channels`	16	32	linear (step=16)	TCN	Kernels applied in each layer

The hyperparameter search process described above is repeated five times for each of the 31 models. Counting the nine disregarded GRU searches from the first of the five runs, this gives a total of 164 experiment searches ($5 \cdot 31 + 9$). The process consumed a total of 328.5 GPU hours, corresponding to 13 GPU days and 16.5 GPU hours. By running four machines in parallel, the required hours are reduced by a factor of four, thus taking 82 hours, or 3 days and 10 hours to complete. The hyperparameters of all the final models and corresponding monitoring time can be found in Appendix C.

7.2 Overall predictive performance

The predictive performance of the trained flow rate estimation models is summarized in Table 7.3 and visualized in Figure 7.1. The error is calculated by comparing the predictions produced by the different models with the ground truth (label) and computing the squared error. By doing so for each training example, a vector of squared errors is obtained. The mean value of this error vector is reported as the MSE loss metric for a single experiment. Note that the error reported here corresponds to the predictive performance of the best model from all 300 trials for each experiment, averaged over the five identical experiments for each model configuration. Losses for individual experiments can be found in Appendix C.

Table 7.3: Average loss with standard deviations over 5 identical hyperparameter searches for each model.

Model	Val MSE	Test MSE	Test MSE (rel. to baseline) [%]
MTL-ONLY	0.92 ± 0.04	3.0 ± 0.55	0.0
FCN-SEQ-3	1.1 ± 0.07	2.74 ± 0.53	-8.49
FCN-SEQ-10	1.59 ± 0.05	4.08 ± 0.64	36.19
FCN-SEQ-30	1.5 ± 0.07	4.41 ± 0.59	47.17
RNN-1D-32	1.01 ± 0.08	6.06 ± 1.73	101.92
RNN-1D-64	0.96 ± 0.13	5.34 ± 1.93	77.94
RNN-1D-128	0.93 ± 0.09	4.22 ± 1.26	40.81
LSTM-1D-32	1.03 ± 0.04	6.93 ± 3.21	130.91
LSTM-1D-64	0.95 ± 0.05	5.09 ± 2.23	69.69
LSTM-1D-128	0.97 ± 0.06	8.61 ± 3.28	187.16
TCN-1D-32	1.01 ± 0.03	2.79 ± 0.75	-7.11
TCN-1D-64	0.91 ± 0.08	2.98 ± 0.55	-0.61
TCN-1D-128	0.97 ± 0.05	3.1 ± 0.62	3.35
RNN-10D-32	4.99 ± 0.13	12.47 ± 2.14	315.92
RNN-10D-64	2.27 ± 0.15	7.37 ± 0.52	145.7
RNN-10D-128	1.12 ± 0.11	4.82 ± 2.63	60.55
LSTM-10D-32	5.12 ± 0.22	14.97 ± 4.03	399.2
LSTM-10D-64	2.45 ± 0.18	10.79 ± 2.26	259.77
LSTM-10D-128	1.24 ± 0.06	7.85 ± 2.73	161.67
TCN-10D-32	5.08 ± 0.18	8.54 ± 0.22	184.89
TCN-10D-64	2.54 ± 0.06	5.95 ± 1.09	98.25
TCN-10D-128	1.19 ± 0.08	3.18 ± 0.98	5.88
RNN-30D-32	11.97 ± 0.28	17.44 ± 2.31	481.6
RNN-30D-64	7.18 ± 0.15	13.93 ± 1.93	364.46
RNN-30D-128	3.27 ± 0.24	10.62 ± 0.91	254.24
LSTM-30D-32	11.46 ± 0.22	21.42 ± 3.99	614.11
LSTM-30D-64	7.17 ± 0.11	14.96 ± 2.8	398.92
LSTM-30D-128	3.71 ± 0.03	13.95 ± 2.61	365.18
TCN-30D-32	11.54 ± 0.43	14.97 ± 1.91	399.24
TCN-30D-64	7.19 ± 0.22	11.45 ± 1.21	281.94
TCN-30D-128	3.25 ± 0.17	7.21 ± 1.25	140.38

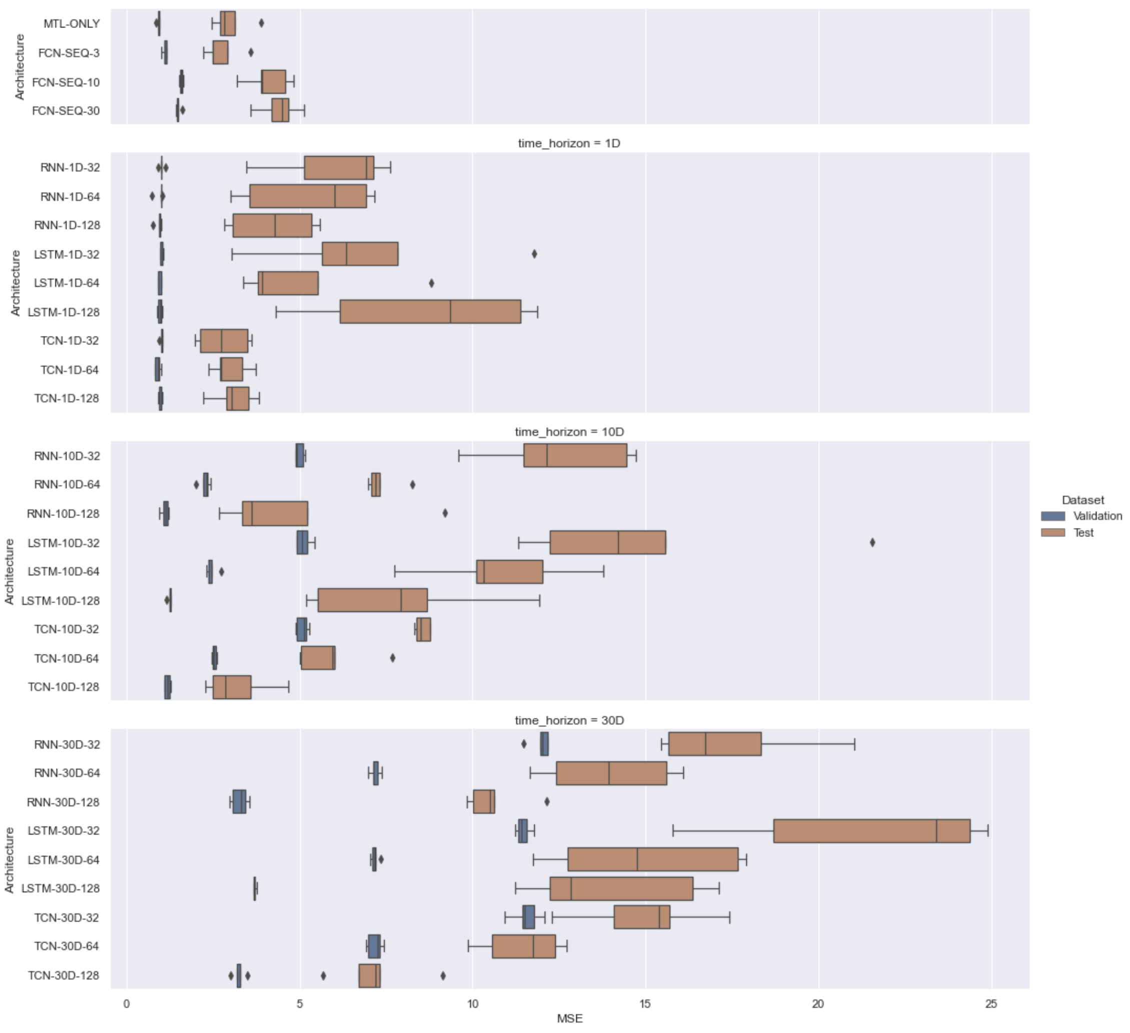
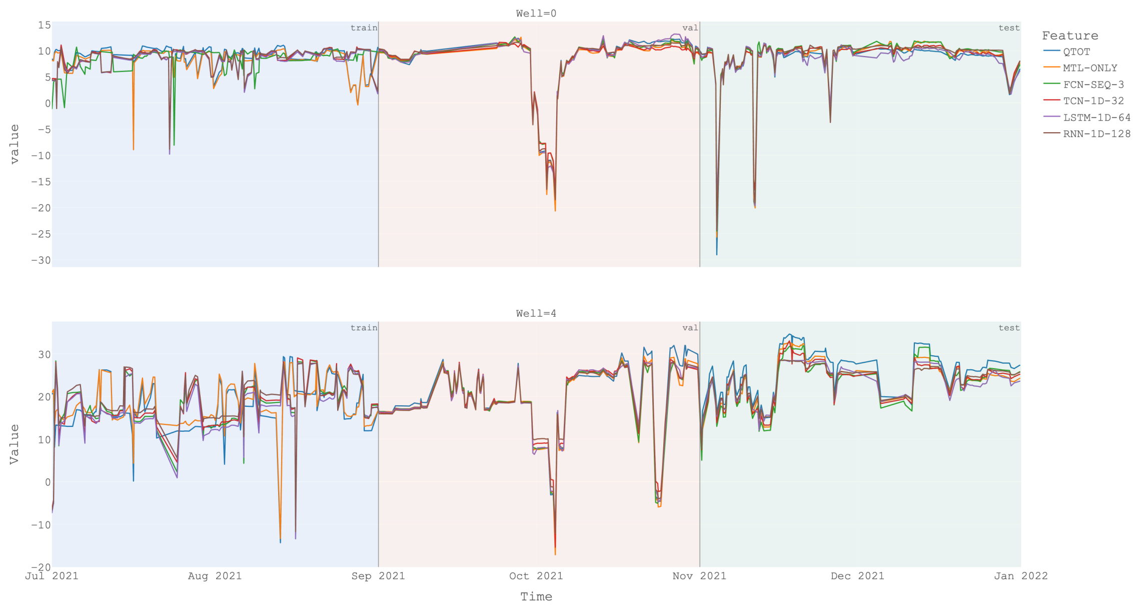
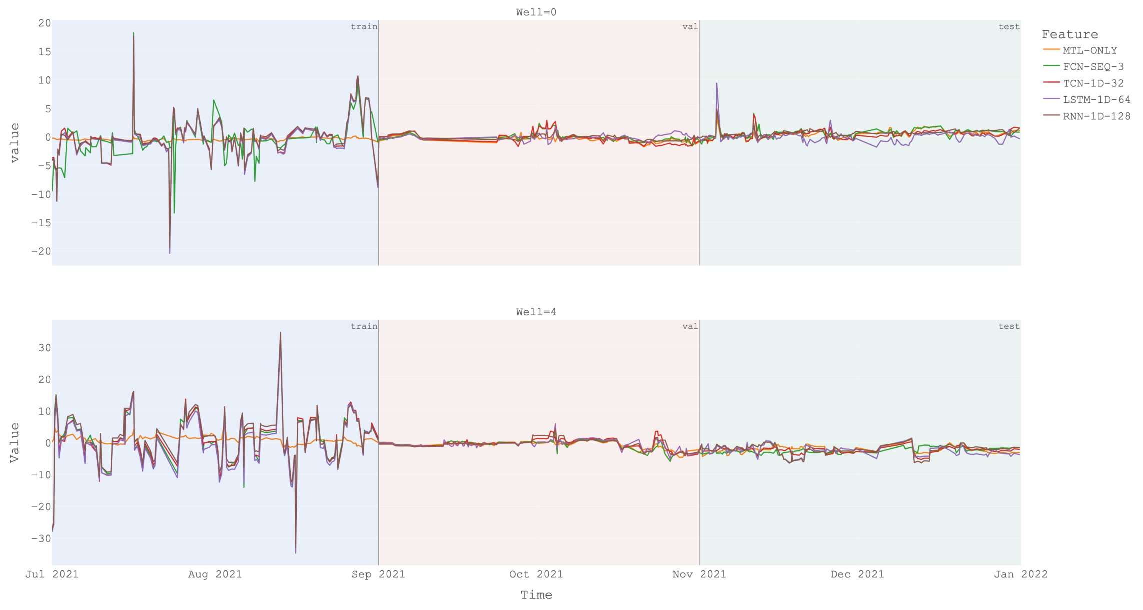


Figure 7.1: Box plots of MSE for all rate estimation models tested. Each box plot shows the validation or test MSE obtained for the given model configuration in 5 identical experiments. The box extends from the first quartile (25%) to the third quartile (75%) with a vertical line representing the median. The whiskers (horizontal lines) extend from the box by 1.5x the inter-quartile range (IQR). Any values outside of the IQR are shown as individual markers.

Based on the obtained average test losses, we select a group of best-performing models to use in subsequent analyses. For each of the studied architectures (MTL-ONLY, FCN-SEQ, RNN, LSTM, TCN) we identify the model obtaining the lowest average test MSE. Hence, the models used for further analyses (referred to as the best-performing models) are MTL-ONLY, FCN-SEQ-3, RNN-1D-128, LSTM-1D-64, and TCN-1D-32. For all mean test MSE analyses, averages over the five experiments of the model are used. For visualizations of predictions, we select the model with the lowest test MSE across the five experiments. Figure 3 illustrates flow rate estimates and errors on wells 0 and 4 for these models across each dataset split. The wells are selected based on the per-well loss obtained, where the models achieve relatively low loss on well 0 and high on well 4 (see Section 7.3). Corresponding visualizations on the test data for all ten wells can be found in Appendix D.



(a) Flow rate estimates on well 0 and 4 across dataset splits.

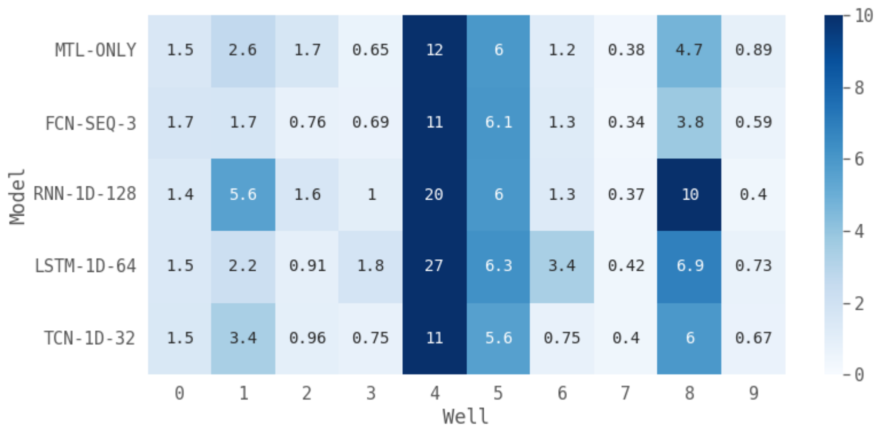


(b) Flow rate deviations from ground truth on well 0 and 4 across dataset splits.

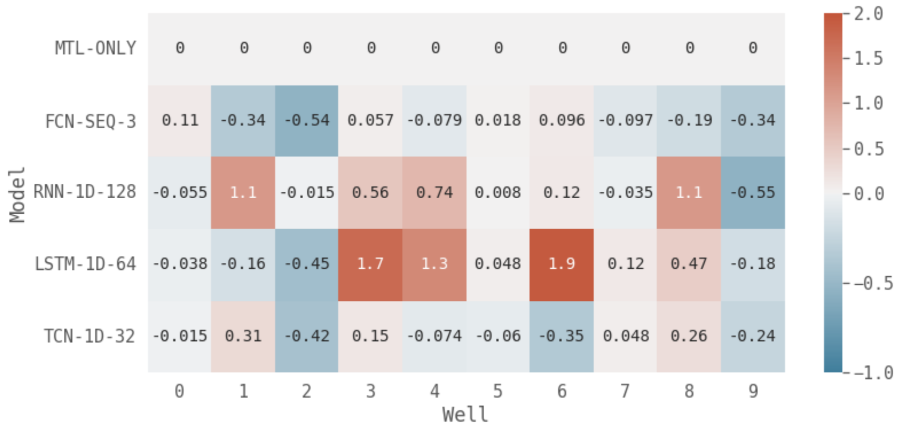
Figure 7.2: Flow rate estimates and error on two months of the training (blue background), validation (red background) and test set (green background) for the best-performing models.

7.3 Per-well predictive performance

A decomposition of average test loss per well can be seen in Figure 7.3a for each of the selected best-performing models. Figure 7.3b shows the relative change in test MSE loss per well as a multiplicative factor relative to the baseline model. The factor is given by $\frac{\bar{\epsilon}_m-\bar{\epsilon}_B}{\bar{\epsilon}_B}$ where $\bar{\epsilon}_m$ and $\bar{\epsilon}_B$ denotes the averaged per-well test loss over all experiments for model m and the baseline model, respectively.



(a) Rate estimation test MSE loss per well for the best-performing models.



(b) Relative change in rate estimation test MSE per well as a multiplicative factor relative to baseline loss.

Figure 7.3: Average flow rate estimation test loss decomposed per well for the best-performing models. The loss is averaged over the five experiments per model.

7.4 Perturbing information in earliest sequence elements

To investigate how the identified models exploit the information in the different elements of the input sequence, we test how estimation performance degrades when replacing the k earliest timesteps of the sequence with different masks. This experiment is conducted for each of the best-performing dynamic sequence models.

The experiment is conducted as follows. We want to investigate to what extent the k earliest timesteps of the sequence contribute to the output estimate. By "removing" the information in the k last sequence elements and studying how the rate estimates change, we may better understand how the model depends on the given elements. Hence, we replace the values in the elements using different masks. Three masking schemes are tested: replacing all k elements with zero-valued entries, replacing them with the mean for the corresponding feature on the given well across the entire dataset, and by simply slicing the sequence, removing the k first elements from the sequence. The dynamic sequence models are designed to support input sequences of arbitrary length, allowing the last scheme.

For each value of k up to the sequence length of the model, the above process is conducted. Hence, we mask all the k earliest elements of the sequence and calculate the test MSE. This is then done for each of the five identified models for each architecture, and an average is computed for each k. The resulting average test losses are visualized in Figure 7.4 for each of the mentioned models and schemes. See Table 8 in Appendix E for a breakdown of all losses.

The motivation behind the three schemes is to replace the information from further in the past in different ways. As we cannot fully know how the models exploit the information in the input sequence, we test multiple approaches. For instance, replacing elements with zeros may produce an unintentionally high loss, as the features are normalized, and thus a value of zero is meaningful. Additionally, the models may have learned patterns between elements over time. By perturbing the input in this way, we may unintentionally cause the model to identify learned patterns due to the masked values. Hence, to somewhat counteract these potential limitations of the experiment, we average over all the five identified models for each architecture. Additionally, by comparing three different schemes, we may better understand whether the highlighted issues occur.

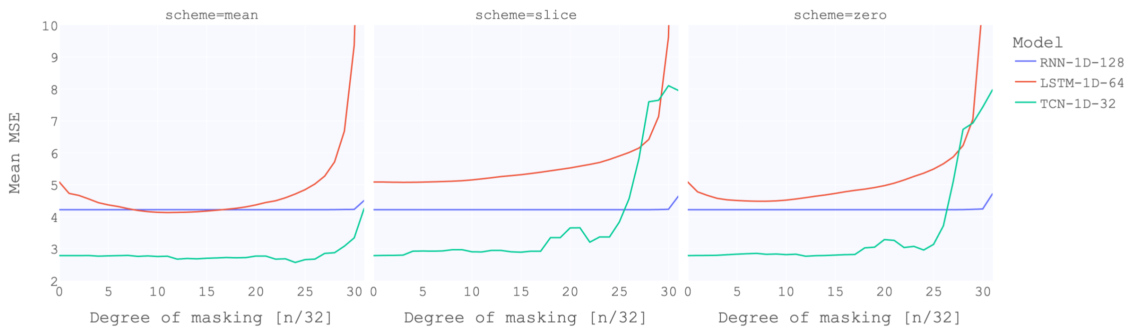


Figure 7.4: Mean MSE over five experiments for each of the best-performing models when masking the earliest parts of the input sequence in three different ways. Starting from the oldest history, each one-unit step along the horizontal axis represents extending the mask to the next $\frac{1}{32}$ elements of the input sequence. Thus, a value of zero indicates no masking, and a value of 31 indicates that only the most recent $\frac{1}{32}$ elements of every input sequence are left unmasked.

7.5 Decreasing temporal granularity in sequences

To study how the models leverage the temporal granularity in the input sequence, we evaluate their performance when given the same input sequences from the test data, but with lower temporal granularity. The temporal granularity is reduced by smoothing the input sequence using a Hann filter. The smoothed input sequences are then used to generate estimates, and the resulting test MSE is reported.

Due to the nature of the inputs for the steady-state mean sequence models, adjusting their temporal granularity is not a well-defined exercise. The sequence elements are not uniformly sampled and represent mean aggregations over different durations. Hence, they are not considered in this analysis. Conversely, the experiment is conducted on each of the dynamic sequence model architectures (RNN, LSTM, TCN). To ensure the results of each experiment are comparable, we choose three high-performing models trained on the same original temporal granularity. Hence, we perform the experiment on RNN-1D-64, LSTM-1D-64, and TCN-1D-64.

We test smoothing the input sequences to a varying extent by applying a Hann filter of different widths on each 64-length sequence. See Figure 7.5 for a visualization of how the smoothing affects a sequence from the test data, originally sampled such that 64 time points span one day. The change in test MSE for the selected models when smoothing the sequences using filter widths from 5 to 25 is shown in Table 7.4.

Table 7.4: MSE test loss for sequence models when a Hann filter of various lengths is applied to the input sequences.

window	RNN-1D-64	LSTM-1D-64	TCN-1D-64
No filter	5.336620	5.089275	2.980744
5	5.332416	5.080083	2.979081
7	5.328476	5.087332	2.976986
9	5.324852	5.100270	2.976939
11	5.329359	5.122141	2.978219
13	5.345006	5.150407	2.986311
15	5.369946	5.178543	3.003470
17	5.400840	5.217480	3.030521
19	5.436245	5.267374	3.065996
21	5.475800	5.324997	3.107052
23	5.519654	5.385316	3.153348
25	5.567754	5.444012	3.204417

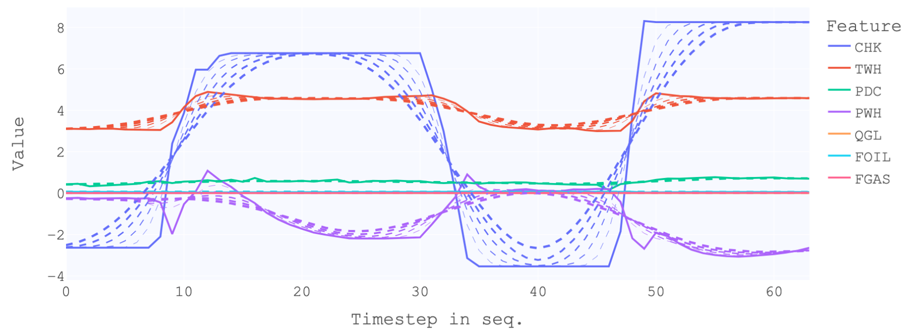


Figure 7.5: Example input sequence with varying degrees of smoothing. The dashed lines represent different features where a smoothing window of sizes 5, 9, 13, 17, 21, and 25 is applied. Thicker lines indicate a higher degree of smoothing. The solid line represents the unaltered feature.

7.6 Estimating dynamic flow rates

To test the identified models' ability to generalize, we evaluate the best-performing models on uniformly resampled test data, including both transient and stable measurements. That is, instead of estimating the mean flow rate over stable periods in the test period, we resample the raw total flow rates from MPFMs during the test period on a four-hour interval. Then, input sequences are generated in the same manner as before, using the ending time point of each four-hour period as the basis for resampling. The sequences are passed to the models, and the estimates are evaluated against the resampled flow rates using mean squared error. As the test period is 61 days, this gives $\frac{24}{4} \cdot 61 = 366$ test examples per well. The per-well MSE obtained for each model is outlined in Table 7.5. The per-well loss relative to the baseline is illustrated in Figure 7.6. Furthermore, the produced rate estimates and corresponding estimation errors for wells 0 and 4 are visualized in Figure 7.7. The same visualizations are shown for all wells in Appendix F.

Table 7.5: Per-well test MSE for best-performing models when evaluated on a uniformly resampled test dataset with a four-hour resampling frequency. The loss is averaged over the five experiments per model.

Well	MTL-ONLY	RNN-1D-128	LSTM-1D-64	TCN-1D-32
0	19.420511	7.575155	11.609964	11.527099
1	29.600805	8.067227	4.900905	24.057627
2	5.432682	2.784511	2.188552	3.623336
3	28.818920	8.338348	14.519588	14.267346
4	46.515068	30.621052	49.258179	23.757130
5	11.715349	7.942098	8.527229	8.838765
6	12.440696	2.558757	5.216428	7.699508
7	4.536375	1.561710	1.150869	2.875715
8	37.378502	17.472137	39.503654	28.717710
9	18.544081	4.824028	4.305014	22.690495
Mean	21.440298	9.174502	14.118039	14.805473

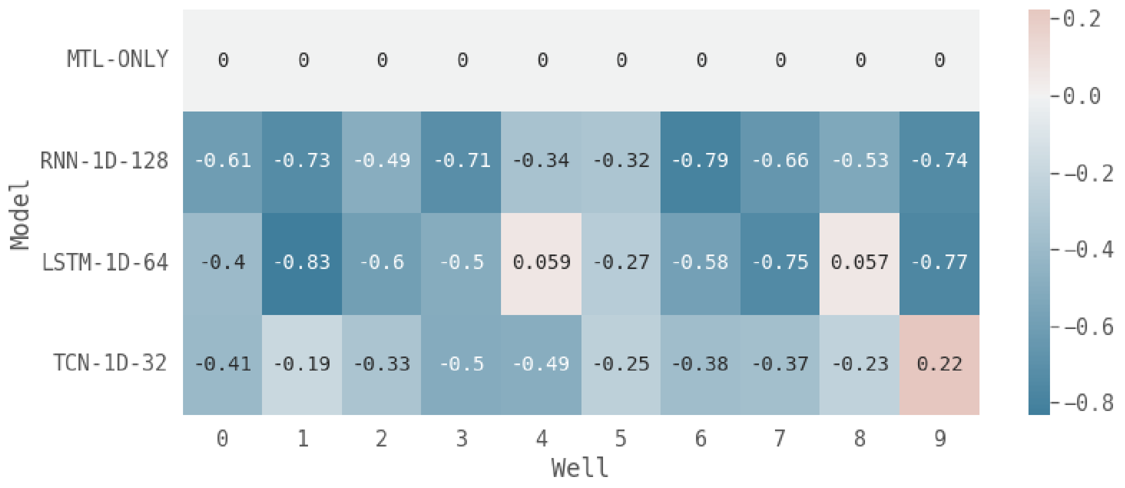
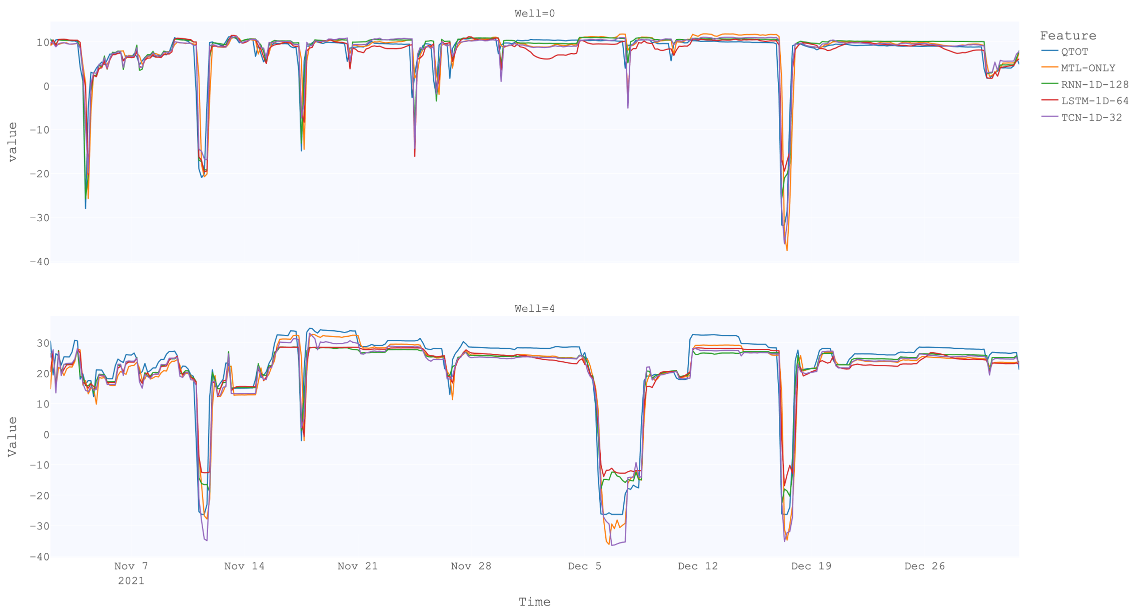
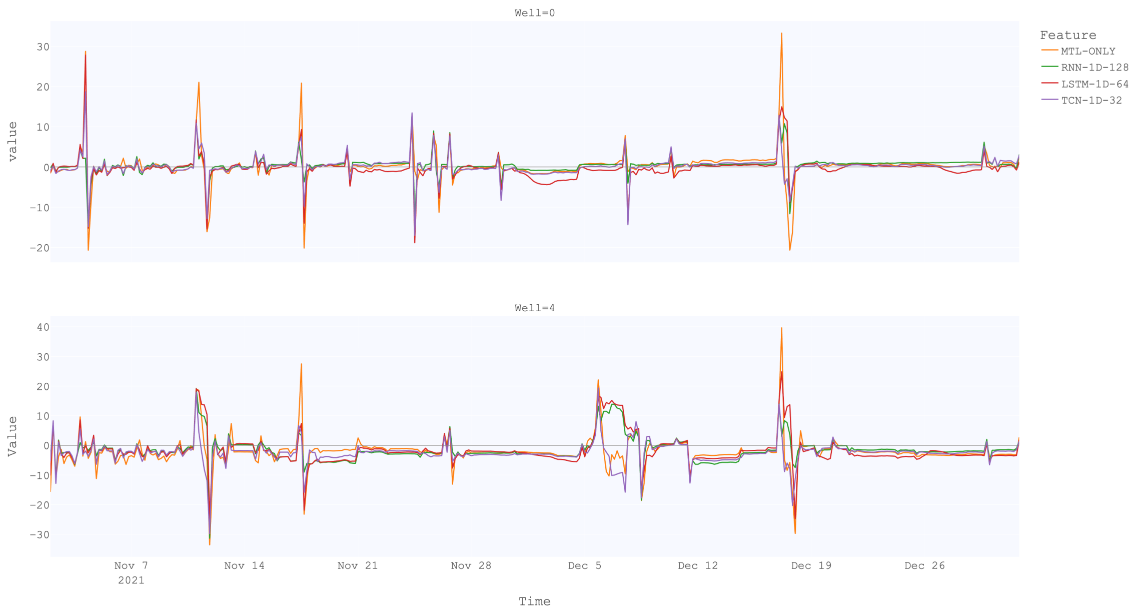


Figure 7.6: Relative change in rate estimation test loss per well as a multiplicative factor relative to baseline loss. The loss is averaged over the five experiments per model.



(a) Flow rate estimates on well 0 and 4.



(b) Flow rate deviations from ground truth on well 0 and 4.

Figure 7.7: Flow rate estimates and error on resampled test dataset for the best-performing models.

7.7 Calculating loss gradients with respect to input data

To better understand how information propagates from different parts of the input sequences to the corresponding model output, gradients of the loss with respect to each element in the input sequences are calculated.

This calculation is performed by leveraging the automatic differentiation features of PyTorch, capable of calculating the gradients with respect to any tensor in the computation graph. As the output is a result of a chain of tensor operations applied to the input, we can precisely calculate the gradient with respect to individual elements in the input tensors. By performing inference on the entire test dataset while recording gradients with respect to the input tensors, the mean gradient with respect to the loss can be calculated.

Figure 7.8 shows the mean over all logarithmically scaled absolute gradients for each (timestep, feature) pair. The gradients are visualized on a log scale to emphasize the underlying patterns.

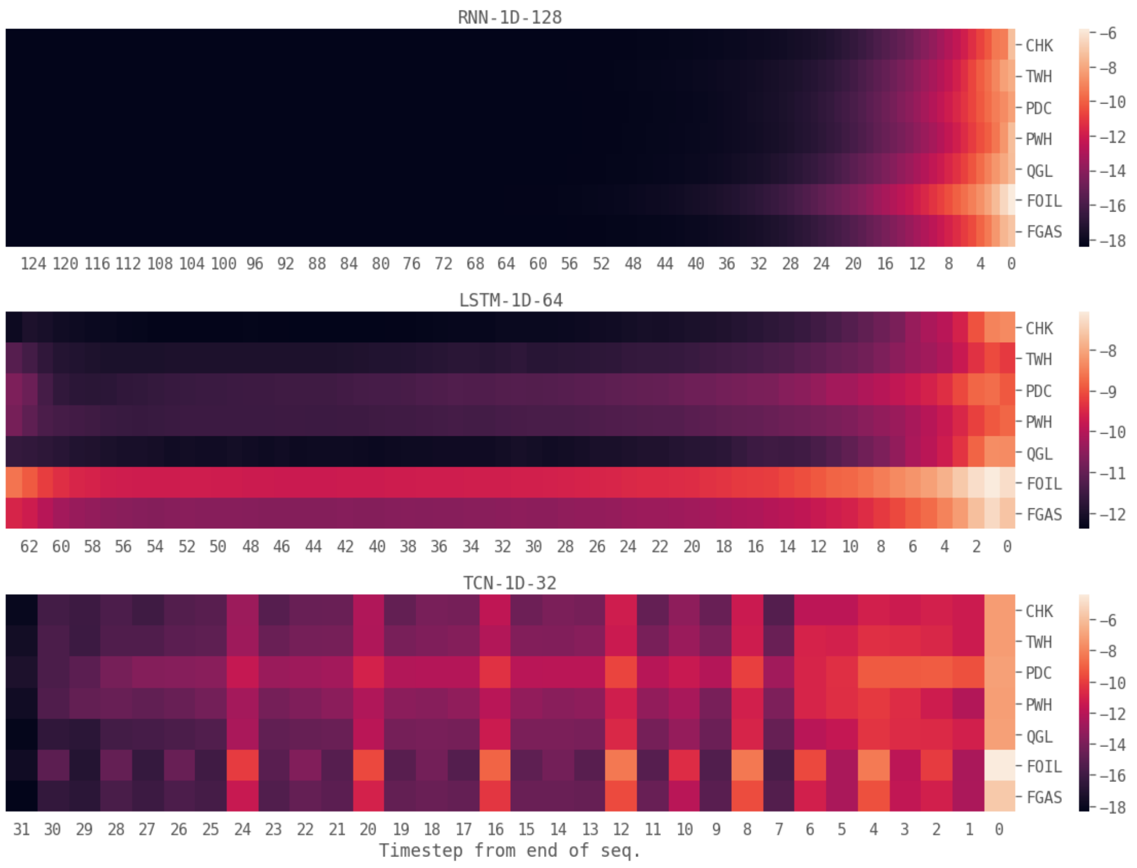


Figure 7.8: Visualization of mean, log-scaled, absolute gradients with respect to input calculated across each timestep/feature combination in the test dataset, reported for the best-performing sequence models.

7.8 Re-training best models on less data

To examine the viability of the proposed models under conditions where fewer labels are available, the best-performing models were re-trained on 50% of the available training data. To simulate lower data availability, the new dataset is generated by sampling 50% of the existing training examples uniformly at random. Hence, the examples span the same time period as before but with higher sparsity. Taking the best-performing model architectures, five new hyperparameter searches were performed for each model to identify optimal models with the new data availability. The average losses obtained are shown in Table 7.6.

Table 7.6: Change in average validation and test MSE loss when re-training best-performing models on 50% of training examples. All MSE losses are averages over five hyperparameter searches. Here “Before” refers to the MSE achieved before removing training examples.

	Validation		Test		
Model	Before	After	Before	After	Difference [%]
MTL-ONLY	0.924043	1.294520	2.999078	4.300697	43.40
FCN-SEQ-3	1.101926	1.863691	2.744508	5.348800	94.89
RNN-1D-128	0.929019	1.092267	4.222928	5.347290	26.63
LSTM-1D-64	0.952395	1.232829	5.089275	7.949846	56.21
TCN-1D-32	1.006969	1.340258	2.785874	2.921457	4.87

Chapter 8

Discussion

In this chapter, we discuss the results presented in Chapter 7, how they correspond to our hypotheses and what they might imply for the research questions presented in Chapter 4. We start by discussing what the results indicate with regards to the overall effects of estimating flow rates using sequences in Section 8.1, examining how the steady-state model compares with the sequence models. Then, Section 8.2 explores how the inclusion of transient history in the input sequence affects estimation performance. Finally, Section 8.3 investigates the implications of estimating flow rates based on sequences spanning different time horizons and temporal granularities. We emphasize that the implications of the results discussed in this chapter should primarily be considered within the context of the case study and do not necessarily hold in general. Furthermore, the results are empirical evidence and therefore should not be considered as proof but rather as indications of truths. Meanwhile, such indications represent valuable and insightful contributions to the literature.

8.1 Overall effects of sequence modeling

By studying the average test MSE obtained for the different models as outlined in Section 7.2, we may gain an initial understanding of the model performances. We see that most sequence models obtain a higher test MSE than the steady-state baseline model (MTL-ONLY), indicating a limited positive effect of leveraging sequences. However, the TCN and FCN-SEQ architectures stand out. Two one-day TCN models and FCN-SEQ-3 beat the baseline. Additionally, of the seven lowest obtained test MSEs, two are from FCN-SEQ models and four are from TCN models. All but one of these are within a 5% deviation from the baseline model's average test loss. Furthermore, we see that most LSTM and RNN models obtain comparable validation losses to TCN and FCN-SEQ, but considerably higher test losses. This may indicate that they do not generalize as well as the other models. Furthermore, standard deviations for RNN and LSTM test losses are considerably higher than the comparable TCN model, suggesting they are more inconsistent in what they learn. The results indicate that it is possible to outperform the state-of-the-art steady-state VFM using a dynamic VFM, in line with our hypotheses. However, studying the test MSE gives only limited indications of this. Furthermore, the average test loss does not consider a range of factors that may affect the results. By examining high-performing models in greater detail, we may paint a broader picture.

The per-well test loss for the best-performing models within each architecture shown in

Figure 7.3 illustrates a few differences. A desirable property of flow rate models is that their predictive ability is fairly similar across wells. However, we observe that the models demonstrate somewhat inconsistent performance for different wells. For instance, for wells 4, 5, and 8, at least four of the five models obtain test losses above 4.7. Meanwhile, all models obtain a test MSE below 2 for wells 0, 2, 3, 7, and 9. Thus, the models seem to learn to model flow rates for some wells more easily than others, regardless of model structure. Hence, we speculate that this inconsistency can be largely explained by the data rather than the models. By investigating the visualizations of the estimates for these wells in Appendix D, we see no striking patterns. Wells 4, 5, and 8 seem to contain periods of higher variability, however so does, for instance, well 2. Additionally, the standard deviations of the flow rates per well across the entire dataset in Section 6.3, show no decisive patterns. One explanation could be differences in operational conditions between the training and test sets, or that the MPFM sensor is misconfigured or has drifted.

Meanwhile, some differences between the steady-state baseline and the sequence models can be observed in the per-well loss analysis. The baseline model obtains the lowest test loss only for well 3. At the same time, on all wells where the baseline yields a test MSE below 2, the sequence models mostly perform comparably, as indicated above. However, on wells 4, 5, and 8, where the baseline performs poorer, it is beaten by the TCN model and FCN-SEQ-3, in two of three wells each. This may give slight indications that the baseline model somewhat overfits to the “easier” wells, whereas the sequence models can estimate flow rates slightly more accurately on more diverse wells. Hence, this observation may indicate that the sequence models are capable of generalizing better to different wells. This indication is strengthened by the larger gap between the average validation and test loss for MTL-ONLY (2.08) compared with FCN-SEQ-3 (1.64) and TCN-1D-32 (1.78).

In evaluating the trained models’ performance when estimating dynamic flow rates as shown in Section 7.6, different nuances between the models appear. By considering the per-well losses in Table 7.5 and Figure 7.6, we see that the baseline is outperformed by RNN on all wells, by the LSTM on all but wells 4 and 8, while TCN performs better on all but well 9. Additionally, all three dynamic sequence models outperform MTL-ONLY when considering the average test MSE across all wells. Furthermore, by examining the estimates visualized for wells 0 and 4 in Figure 7.7 and for all wells in Appendix F, we see that the baseline tends to overshoot and undershoot to more extreme degrees, especially when compared to the LSTM and RNN. These observations suggest that the dynamic sequence models have learned more general relationships from the input data to flow estimates than the baseline model. As the dynamic sequence models perform better in estimating the uniformly sampled flow rates, they seem to generalize somewhat better to more dynamic operational conditions. These observations may be explained by the fact that the dynamic models consider more transient data from before stable periods. As temporal considerations are the fundamental differences between the baseline and the sequence models, this result may indicate that the sequence models model temporal dependencies present in the production system. This result demonstrates one of the key benefits of leveraging sequences when estimating flow rates.

An important factor when considering the viability of a data-driven VFM is its dependency on data availability. As mentioned, limited flow rate measurements are available in most wells. Furthermore, the study performed in this thesis uses a somewhat exceptionally extensive dataset. Hence, we evaluate the viability of the models when re-trained on a more sparse training set in Section 7.8. The results show considerable differences, where the TCN model is hardly affected by cutting the training volume in half. Meanwhile, FCN-SEQ-3 yields almost twice as high a loss as before. This may be explained by the fact that the vectors in the input sequences are further spread out in time after removing half the stable periods and that this considerably complicates the problem. The test MSE loss of the RNN and TCN deteriorates less than that of the baseline, both in relative and absolute terms. Meanwhile, the TCN is the only model that outperforms the baseline in absolute terms both before and after re-training on less data. One factor influencing these results is that the dynamic sequence models leverage unlabeled data in the sequences and therefore are less vulnerable to a more sparse set of labels. Furthermore, the TCN's superior performance may be explained in part by its architecture. The training benefits highlighted in Section 3.6.3 such as the shorter gradient path may allow TCNs to leverage fewer sequences to obtain comparable results than recurrent models do. This result demonstrates that especially the TCN architecture may be a viable sequence model alternative when limited data is available.

While the results indicate varying effects of leveraging sequences in flow modeling, we emphasize that the steady-state baseline model resembles the state-of-the-art commercial model used by Solution Seeker. It is a complex model which performs exceptionally well in estimating flow rates in steady-state periods. By studying visualizations of the model estimates in Figure 7.2 and Appendix D, we see that outperforming the baseline model requires a highly accurate rate estimation model. Furthermore, as the studied problem regards estimating flow rates during stable periods of varying lengths, the dynamic sequence models are faced with a more challenging task than the steady-state model. The dynamic sequence models must learn to infer which part of the input sequence to estimate a mean flow rate over, whereas the baseline is implicitly informed of this through the input. Hence, the strengths and benefits of leveraging sequences indicated by the results should be considered in light of this. That is, it should be taken into account that the sequence models are faced with a more difficult task and are compared against an advanced baseline model when evaluating the implications of positive results.

8.2 Including transient history

Considering average test MSE in Section 7.2, using a sequence of three steady-state mean vectors appears to outperform models considering sequences of data from both stable and non-stable periods. FCN-SEQ-3 obtains the lowest average test loss of all the models tested, beating the best TCN by 0.03 test MSE on average and with a slightly lower standard deviation. Due to this insignificant difference in average test MSE, understanding which approach is the most viable cannot be done based on this metric alone. However, we see that for longer sequences of steady-state mean vectors, the performance is slightly worse, as they are outperformed by TCN-models and the baseline. This increase in average test MSE may be the same effect seen in Table 7.6 where FCN-SEQ-3 was re-trained on less training data. As the steady-state mean sequence length increases, the effective time horizon spanned by the sequence grows considerably. If the information in these elements has little influence over the target flow rate, they may only serve to increase model complexity with limited apparent benefit. Meanwhile, when studying average test loss, we see that using dynamic sequences yields highly varying performance.

As highlighted by the average test losses per well in Figure 7.3, FCN-SEQ-3 yields a lower mean loss than all the dynamic sequence models on four of the wells and beats the LSTM on all but well 0. Considering consistent behavior across wells, FCN-SEQ-3 obtains losses comparable to the other models on the low-loss wells. Furthermore, it considerably outperforms the other wells on well 8, where the next best model obtains a 23.7% higher loss. By examining the visualizations of the estimates on this well in Appendix D we can see that all models undershoot the flow rate, while FCN-SEQ-3 does so to a lesser extent. Hence, its per-well predictive behavior may be considered somewhat consistent, in line with the behavior of, for instance, the TCN. Comparing the behaviors on other wells, we see that the models learn approximately the same behaviors on most wells. However, as we have seen, the performance of the steady-state mean sequence model degrades considerably with lower availability of labels, rendering it less consistent than its dynamic sequence model counterparts in that regard.

By examining how the dynamic sequence models are affected by replacing information in the earlier timesteps of the input sequences, as shown in Section 7.4, we see that the best-performing one-day models are not considerably affected by perturbations. As the earliest timesteps of these sequences correspond to close to 24 hours before the target time period, the information removed is often outside the stable periods. However, as the models still perform comparably when these timesteps are masked, it may seem they are still able to model flow rates proficiently when considering measurements primarily from stable periods. From analyzing the lagged autocorrelations in Section 6.3 we saw that temporal dependencies exist beyond the duration of stable periods. Hence, this result indicates that such information is not relevant, or at least not leveraged, when estimating flow rates, at least for the one-day models. This analysis is further discussed in the context of temporal granularity and time horizons in Section 8.3.

8.3 Time horizons and temporal granularity

Of the time horizons tested, Section 7.2 demonstrates that shorter time horizons yield the highest performance. The models using one or three recent stable periods and the models considering the last day of data achieve considerably lower average test MSE compared with 10 and 30-day horizons. However, the 10-day TCN and RNN models using the highest temporal granularity are notable high-performers relative to other 10-day models. Furthermore, no models with a 30-day time horizon perform comparably to the best-performing models. Within both 10 and 30-day models, we observe a tendency toward higher performance with higher temporal granularity as hypothesized. This may indicate that the lower granularity sequences are of an insufficient resolution for the models to exploit the long history in meaningful ways relating to flow estimation. Hence, it may seem the models are better able to exploit measurements over the time horizon when it is sufficiently frequently resampled. Nonetheless, of the three granularities tested, it seems that the last 30 days yield an ill-suited summary for estimating the current flow rate.

For the one-day models, the temporal granularity does not seem to correlate as strongly with average test MSE. The one-day RNNs and LSTMs demonstrate a slight tendency towards higher average test loss with higher granularity. For the TCNs, similar performance is obtained for all three granularities. Hence, the TCNs seem capable of exploiting the information in the input sequence to produce sound flow rate estimates even when the granularity is lowered. This result may suggest that the past day contains sufficient relevant information for generating accurate estimates and that the dynamics can be represented in a relatively low resolution (a sampling interval of 45 minutes suffices for TCN). This observation is also supported by examining the effect of reducing the temporal granularity of one-day sequence models with a sequence length of 64 as done in Section 7.5. The MSE loss is only slightly increased when smoothing out the input data. Even with the most extreme smoothing that considerably flattens out the measurements, the test MSE only increases by 7.5% for TCN, 6.4% for LSTM, and 4.2% for RNN on average. Hence, it seems that signals of considerably high frequencies are not required to accurately estimate flow rates.

Meanwhile, the test losses of one-day RNNs and LSTMs have considerably higher standard deviations. Considering the already discussed high gaps between test and validation losses compared with TCNs, this may indicate that the recurrent models are less robust. That is, the learned relationships seem to be less consistently able to yield high performance compared with TCNs. This could perhaps be explained by the vastly different way in which the recurrent models learn compared with TCNs. As the recurrent models must propagate loss gradients through each time step, the stability of the gradients is fragile compared with TCNs, where gradients propagate through the network layers instead, which is a considerably shorter path. This substantial difference is illustrated in Section 7.7, where the loss gradients with respect to input sequences are presented. Here, we see systematically sustained higher activations across the input sequence for the TCN. Meanwhile, the RNN demonstrates gradually lower loss gradients for elements further back in time in the sequence across all features. Unsurprisingly, the gradients for the LSTM do not vanish as dramatically as for the RNN. The LSTM is designed to accomplish precisely this behavior (see Section 3.6.2). These observations indicate that the loss of TCNs is affected to a greater extent by earlier elements in the sequence. Hence, it may indicate that this architecture is more capable of exploiting information across the entire sequence than the recurrent architectures are.

The perturbations of the earliest elements of the input sequences demonstrated in Section 7.4 further illustrate the different degrees to which the dynamic sequence models exploit information across the sequence. We see that the RNN remains practically unchanged for all perturbations, indicating that it does not rely on information from the past day to any notable extent. This result is in line with the previous discussion regarding how and what the RNN learns. Meanwhile, the LSTM seems to perform slightly better when the earliest parts of the one-day period sequence are masked before the Mean MSE gradually increases as the degree of masking increases. When the masking covers a majority of the one-day period, the mean MSE explodes. The upwards trend as the masking increases may indicate that the LSTM depends on the more recent information in the sequence when producing estimates. A similar pattern occurs for the TCN model, which is surprisingly unaffected by a considerable degree of masking before the mean MSE drastically increases when the majority of the one-day period is masked.

We suspect that the initial improvements observed for LSTMs when using mean and zero masking may be explained by the initialization of the hidden state in the LSTM cells. Most recurrent layer implementations have the ability to let the final hidden state of previous batches serve as the initial hidden state in subsequent batches during training and may benefit from "warming up" the hidden state before doing inferences by passing earlier timesteps close in time through the recurrent layers of the network. We suspect that we see similar behavior here, where the masking of early sequence elements alters the initialization of the hidden state for subsequent timesteps within the same sequence. In this scenario, the early sequence elements may serve as a bootstrapping mechanic for subsequent recurrences within the same sequence. However, no deeper examinations have been performed in an attempt to better understand this behavior. Furthermore, it may emphasize the risks of blindly accepting the results of this experiment at face value. As highlighted in Section 7.4, we have limited knowledge of how the models respond to such perturbations. While the underlying aim of the masking is to 'forget" or 'not see" information, this may not be the realized effects.

Chapter 9

Concluding remarks and future research

This thesis has explored the problem of virtual flow metering, searching for models producing improved multiphase flow rate estimates. The promise of considering the past and thus exploiting widely available unlabeled data while learning across wells were studied through proposed state-of-the-art multi-task learning sequence models. Different representations of the well state history were produced and used in MTL sequence models to yield multiphase flow rate estimates. An empirical study was conducted using field data to examine the performance of the proposed sequence models when compared with a state-of-the-art MTL steady-state baseline model. The results indicate that using sequence models can improve flow rate estimation performance to a certain extent when combined with MTL models. Especially, high-performing sequence models demonstrate several behaviors ideal for VFMs where the baseline is lacking. The main contribution of this report is thus a new proposed methodology alleviating the issue of limited data volume available in well flow modeling by modeling well state history while learning across wells.

The results obtained in this report are only preliminary indications and come with some caveats. Future research is required to evaluate the promise of the methodology more precisely. However, the results still enable us to answer the research questions to a certain extent. The four supporting research questions presented in Chapter 4 were

1. How do models trained on historical sequences of well state data affect model performance in estimating flow rates compared with steady-state models?
2. How does the inclusion of well history from transient periods affect performance on flow estimation in stable periods?
3. How does the time horizon considered in input sequences of dynamic VFMs affect well flow rate estimation performance?
4. How does the temporal granularity of the input sequences of dynamic VFMs affect well flow rate estimation performance?

To which we hypothesized that

1. Models trained on historical sequences of well state data will improve performance in estimating flow rates compared with steady-state models

2. Including well history from transient periods leading up to stable periods will improve performance on rate estimation in said stable periods
3. For a given input sequence length, models that consider shorter time horizons (higher temporal granularity) will estimate well flow rates more accurately than models considering longer horizons (lower granularity)
4. For a given input time horizon, models that consider a longer input sequence (higher temporal granularity) will estimate flow rates more accurately than models considering a shorter input sequence (lower granularity)

The results discussed in the previous chapter strengthen the first hypothesis, giving indications of somewhat limited positive effects of using sequences over a steady-state model. However, the results should be considered in light of the state-of-the-art baseline and the more complex modeling problem faced by the sequence models. Second, the hypothesis regarding transient well history is somewhat strengthened by the results. While FCN-SEQ-3 obtained the lowest mean test loss, it is considerably dependent on data availability and demonstrates inferior estimation behavior in some cases. Third, as hypothesized, models considering a shorter time horizon seem to perform better than those considering longer horizons. Finally, there are clear tendencies towards better performance for models regarding longer sequences for a given time horizon. However, this is not the case for the one-day horizon, where a lower temporal granularity is sufficient.

Consequently, these answers shed light on the primary research question, which was formulated as follows:

> *How does leveraging sequences of historical well state data affect the flow rate estimates of data-driven MTL-based virtual flow meters?*

The results certainly give reason to believe that sequence modeling may improve flow rate estimates of MTL-based VFMs. If not in terms of a lower mean test loss, they certainly may improve VFMs through their several superior model traits. Thus, this report has fulfilled its objective of examining the viability of sequence models. Furthermore, the results may aid Solution Seeker in their research on virtual flow metering, providing valuable insights into where further research efforts should be directed.

The thesis has conducted a somewhat broad preliminary exploration of sequence modeling for virtual flow metering. Furthermore, in obtaining the results in the thesis, several assumptions, limitations, and simplifications were made. All of these measures may affect the obtained results, and there is a need for further research to determine the proficiency of sequence models more accurately. Therefore, we now discuss recommendations for future research.

As mentioned, the thesis imposes an added complexity to the sequence models by limiting the problem to estimating flow rates in stable periods. Further research should investigate the promise of sequence models when removing this limitation. This could for instance be accomplished by performing a similar study where uniformly sampled flow rates are estimated, given a resampling frequency. In this way, one could avoid tasking the models with learning over what period to estimate flow rates (as it would be constant), while simultaneously producing a model potentially capable of estimating flow rates under more dynamic conditions. In such a study, it could be relevant to examine how different input representations influence the performance in terms of temporal granularity and time horizon. It may be the case that in this more dynamic problem setup, the models depend more on data from further back in the past than observed in this thesis.

As discussed, using sequences is one way of leveraging unlabeled data in virtual flow metering. Meanwhile, the semi-supervised learning approach studied in our project report represents another successful method of improving flow rate estimates through unlabeled data. We recommend that future research study the effects of combining sequence modeling and semi-supervised learning in a multi-task setting. This could for instance be done by studying a sequential autoencoder that uses sequences of well state history to learn more informative state representations in an unsupervised manner. Sequence autoencoders using recurrent network architectures are well-studied in the literature and have been applied in other domains for similar purposes [Dai and Le, 2015], and examples of TCN encoder-decoder for inspiration are also present in the literature [Lea et al., 2017]. The learned representations are thus informed by the well state history and can be trained using widely available unlabeled data. These representations may then be used as the input state representation to estimate flow rates. The promise of this approach is the combined exploitation of both temporal information and a considerably larger volume of the available data, which could result in high-performing models.

We recommend further research evaluate the applicability of the proposed models for industrial applications. As the purpose of this thesis was to study the plausibility of the suggested models in a broader scope, we did not prioritize such analyses. For example, we recommend conducting sensitivity analyses of models identified by the proposed methodology to ensure they adhere to physical relationships in the production system, as expected from a commercial VFM. One such relationship is the positive correlation between the pressure upstream of the choke valve and the output flow rates. A viable VFM will successfully estimate higher flow rates if the upstream pressure changes, all else equal. Similarly, we expect the opposite for an isolated upstream pressure decrease. Furthermore, the proposed models' applicability may be affected by the way in which they are trained in the thesis. Here, we use the final two months of the available data for testing, the two previous months before for validation, and the remainder of the three-year period for training. In production, the model should be retrained frequently as new sensor measurements arrive so that it is trained on the most up-to-date data. As the operating conditions of wells change over time, one would expect a model to perform worse on new data as time passes without such a training scheme. Hence, the models should be examined when trained using such a scheme to evaluate their applicability under representative production conditions.

Finally, we would like to conclude the thesis by highlighting that we have proposed models that outperform a state-of-the-art commercial virtual flow metering model that has been developed through the investment of an immense number of hours. Hence, the promise of sequence models has been demonstrated. In turn, the results contribute to the virtual flow metering literature and may aid Solution Seeker's research. Consequently, the purpose of the thesis is fulfilled.

Bibliography

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

T. A. Al-Qutami, R. Ibrahim, I. Ismail, and M. A. Ishak. Development of soft sensor to estimate multiphase flow rates using neural networks and early stopping. *International Journal on Smart Sensing & Intelligent Systems*, 10(1), 2017.

T. A. AL-Qutami, R. Ibrahim, I. Ismail, and M. A. Ishak. Virtual multiphase flow metering using diverse neural network ensemble and adaptive simulated annealing. *Expert Systems with Applications*, 93:72–85, 2018.

N. Andrianov. A machine learning approach for virtual flow metering and forecasting. *IFAC-PapersOnLine*, 51(8):191–196, 2018.

S. Bai, J. Z. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

H. P. Bieker, O. Slupphaug, and T. A. Johansen. Real-time production optimization of oil and gas production systems: A technology survey. *SPE Production & Operations*, 22(04):382–391, 2007.

T. Bikmukhametov and J. Jäschke. First principles and machine learning virtual flow metering: A literature review. *Journal of Petroleum Science and Engineering*, 184: 106487, 2020.

S. Böck, M. E. Davies, and P. Knees. Multi-task learning of tempo and beat: Learning one to improve the other. In *ISMIR*, pages 486–493, 2019.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2): 57–83, 2002.

R. Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

O. Chapelle, B. Scholkopf, and A. Zien. *Semi-supervised learning*. MIT Press, Cambridge, 1 edition, 2006.

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

S. Corneliussen, J.-P. Couput, E. Dahl, E. Dykesteen, K.-E. Frøysa, E. Malde, H. Mostue, P. O. Moksnes, L. Scheers, and H. Tunheim. *Handbook of Multiphase Flow Metering*. The Norwegian Society for Oil and Gas Measurement and The Norwegian Society of Chartered Technical and Scientific Professionals, 2 edition, 2005.

A. M. Dai and Q. V. Le. Semi-supervised sequence learning. *Advances in neural information processing systems*, 28, 2015.

G. Falcone, G. Hewitt, C. Alimonti, and B. Harrison. Multiphase flow metering: current trends and future developments. *Journal of Petroleum Technology*, 54(04):77–84, 2002.

S. Falkner, A. Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

Z. Gan, C. Li, J. Zhou, and G. Tang. Temporal convolutional networks interval prediction model for wind speed forecasting. *Electric Power Systems Research*, 191:106865, 2021.

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

A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

B. Grimstad, M. Hotvedt, A. T. Sandnes, O. Kolbjørnsen, and L. S. Imsland. Bayesian neural networks for virtual flow metering: An empirical study. *arXiv preprint arXiv:2102.01391*, 2021.

A. Gryzlov, L. Mironova, S. Safonov, and M. Arsalan. Artificial intelligence and data analytics for virtual flow metering. In *SPE Middle East Oil & Gas Show and Conference*. OnePetro, 2021.

V. Gunnerud and B. Foss. Oil production optimization—a piecewise linear model, solved with two decomposition strategies. *Computers & Chemical Engineering*, 34(11):1803–1812, 2010.

L. S. Hansen, S. Pedersen, and P. Durdevic. Multi-phase flow metering in offshore oil and gas transportation pipelines: Trends and perspectives. *Sensors*, 19(9):2184, 2019.

Y. He and J. Zhao. Temporal convolutional networks for anomaly detection in time series. In *Journal of Physics: Conference Series*, volume 1213, page 042050. IOP Publishing, 2019. no. 4.

M. F. Heggland and P. Kjærran. An exploration of semi-supervised multi-task learning for multiphase flow rate estimation in oil and gas wells. Project report for NTNU course TIØ4500, 2021.

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

M. Hotvedt, B. Grimstad, and L. Imsland. Developing a hybrid data-driven, mechanistic virtual flow meter - a case study. *IFAC-PapersOnLine*, 53(2):11692–11697, 2020.

E. Idsø, I. L. Sperle, R. Aasheim, and M. S. Wold. Automatic subsea deduction well testing for increased accuracy and reduced test time. In *Abu Dhabi International Petroleum Exhibition and Conference*. OnePetro, 2014.

International Organization for Standardization. ISO 13628-1:2005 Petroleum and natural gas industries — Design and operation of subsea production systems — Part 1: General requirements and recommendations. Standard, International Organization for Standardization, Brussels, BE, Nov. 2005.

L. P. Kaelbling. *MIT 6.036 - Introduction to Machine Learning - Lecture Notes*. Massachusetts Institute of Technology, 2019. `https://phillipi.github.io/6.882/2020/notes/6.036_notes.pdf`.

A. Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. URL `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Retrieved June 3 2022.

T. S. Kim and A. Reiter. Interpretable 3d human action analysis with temporal convolutional networks. In *2017 IEEE conference on computer vision and pattern recognition workshops (CVPRW)*, pages 1623–1631. IEEE, 2017.

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

P. Lara-Benítez, M. Carranza-García, J. M. Luna-Romera, and J. C. Riquelme. Temporal convolutional networks applied to energy-related time series forecasting. *applied sciences*, 10(7):2322, 2020.

C. Lea, R. Vidal, A. Reiter, and G. D. Hager. Temporal convolutional networks: A unified approach to action segmentation. In *European Conference on Computer Vision*, pages 47–54. Springer, 2016.

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

D. Li and J. Qian. Text sentiment analysis based on long short-term memory. In *2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI)*, pages 471–475. IEEE, 2016.

P. Li, Z. Zhang, Q. Xiong, B. Ding, J. Hou, D. Luo, Y. Rong, and S. Li. State-of-health estimation and remaining useful life prediction for the lithium-ion battery based on a variant long short term memory neural network. *Journal of power sources*, 459:228069, 2020.

K. Loh, P. S. Omrani, and R. van der Linden. Deep learning and data assimilation for real-time production prediction in natural gas wells. *arXiv preprint arXiv:1802.05141*, 2018.

K. Løvland. Semi-supervised learning for well flow modelling. Master's thesis, NTNU, 2021. Supervised by Solution Seeker, full text non-public until 2024-05-31.

F. Lässig. Temporal convolutional networks and forecasting, 2020. URL `https://unit8.com/resources/temporal-convolutional-networks-and-forecasting/`. Retrieved June 5 2022.

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

R. Mercante and T. A. Netto. Virtual flow predictor using deep neural networks. *Journal of Petroleum Science and Engineering*, 213:110338, 2022.

T. Mitchell. *Machine Learning*. McGraw Hill, 1997. `http://www.cs.cmu.edu/~tom/mlbook.html`.

C. Olah. Understanding lstm networks, 2015. URL `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Retrieved June 3 2022.

A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

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

S. Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

A. T. Sandnes, B. Grimstad, and O. Kolbjørnsen. Multi-task learning for virtual flow metering. *arXiv preprint arXiv:2103.08713*, 2021.

P. Shoeibi Omrani, I. Dobrovolschi, S. Belfroid, P. Kronberger, and E. Munoz. Improving the accuracy of virtual flow metering and back-allocation through machine learning. In *Abu Dhabi International Petroleum Exhibition & Conference*. OnePetro, 2018.

R. H. Shumway, D. S. Stoffer, and D. S. Stoffer. *Time series analysis and its applications*, volume 3. Springer, 2000.

S. Siami-Namini, N. Tavakoli, and A. S. Namin. A comparison of arima and lstm in forecasting time series. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 1394–1401. IEEE, 2018.

A. Society of Petroleum Engineers. The SI Metric System of Units and SPE Metric Standard, 1994. URL `https://www.spe.org/authors/docs/metric_standard.pdf`.

Stanford University. *Multi-Task Learning & Transfer Learning Basics*. Lecture from fall 2021 course CS330: Deep Multi-Task and Meta Learning, 2021. `https://cs330.stanford.edu/slides/cs330_multitask_transfer_2021.pdf`.

J. Sun, X. Ma, and M. Kazi. Comparison of decline curve analysis dca with recursive neural networks rnn for production forecast of multiple wells. In *SPE Western Regional Meeting*. OnePetro, 2018.

L. Sun, K. Jia, D.-Y. Yeung, and B. E. Shi. Human action recognition using factorized spatio-temporal convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4597–4605, 2015.

M. Sundermeyer, R. Schlüter, and H. Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

J. E. Van Engelen and H. H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.

H. Wang, D. Hu, M. Zhang, and Y. Yang. Multiphase flowrate measurement with time series sensing data and sequential model. *International Journal of Multiphase Flow*, 146:103875, 2022.

P. Wang. *Development and applications of production optimization techniques for petroleum fields*. Stanford University, 2003.

R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Backpropagation: Theory, architectures, and applications*, pages 433–486, 1995. Hillsdale, NJ: Erlbaum.

Z. Xu, F. Wu, L. Zhu, and Y. Li. Lstm model based on multi-feature extractor to detect flow pattern change characteristics and parameter measurement. *IEEE Sensors Journal*, 21(3):3713–3721, 2020.

G. Zangl, R. Hermann, and C. Schweiger. Comparison of methods for stochastic multiphase flow rate estimation. In *SPE Annual Technical Conference and Exhibition*. OnePetro, 2014.

H. Zhang, Y. Yang, M. Yang, L. Min, Y. Li, and X. Zheng. A novel cnn modeling algorithm for the instantaneous flow rate measurement of gas-liquid multiphase flow. In *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, pages 182–187, 2020.

W. Zhao, Y. Gao, T. Ji, X. Wan, F. Ye, and G. Bai. Deep temporal convolutional networks for short-term traffic flow forecasting. *IEEE Access*, 7:114496–114507, 2019.

Appendix

A Lagged autocorrelations per feature

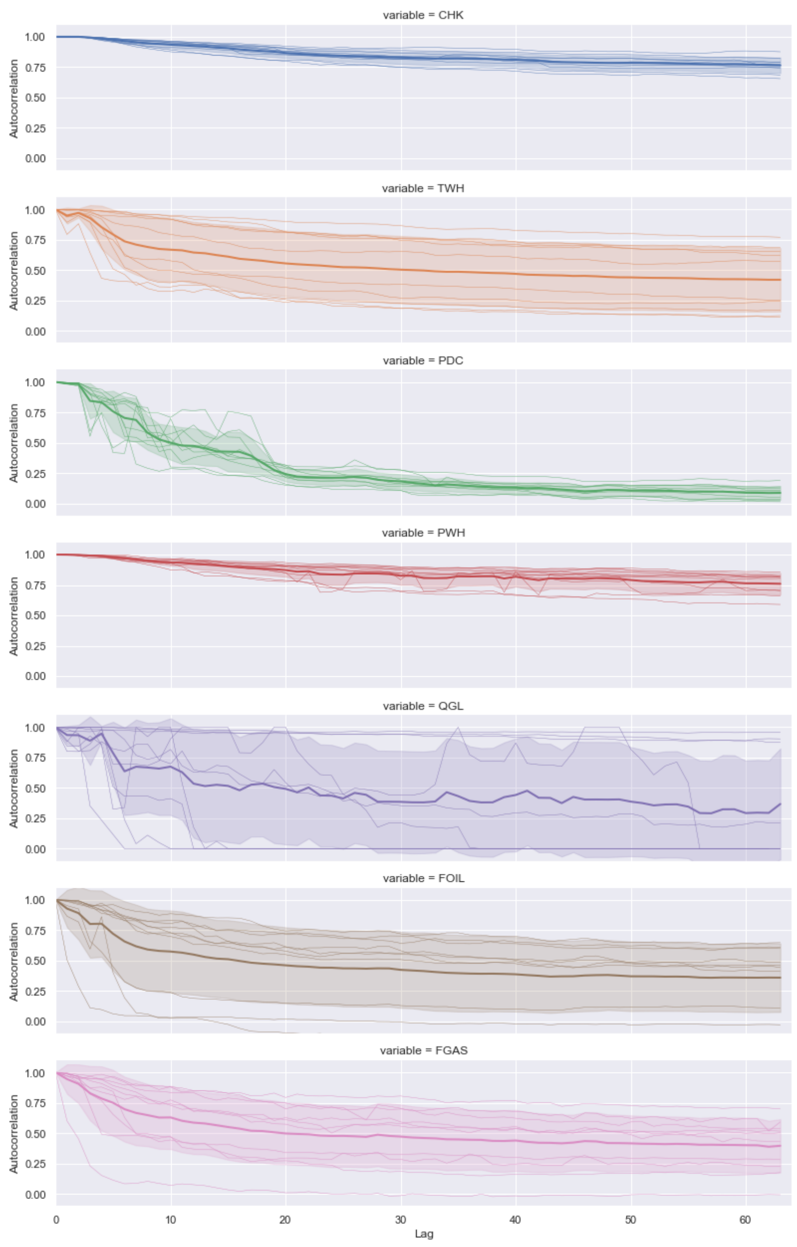


Figure 1: Lagged autocorrelations per feature. The mean across all wells is outlined as a thick center line, with thin lines represent the per-feature lagged autocorrelation for a single well. The colored area denote the respective standard deviations, centered around the per-feature mean.

B First run mean performance of architectures

Table 1: Mean validation and test MSE and runtime for each model architecture in the first run of experiments.

Architecture	Mean val MSE	Mean test MSE	Mean runtime
MTL-ONLY	0.848140	2.474796	0h 19m 54s
FCN-SEQ	1.363309	3.595114	0h 40m 7s
RNN	3.656395	9.281253	1h 35m 13s
LSTM	3.824773	12.462351	2h 13m 58s
GRU	3.723847	11.276725	2h 3m 0s
TCN	3.753602	6.643850	2h 45m 10s

C Overview of all trained models

Table 2: Overview of all MTL-ONLY model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	monitoring_time	val_loss	test_loss
MTL-ONLY	V1	0.00743	0.00010	[128] * 2	1194	0.84814	2.47480
MTL-ONLY	V2	0.00680	0.00017	[80] * 3	2221	0.95057	3.12063
MTL-ONLY	V3	0.00262	0.00015	[48] * 3	2223	0.94426	2.81401
MTL-ONLY	V4	0.00970	0.00029	[96] * 2	2478	0.95717	2.70075
MTL-ONLY	V5	0.00304	0.00019	[128] * 3	2302	0.92008	3.88521

Table 3: Overview of all FCN-SEQ model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	monitoring_time	val_loss	test_loss
FCN-SEQ-3	V1	0.00586	0.00017	[96] * 2	2439	1.08612	2.91600
FCN-SEQ-3	V2	0.00673	0.00011	[80] * 2	2608	0.99629	2.22589
FCN-SEQ-3	V3	0.00856	0.00027	[128] * 2	2717	1.16710	2.50964
FCN-SEQ-3	V4	0.00540	0.00015	[48] * 2	2270	1.15891	3.58678
FCN-SEQ-3	V5	0.00961	0.00013	[128] * 2	2633	1.10120	2.48423
FCN-SEQ-10	V1	0.00955	0.00066	[48] * 1	2139	1.53850	3.19229
FCN-SEQ-10	V2	0.00713	0.00010	[96] * 1	2268	1.54256	4.81772
FCN-SEQ-10	V3	0.00752	0.00012	[64] * 1	2465	1.62785	4.58762
FCN-SEQ-10	V4	0.00990	0.00070	[128] * 1	2467	1.59877	3.93214
FCN-SEQ-10	V5	0.00622	0.00012	[48] * 1	2460	1.65726	3.89256
FCN-SEQ-30	V1	0.00930	0.00011	[80] * 1	2644	1.46530	4.67705
FCN-SEQ-30	V2	0.00928	0.00018	[96] * 1	2659	1.47817	4.18882
FCN-SEQ-30	V3	0.00935	0.00027	[48] * 1	2736	1.43727	4.48242
FCN-SEQ-30	V4	0.00926	0.00013	[96] * 1	2743	1.50346	3.57156
FCN-SEQ-30	V5	0.00984	0.00013	[80] * 1	2894	1.60722	5.14821

Table 4: Overview of all RNN model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	recurrent_layers	hidden_size	monitoring_time	val_loss	test_loss
RNN-1D-32	V1	0.00114	0.00168	[64] * 2	1	64	3526	0.99822	7.14770
RNN-1D-32	V2	0.00144	0.00010	[48] * 2	1	64	4169	1.01144	7.62580
RNN-1D-32	V3	0.00249	0.00012	[64] * 1	1	64	3691	0.91460	6.90777
RNN-1D-32	V4	0.00222	0.00012	[64] * 2	1	32	4541	0.99556	3.46571
RNN-1D-32	V5	0.00236	0.00016	[48] * 2	1	64	4594	1.12967	5.13221
RNN-1D-64	V1	0.00235	0.00016	[64] * 2	1	48	5382	0.73879	3.01908
RNN-1D-64	V2	0.00218	0.00013	[48] * 2	1	64	5418	0.99854	7.17545
RNN-1D-64	V3	0.00604	0.00017	[32] * 1	1	64	5415	1.04634	6.93510
RNN-1D-64	V4	0.00137	0.00123	[64] * 2	1	64	5579	1.01149	6.00007
RNN-1D-64	V5	0.00108	0.00219	[64] * 2	1	64	5144	1.01501	3.55340
RNN-1D-128	V1	0.00182	0.00024	[64] * 2	1	64	7693	0.94354	3.07283
RNN-1D-128	V2	0.00109	0.00234	[64] * 2	1	64	6884	0.93644	5.35535
RNN-1D-128	V3	0.00183	0.00021	[48] * 2	1	32	7504	1.00357	5.57804
RNN-1D-128	V4	0.00181	0.00019	[64] * 2	1	64	6963	0.77218	4.27909
RNN-1D-128	V5	0.00154	0.00179	[64] * 2	1	64	8343	0.98936	2.82934
RNN-10D-32	V1	0.00249	0.00012	[64] * 2	1	64	4399	4.90305	12.14026
RNN-10D-32	V2	0.00286	0.00011	[64] * 2	1	48	3746	4.87576	14.45871
RNN-10D-32	V3	0.00884	0.00018	[32] * 1	1	32	4472	5.16788	11.48673
RNN-10D-32	V4	0.00510	0.00407	[48] * 1	1	48	4637	5.10060	14.70314
RNN-10D-32	V5	0.00110	0.00011	[48] * 2	1	32	4688	4.92357	9.58034
RNN-10D-64	V1	0.00312	0.00012	[48] * 2	1	32	5673	2.42332	7.07640
RNN-10D-64	V2	0.00383	0.00011	[48] * 2	1	32	5626	2.23635	8.27060
RNN-10D-64	V3	0.00131	0.00022	[64] * 1	1	32	4920	2.02400	6.97323
RNN-10D-64	V4	0.00121	0.00032	[64] * 2	1	32	5575	2.35786	7.19928
RNN-10D-64	V5	0.00125	0.00011	[32] * 2	1	64	5843	2.32396	7.32456
RNN-10D-128	V1	0.00149	0.00015	[48] * 1	1	64	7837	1.18044	9.20789
RNN-10D-128	V2	0.00186	0.00057	[32] * 2	1	32	7314	0.95278	2.67073
RNN-10D-128	V3	0.00145	0.00012	[64] * 1	1	64	7799	1.08059	3.62224
RNN-10D-128	V4	0.00306	0.00010	[64] * 2	1	64	6930	1.13226	3.35524
RNN-10D-128	V5	0.00324	0.00093	[64] * 1	1	64	7418	1.23399	5.21947
RNN-30D-32	V1	0.00885	0.00072	[48] * 2	1	32	3899	11.48474	15.66988
RNN-30D-32	V2	0.00867	0.00012	[32] * 1	1	48	3629	11.97072	16.72535
RNN-30D-32	V3	0.00657	0.00012	[48] * 2	1	64	3980	12.17525	18.32551
RNN-30D-32	V4	0.00806	0.00528	[64] * 1	1	48	4190	12.17135	21.03477
RNN-30D-32	V5	0.00815	0.00137	[48] * 2	1	32	3898	12.02496	15.45741
RNN-30D-64	V1	0.00319	0.00066	[48] * 2	1	32	5140	7.25180	15.58243
RNN-30D-64	V2	0.00645	0.00036	[64] * 1	1	48	5086	7.38295	16.07423
RNN-30D-64	V3	0.00624	0.00062	[48] * 1	1	64	5157	7.12717	12.41104
RNN-30D-64	V4	0.00291	0.00113	[64] * 2	1	64	4966	7.12972	11.65067
RNN-30D-64	V5	0.00110	0.00098	[64] * 1	1	48	5067	6.99315	13.92848
RNN-30D-128	V1	0.00368	0.00014	[32] * 2	1	48	7874	2.98365	10.61480
RNN-30D-128	V2	0.00246	0.00035	[48] * 1	1	48	7555	3.44000	12.14915
RNN-30D-128	V3	0.00218	0.00014	[64] * 2	1	48	7817	3.07770	10.49113
RNN-30D-128	V4	0.00492	0.00085	[64] * 2	1	48	7631	3.31185	9.83414
RNN-30D-128	V5	0.00214	0.00038	[64] * 2	1	48	7160	3.54586	10.03065

Table 5: Overview of all LSTM model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	recurrent_layers	hidden_size	monitoring_time	val_loss	test_loss
LSTM-1D-32	V1	0.00140	0.00012	[64] * 2	3	32	5715	1.05156	3.03612
LSTM-1D-32	V2	0.00219	0.00082	[64] * 2	3	64	6460	0.99299	11.76018
LSTM-1D-32	V3	0.00188	0.00014	[64] * 1	3	64	7706	0.97756	5.63495
LSTM-1D-32	V4	0.00155	0.00021	[64] * 1	2	48	6334	1.04544	6.35382
LSTM-1D-32	V5	0.00125	0.00017	[32] * 2	3	64	6420	1.05900	7.84008
LSTM-1D-64	V1	0.00220	0.00011	[48] * 2	3	32	9103	0.92354	3.80559
LSTM-1D-64	V2	0.00208	0.00018	[48] * 2	3	48	11227	0.91111	5.52252
LSTM-1D-64	V3	0.00211	0.00035	[64] * 2	2	64	10534	0.99337	3.93275
LSTM-1D-64	V4	0.00376	0.00011	[64] * 1	2	32	7905	0.91326	3.38636
LSTM-1D-64	V5	0.00423	0.00015	[48] * 1	1	64	7929	1.02069	8.79916
LSTM-1D-128	V1	0.00302	0.00045	[64] * 1	2	32	10744	0.99222	11.37327
LSTM-1D-128	V2	0.00180	0.00014	[32] * 2	3	64	16961	1.05121	4.32897
LSTM-1D-128	V3	0.00371	0.00012	[64] * 2	2	64	17448	0.99510	11.85775
LSTM-1D-128	V4	0.00213	0.00016	[64] * 2	1	64	11255	0.92657	6.15605
LSTM-1D-128	V5	0.00199	0.00110	[64] * 2	3	48	14139	0.89549	9.34414
LSTM-10D-32	V1	0.00582	0.00188	[64] * 1	1	48	3810	5.23352	21.53568
LSTM-10D-32	V2	0.00380	0.00193	[48] * 2	1	48	4764	5.44962	15.57235
LSTM-10D-32	V3	0.00941	0.00097	[32] * 2	1	32	4509	4.93445	12.23008
LSTM-10D-32	V4	0.00773	0.00061	[64] * 2	1	64	4799	5.07765	14.20209
LSTM-10D-32	V5	0.00680	0.00224	[48] * 2	1	32	4252	4.92926	11.31617
LSTM-10D-64	V1	0.00293	0.00022	[64] * 2	1	48	5738	2.74896	13.76241
LSTM-10D-64	V2	0.00851	0.00043	[64] * 1	1	32	6565	2.47306	10.10830
LSTM-10D-64	V3	0.00774	0.00119	[32] * 1	1	64	6584	2.30388	7.73100
LSTM-10D-64	V4	0.00610	0.00113	[32] * 2	1	32	5620	2.37235	10.33449
LSTM-10D-64	V5	0.00737	0.00013	[32] * 2	1	64	6778	2.35939	12.01302
LSTM-10D-128	V1	0.00830	0.00013	[64] * 1	1	64	11569	1.27024	5.18880
LSTM-10D-128	V2	0.00369	0.00016	[48] * 2	3	48	10460	1.24240	5.51504
LSTM-10D-128	V3	0.00230	0.00023	[64] * 1	2	48	15152	1.14721	11.93194
LSTM-10D-128	V4	0.00281	0.00155	[64] * 2	1	48	8264	1.28853	7.92240
LSTM-10D-128	V5	0.00463	0.00130	[48] * 2	1	48	9851	1.26025	8.67977
LSTM-30D-32	V1	0.00364	0.00042	[48] * 1	2	48	5053	11.32110	18.68483
LSTM-30D-32	V2	0.00393	0.00698	[64] * 2	2	32	4917	11.23395	24.87073
LSTM-30D-32	V3	0.00643	0.00291	[32] * 2	2	48	5840	11.55937	23.37901
LSTM-30D-32	V4	0.00254	0.00108	[48] * 2	1	32	4757	11.40361	15.78658
LSTM-30D-32	V5	0.00530	0.00092	[32] * 1	2	48	5807	11.78141	24.36311
LSTM-30D-64	V1	0.00145	0.00192	[48] * 2	3	32	8839	7.19580	17.67182
LSTM-30D-64	V2	0.00744	0.00306	[32] * 2	1	64	5887	7.10082	14.75125
LSTM-30D-64	V3	0.00166	0.00119	[64] * 2	3	48	7697	7.34726	12.75101
LSTM-30D-64	V4	0.00383	0.00587	[48] * 2	3	32	7436	7.15225	11.73970
LSTM-30D-64	V5	0.00543	0.00400	[32] * 2	1	32	5521	7.05736	17.90078
LSTM-30D-128	V1	0.00703	0.00644	[32] * 2	1	32	11779	3.68602	17.10265
LSTM-30D-128	V2	0.00644	0.00018	[48] * 1	1	32	10537	3.72086	12.82514
LSTM-30D-128	V3	0.00555	0.00067	[48] * 1	1	64	9913	3.75429	16.35171
LSTM-30D-128	V4	0.00689	0.00020	[32] * 2	2	32	11707	3.67226	11.23481
LSTM-30D-128	V5	0.00820	0.00016	[32] * 1	1	48	9687	3.69600	12.24094

Table 6: Overview of all GRU model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	recurrent_layers	hidden_size	monitoring_time	val_loss	test_loss
GRU-1D-32	V1	0.00152	0.00012	[48] * 2	1	32	4685	1.03657	4.00056
GRU-1D-64	V1	0.00235	0.00011	[48] * 2	2	64	6971	0.95851	7.82788
GRU-1D-128	V1	0.00335	0.00045	[48] * 2	2	48	11699	0.98059	5.17498
GRU-10D-32	V1	0.00457	0.00070	[64] * 1	1	32	3841	4.68497	14.72296
GRU-10D-64	V1	0.00257	0.00018	[48] * 1	1	48	5975	2.54789	7.96057
GRU-10D-128	V1	0.00512	0.00125	[48] * 1	2	64	10931	1.11505	12.11090
GRU-30D-32	V1	0.00177	0.00076	[64] * 1	1	32	3478	11.45615	19.79835
GRU-30D-64	V1	0.00177	0.00019	[64] * 1	1	32	5253	7.01634	13.53220
GRU-30D-128	V1	0.00498	0.00418	[32] * 2	2	48	13591	3.71855	16.36212

Table 7: Overview of all TCN model runs.

arch_id	group	learning_rate	l2_reg	fcn_hidden_dims	num_channels	kernel_size	monitoring_time	val_loss	test_loss
TCN-1D-32	V1	0.00664	0.00021	[48] * 1	[32] * 3	4	7480	0.95517	1.99000
TCN-1D-32	V2	0.00424	0.00016	[48] * 2	[16] * 2	7	6748	1.03001	2.72778
TCN-1D-32	V3	0.00340	0.00020	[48] * 2	[24] * 3	6	7589	1.01230	2.11908
TCN-1D-32	V4	0.00493	0.00016	[64] * 1	[32] * 3	6	7543	1.00593	3.60888
TCN-1D-32	V5	0.00438	0.00079	[64] * 2	[24] * 3	4	7674	1.03144	3.48363
TCN-1D-64	V1	0.00588	0.00011	[48] * 1	[24] * 3	7	7717	0.93594	2.38162
TCN-1D-64	V2	0.00266	0.00010	[64] * 2	[32] * 3	7	9131	0.92778	2.72765
TCN-1D-64	V3	0.00320	0.00014	[64] * 2	[24] * 3	7	8943	0.82222	3.73130
TCN-1D-64	V4	0.00724	0.00019	[32] * 1	[16] * 3	6	8300	1.02177	2.71433
TCN-1D-64	V5	0.00375	0.00045	[48] * 2	[32] * 5	3	9522	0.84035	3.34883
TCN-1D-128	V1	0.00677	0.00019	[48] * 2	[24] * 6	3	12723	1.00872	2.21166
TCN-1D-128	V2	0.00222	0.00042	[32] * 2	[32] * 6	3	9583	1.02633	2.88867
TCN-1D-128	V3	0.00388	0.00023	[64] * 2	[32] * 5	5	10091	0.90446	3.04900
TCN-1D-128	V4	0.00765	0.00011	[32] * 1	[32] * 5	4	13611	0.95551	3.82381
TCN-1D-128	V5	0.00968	0.00011	[64] * 2	[32] * 6	3	12602	0.95071	3.52397
TCN-10D-32	V1	0.00989	0.00013	[48] * 2	[16] * 3	4	7371	4.88199	8.37322
TCN-10D-32	V2	0.00992	0.00074	[64] * 1	[16] * 3	4	8736	5.13424	8.30860
TCN-10D-32	V3	0.00969	0.00027	[64] * 2	[8] * 4	3	7986	5.28428	8.75988
TCN-10D-32	V4	0.00834	0.00033	[32] * 2	[8] * 3	5	7304	4.91676	8.49383
TCN-10D-32	V5	0.00675	0.00070	[32] * 2	[8] * 3	4	7753	5.20103	8.78510
TCN-10D-64	V1	0.00272	0.00017	[48] * 2	[24] * 5	3	9714	2.53919	6.01715
TCN-10D-64	V2	0.00986	0.00011	[32] * 1	[24] * 5	3	9215	2.50196	5.05386
TCN-10D-64	V3	0.00420	0.00043	[64] * 2	[8] * 4	4	8877	2.47386	5.96419
TCN-10D-64	V4	0.00542	0.00013	[64] * 2	[16] * 4	5	9855	2.61531	7.68779
TCN-10D-64	V5	0.00612	0.00022	[48] * 2	[8] * 5	3	9887	2.58263	5.00588
TCN-10D-128	V1	0.00966	0.00029	[48] * 1	[32] * 5	5	11818	1.28239	4.67901
TCN-10D-128	V2	0.00970	0.00013	[48] * 2	[8] * 5	4	11451	1.09792	2.26833
TCN-10D-128	V3	0.00716	0.00010	[48] * 2	[24] * 4	6	13342	1.24608	3.57816
TCN-10D-128	V4	0.00603	0.00012	[64] * 2	[16] * 4	7	10718	1.20410	2.50251
TCN-10D-128	V5	0.00897	0.00012	[64] * 2	[24] * 4	7	13448	1.11054	2.84967
TCN-30D-32	V1	0.00589	0.00039	[64] * 2	[24] * 4	3	7754	11.77172	12.28644
TCN-30D-32	V2	0.00526	0.00318	[48] * 1	[24] * 3	5	7108	12.06616	15.68633
TCN-30D-32	V3	0.00684	0.00010	[48] * 2	[32] * 4	3	7799	11.49699	15.39291
TCN-30D-32	V4	0.00879	0.00190	[32] * 1	[24] * 4	3	8162	10.91885	14.09386
TCN-30D-32	V5	0.00834	0.00014	[64] * 1	[24] * 3	5	8239	11.42908	17.40408
TCN-30D-64	V1	0.00736	0.00014	[48] * 1	[16] * 5	3	10173	6.92589	12.72922
TCN-30D-64	V2	0.00624	0.00107	[64] * 1	[32] * 3	6	8453	7.43847	12.38099
TCN-30D-64	V3	0.00849	0.00012	[64] * 2	[32] * 4	4	9604	7.26051	11.73312
TCN-30D-64	V4	0.00971	0.00155	[48] * 1	[24] * 4	4	8900	6.98573	9.87738
TCN-30D-64	V5	0.00350	0.00229	[48] * 2	[16] * 5	3	9712	7.32371	10.55341
TCN-30D-128	V1	0.00765	0.00015	[32] * 1	[32] * 5	4	12822	3.48141	9.12632
TCN-30D-128	V2	0.00911	0.00041	[48] * 1	[32] * 6	3	13602	3.28898	7.32451
TCN-30D-128	V3	0.00538	0.00010	[64] * 2	[32] * 5	5	13970	3.27755	6.72480
TCN-30D-128	V4	0.00982	0.00049	[64] * 1	[24] * 5	4	13659	3.00928	7.19003
TCN-30D-128	V5	0.00915	0.00011	[32] * 2	[32] * 6	3	12462	3.19917	5.68047

D Best model predictions on all wells

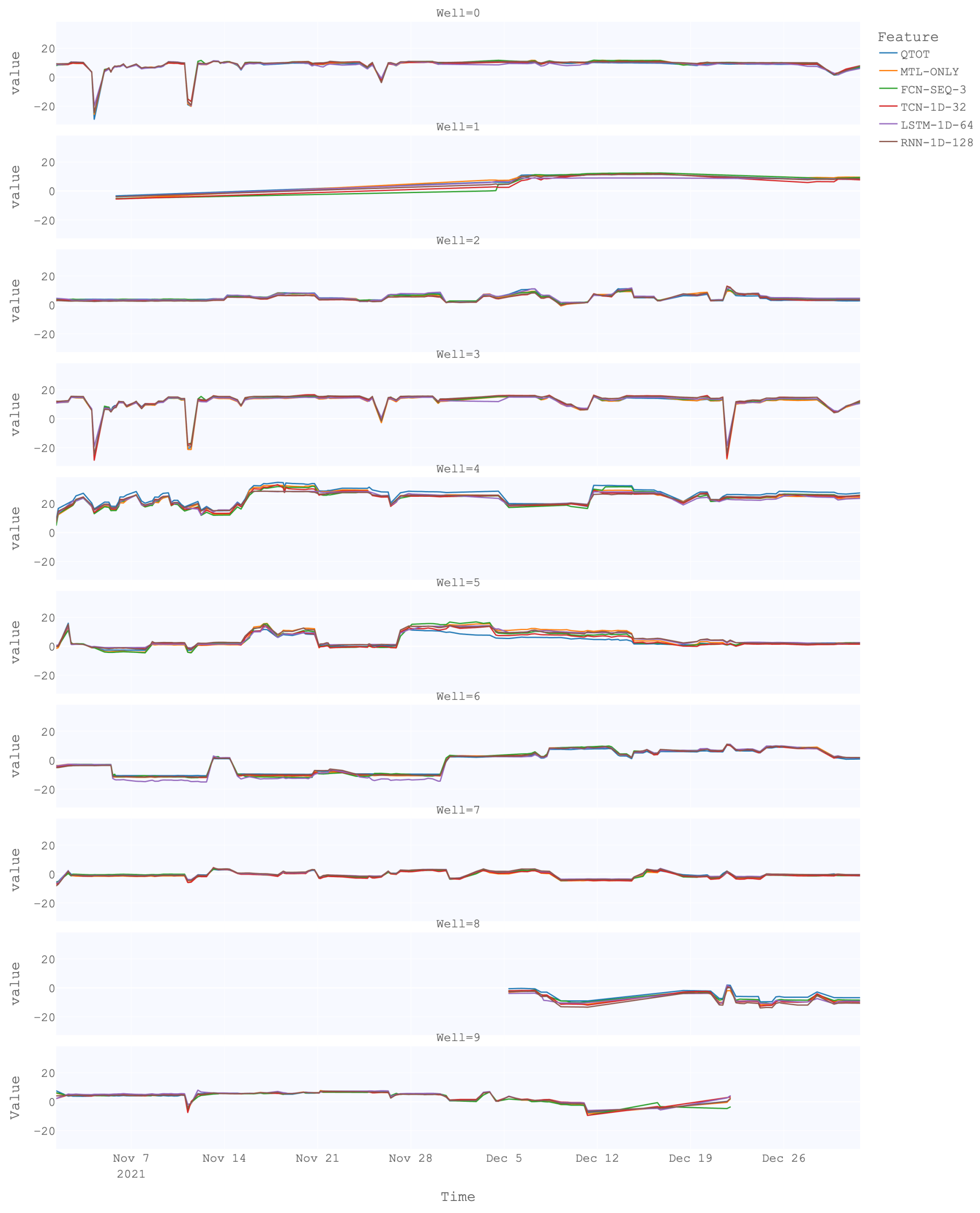


Figure 2: Illustration of flow rate estimates on test data for the models with lowest test MSE.

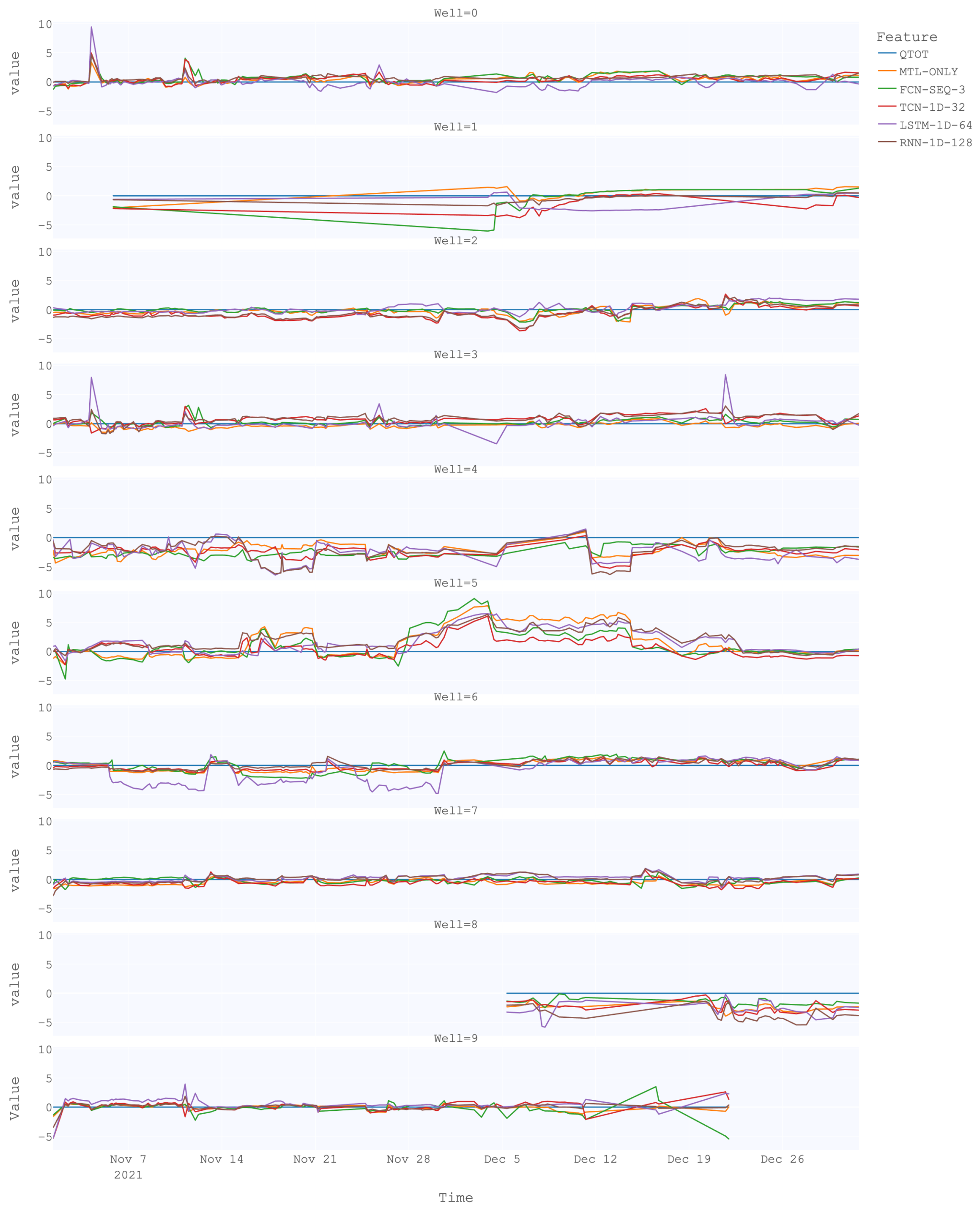


Figure 3: Illustration of errors in flow rate estimates on test data for the models with lowest test MSE.

E Losses when perturbing information in earliest sequence elements

Table 8: Average MSE test loss over five experiments for each of the best-performing sequence models when a fraction of the input sequence elements corresponding to the earliest timesteps are masked using different schemes. Starting from the oldest history, each degree of masking represents extending the mask to the next $\frac{1}{32}$ elements of the input sequence. Thus, a value of zero indicates no masking and a value of 31 indicate that only the most recent $\frac{1}{32}$ elements of every input sequence is left unmasked.

Degree of masking	RNN-1D-128			LSTM-1D-64			TCN-1D-32		
[n/32]	mean	slice	zero	mean	slice	zero	mean	slice	zero
0	4.2229	4.2229	4.2229	5.0893	5.0893	5.0893	2.7859	2.7859	2.7859
1	4.2229	4.2229	4.2229	4.7322	5.0882	4.7809	2.7861	2.7901	2.7887
2	4.2229	4.2229	4.2229	4.6685	5.0820	4.6690	2.7851	2.7927	2.7904
3	4.2229	4.2229	4.2229	4.5585	5.0782	4.5772	2.7880	2.7994	2.7954
4	4.2229	4.2229	4.2229	4.4398	5.0800	4.5314	2.7677	2.9237	2.8128
5	4.2229	4.2229	4.2229	4.3718	5.0858	4.5104	2.7776	2.9298	2.8301
6	4.2229	4.2229	4.2229	4.3210	5.0938	4.4952	2.7843	2.9245	2.8436
7	4.2229	4.2229	4.2229	4.2544	5.1031	4.4869	2.7914	2.9315	2.8551
8	4.2229	4.2229	4.2229	4.1983	5.1139	4.4872	2.7606	2.9695	2.8256
9	4.2229	4.2229	4.2229	4.1609	5.1292	4.4982	2.7754	2.9715	2.8353
10	4.2229	4.2229	4.2229	4.1404	5.1544	4.5215	2.7568	2.9050	2.8156
11	4.2229	4.2229	4.2229	4.1335	5.1870	4.5578	2.7665	2.8996	2.8284
12	4.2229	4.2229	4.2229	4.1378	5.2221	4.5999	2.6749	2.9489	2.7653
13	4.2229	4.2229	4.2229	4.1441	5.2599	4.6448	2.6957	2.9470	2.7826
14	4.2229	4.2229	4.2229	4.1575	5.2876	4.6838	2.6845	2.9041	2.7886
15	4.2229	4.2229	4.2229	4.1796	5.3173	4.7313	2.7024	2.8919	2.8014
16	4.2229	4.2229	4.2229	4.2059	5.3547	4.7831	2.7126	2.9231	2.8142
17	4.2229	4.2229	4.2229	4.2348	5.3954	4.8282	2.7249	2.9223	2.8206
18	4.2229	4.2229	4.2230	4.2669	5.4403	4.8658	2.7165	3.3479	3.0271
19	4.2229	4.2229	4.2229	4.3112	5.4846	4.9157	2.7231	3.3495	3.0490
20	4.2229	4.2229	4.2229	4.3723	5.5288	4.9748	2.7712	3.6527	3.2868
21	4.2229	4.2229	4.2229	4.4469	5.5845	5.0522	2.7715	3.6555	3.2617
22	4.2229	4.2229	4.2230	4.5012	5.6394	5.1516	2.6745	3.2056	3.0373
23	4.2229	4.2229	4.2230	4.5950	5.7013	5.2626	2.6859	3.3690	3.0741
24	4.2230	4.2229	4.2232	4.7127	5.7948	5.3656	2.5706	3.3681	2.9601
25	4.2230	4.2228	4.2234	4.8492	5.9027	5.4942	2.6600	3.8346	3.1365
26	4.2229	4.2227	4.2234	5.0264	6.0143	5.6590	2.6756	4.5710	3.7171
27	4.2228	4.2223	4.2229	5.2736	6.1511	5.8740	2.8541	5.8256	5.1010
28	4.2246	4.2228	4.2249	5.7179	6.4217	6.2307	2.8763	7.6002	6.7356
29	4.2287	4.2268	4.2336	6.6828	7.1416	7.0539	3.0809	7.6482	6.9343
30	4.2323	4.2345	4.2454	9.3719	9.6159	10.6925	3.3408	8.1033	7.4320
31	4.5079	4.6419	4.7215	19.6239	21.3306	37.2753	4.2545	7.9559	7.9780

F Best model predictions on resampled test data for all wells

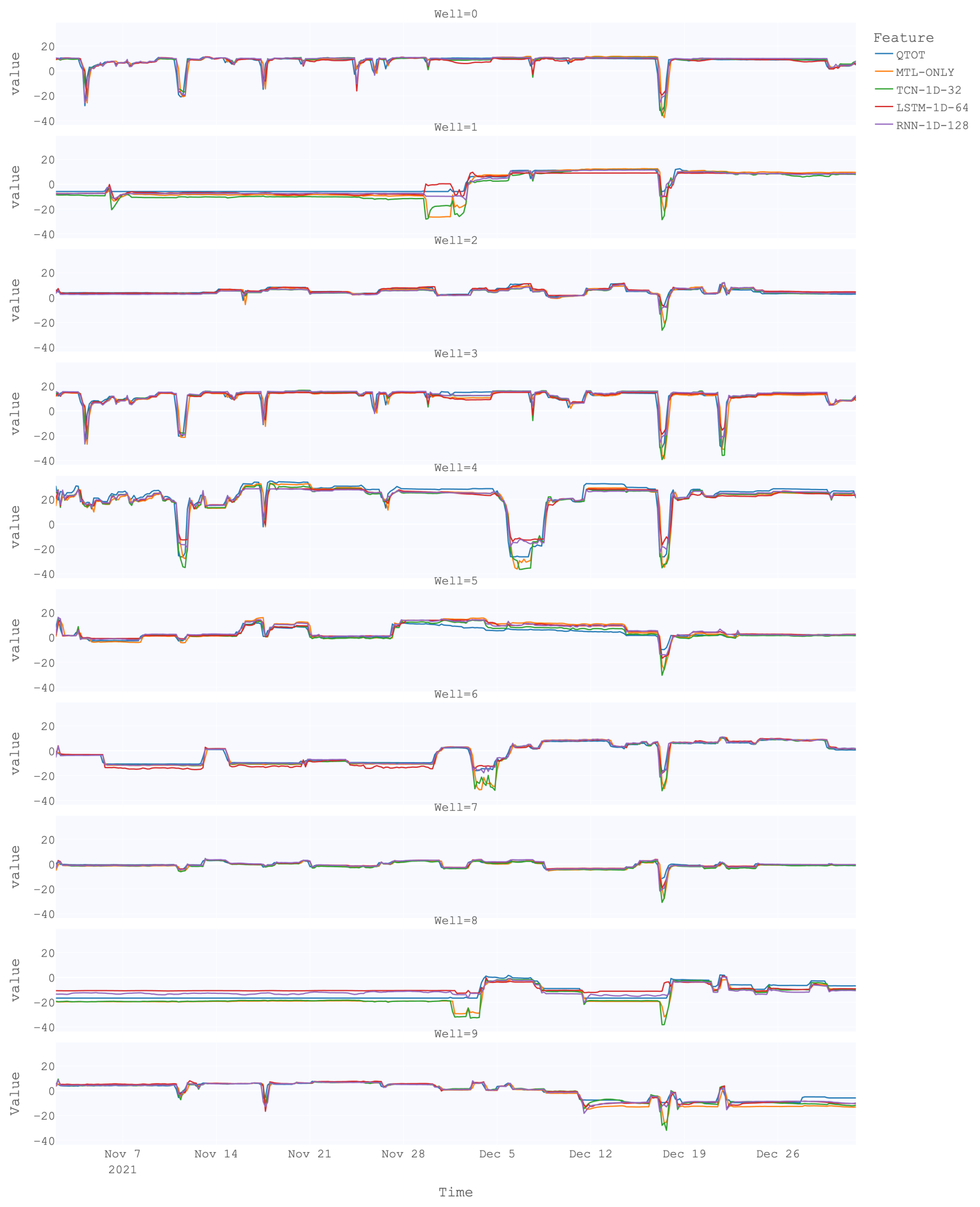


Figure 4: Illustration of flow rate estimates on the unfiltered test dataset uniformly resampled to four-hour periods for the models with lowest test MSE.

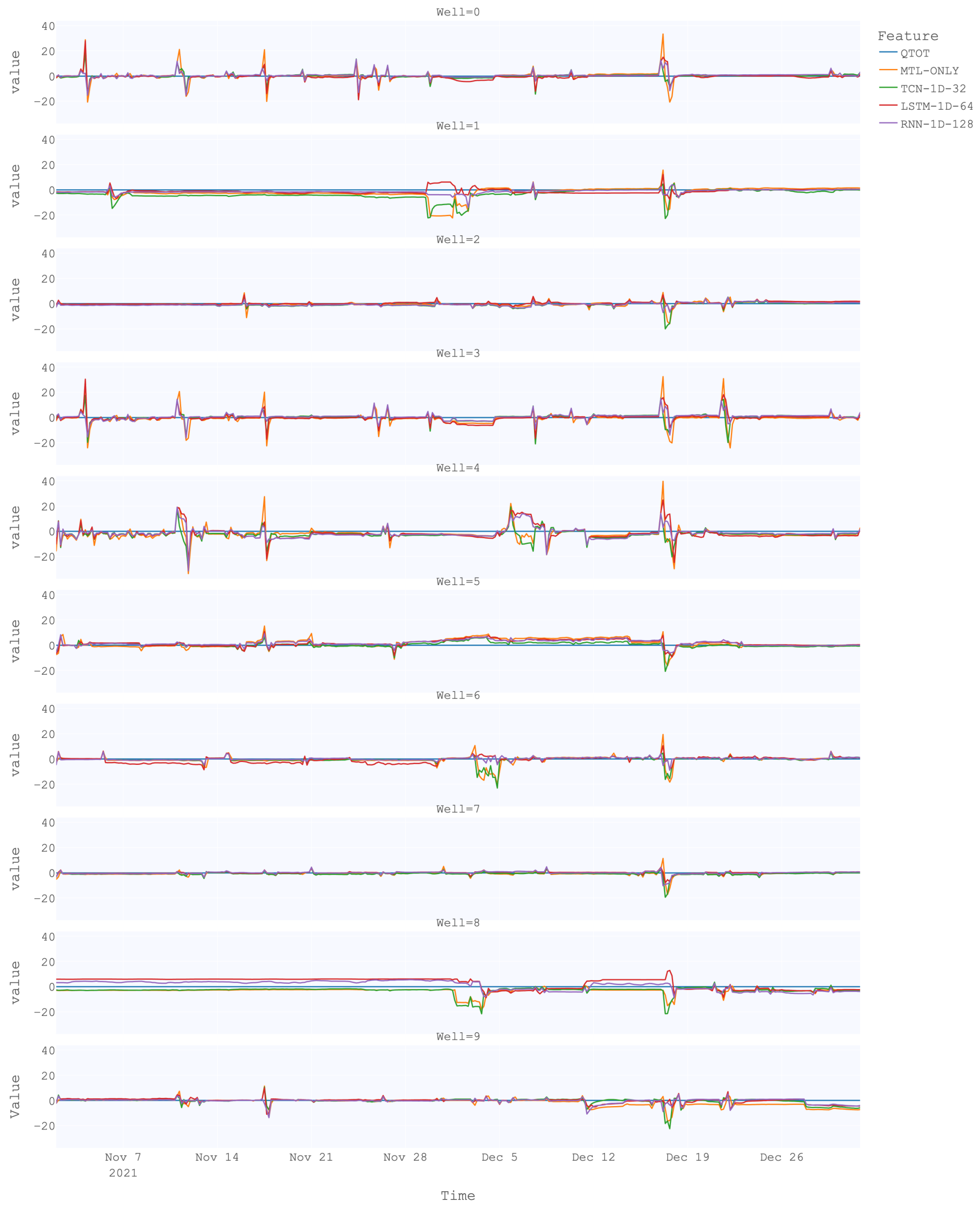


Figure 5: Illustration of errors in flow rate estimates on the unfiltered test dataset uniformly resampled to four-hour periods for the models with lowest test MSE.

Morgan Heggland & Patrik Kjærran

An exploration of sequence models using multi-task learning for multiphase flow rate estimation in oil and gas wells

NTNU
Kunnskap for en bedre verden

Solution
Seeker