Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Machine Learning Prediction of Autonomous Inflow Control Device (AICD) Performance Curves

Stian Dean Howard

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Machine Learning Prediction of Autonomous Inflow Control Device (AICD) Performance Curves

Author:	Stian Dean Howard
Supervisor:	PD Dr.-Ing. habil. Xiangyu Hu
Partner:	Morten H. Jondahl (Ranold A.S.)
Submission Date:	December 8th, 2022

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

December 8th, 2022 Stian Dean Howard

Acknowledgments

I would like to express my sincere gratitude to all the friends, family, and colleagues who supported me throughout the writing of this thesis.

I would especially like to thank:

Xiangyu Hu, for taking on the role of supervisor and examiner for this thesis.

Morten Jondahl, for sharing his resources and knowledge with me. Without Morten, this project could not have happened.

Tendeka, for generously providing the data that is central to the thesis.

And last, but by no means least, Siv Howard, my mother, for providing me with invaluable feedback and advice despite her busy travel schedule.

Abstract

Inflow control devices (ICDs) are valves placed in oil and gas wells to control the flow of fluids from the reservoir into the well. The newest generations of ICDs, autonomous inflow control devices (AICDs), adjust their physical geometry according to the characteristics of fluids passing through them, thereby reducing the flow of undesired fluids into the well. Simulators need accurate, high-performance mathematical models to predict how AICDs perform under various fluid and flow conditions. The oil and gas industry has largely settled on using an AICD equation, which was designed for Equinor's rate-controlled production valve, that is fitted to each unique AICD technology. While this AICD equation effectively predicts trends seen in some AICD performance curves, it is incapable of accurately adapting to new AICD technologies with novel performance characteristics.

Machine learning models are intuitive alternatives to the AICD equation, providing built-in adaptability to future technologies. This thesis presents a machine learning model that predicts AICD performance with greater accuracy than the industry standard AICD equation. The model is versatile and therefore not restricted to one specific AICD technology. The benefit of using transfer learning to compensate for small data sets is also tested by training one model with simulated data before the experimental data and another with only the experimental data. A model constituting a fitted AICD equation and the two machine learning models, one with and one without transfer learning, are tested with two experimental data sets for Tendeka's FloSure AICD technology. All three models are also tested on the same data sets with 70% of the data randomly removed.

The machine learning models predicted FloSure's performance with greater accuracy than the fitted AICD equation on both tested data sets. Transfer learning, using data generated from the training data set, was found to be beneficial for all models tested, however, it was most effective on systems with larger data sets. At an average of 48 μs per prediction over 100,000 executions, the machine learning models were approximately 400–500 times slower than the AICD equation, which took on average 0.083 μs. Given adequate training data, machine learning models provide an improvement in the accuracy of predicted AICD performance curves and increase the generalizability of prediction models for future technologies. If the performance loss is acceptable for the simulator, machine learning models are good candidates for replacing the AICD equation as the industry standard.

Contents

Acknowledgements vii

Abstract ix

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Goal and Structure . . . 2

2 Overview of Pressure Curves and the AICD Equation 5
- 2.1 Performance Curves . . . 5
  - 2.1.1 Performance Curve Basics and Visualization . . . 5
  - 2.1.2 Performance Curves as Simulation Boundary Conditions . . . 7
- 2.2 AICD Equation . . . 8
  - 2.2.1 Origins of the AICD Equation . . . 8
  - 2.2.2 Optimizing the AICD Equation . . . 9
  - 2.2.3 Alternatives to the AICD Equation . . . 11

3 Introduction to Machine Learning 13
- 3.1 Deep Learning and Artificial Neural Networks . . . 13
  - 3.1.1 Artificial Neural Network Design and Function . . . 13
  - 3.1.2 Training an Artificial Neural Network . . . 17
  - 3.1.3 PyTorch Framework . . . 19
- 3.2 Machine Learning for Performance Curve Prediction . . . 20
  - 3.2.1 Data Limitations . . . 21
  - 3.2.2 Transfer Learning . . . 22
  - 3.2.3 K-Fold Cross Validation . . . 23

4 Methodology and Implementation 25
- 4.1 Experimental Design . . . 25
- 4.2 Data Acquisition and Processing . . . 26
  - 4.2.1 Data Requirements . . . 26
  - 4.2.2 Data Sources . . . 27
  - 4.2.3 Data Processing and Partitioning . . . 28

4.3 AICD Equation Implementation . . . 29
4.3.1 AICD Equation Selection and Optimization Strategy . . . 29
4.3.2 Code Implementation . . . 30
4.4 Machine Learning Implementations and Structure . . . 30
4.4.1 First Network for Baseline Training . . . 30
4.4.2 Second Network for Transfer Learning . . . 32
4.5 Performance Comparison and Timing . . . 34
4.5.1 Model Prediction Error . . . 36
4.5.2 C++ Implementation of the Models . . . 38
4.5.3 Timing Script . . . 41

5 Results and Conclusion **43**
5.1 Results . . . 43
5.1.1 Training Performance . . . 43
5.1.2 Model Error Analysis . . . 45
5.1.3 Model Runtimes . . . 46
5.2 Conclusion . . . 48

Bibliography **51**

1 Introduction

1.1 Context

The oil and gas industry is notable for large-scale investments and consequences; billions of dollars of investment can be required to construct wells to extract oil and gas. Mistakes can have severe consequences on the environment and field productivity and can result in expensive workover and cleanup operations. With these traits, it's understandable that the industry invests heavily in simulations to optimize operations and test new technologies that can improve safety, reduce risk, and increase productivity. Unfortunately, these large-scale consequences are also the reason why the industry is often reluctant to replace field-tested technologies without extensive simulations and trials in lower-risk scenarios.

While sub-sea oil and gas extraction is a process involving hundreds of moving parts, the 'below wellhead' systems can broadly be split into two areas: the well and the reservoir, each with separate simulation systems. An inflow control device (ICD), is a valve that joins the reservoir to the well. Originally, wells were designed with a simple 'open hole' aperture allowing fluids to flow freely into the well in what is known as barefoot, or 'open hole', completion. However, as technology and experience in the field progressed, it was found that more advanced devices could be used to extend the production life of a well and limit water or gas breakthroughs. The newest devices, autonomous inflow control devices (AICDs), are capable of changing their characteristics autonomously according to the characteristics of the fluids flowing through them and thereby allowing primarily the desired oil or gas to be extracted.

For these devices to be correctly simulated in well and reservoir simulators, a mathematical model is needed to predict the behavior of the AICD. Considering that these models need to be accessed many times over the course of a simulation, a complete 3-phase computational fluid dynamics simulation is far too computationally expensive to employ. To quickly predict the performance of an AICD for any arbitrary combination of input fluid characteristics across the device, the industry has largely settled on the AICD equation. The AICD equation needs to be uniquely optimized using experimental flow measurements collected in large test rigs capable of replicating fluid and environmental conditions of a well in a reservoir. However, despite optimizing the ACID equation with experimental data, the AICD equation is inaccurate and incapable of adapting to new designs and technologies.

1.2 Motivation

The newest generations of ICDs are becoming increasingly advanced and have flow behaviors that differ substantially from typical flow seen in older generations. Some of the newer AICDs, such as Tendeka's FloSure system and Innowell's DAR system, have geometries that physically change according to the characteristics of the fluids flowing through them. In the case of valves that are physically 'closed' or 'open' according to fluid characteristics, there is a potential to see discontinuities in their flow performance curves. The AICD equation, and the direct variations based on it, are continuous equations incapable of dealing with discontinuous or unusual flow properties that do not follow the typical shape of the equation.

These challenges lead to a natural incentive to create a new system that is capable of more accurately predicting the performance of tomorrow's AICDs. Machine learning is an obvious choice of a potential replacement model. Notably, most of the required training data is already available as valve manufacturers perform simulations and retrieve test rig measurements during development. Additionally, machine learning models can provide a 'black box' which can be altered by each manufacturer to better suit their individual system while still providing out-of-the-box flexibility and generalizability. Ideally, a machine learning model would be capable of being trained on any AICD's performance curve regardless of the profile shape or technology and be able to provide predicted AICD performance to any system that provides its required variables.

1.3 Goal and Structure

The goal of the project is to create a machine learning model that:

- can be trained on the performance curve of an AICD
- outperforms the standard AICD equation in predicting the AICD performance curve
- is system agnostic (i.e. can predict the performance of any AICD regardless of its technology)
- is performant enough that it can be employed in reservoir and well simulators[1]
- is simple enough that an engineer with little computer science or machine learning experience can train and distribute ready-to-use

This thesis will first provide an overview of performance curves and the industry-accepted AICD equation followed by an examination of the components constituting the AICD

[1] Computational performance requirements for AICD performance curve prediction varies by simulator and use case. For this project, performance will be judged with the assumption that the prediction model will be called hundreds of thousands to millions of times over the space of several minutes or hours.

equation, and how it is used to model and predict AICD performance. After, a brief look will be taken at machine learning focusing on the aspects that influence model design in this application, as well as some of the larger restrictions imposed by machine learning and how they can be dealt with. Finally, two machine learning models will be built, trained on real AICD performance curves, and compared to an implementation of the AICD equation.

2 Overview of Pressure Curves and the AICD Equation

2.1 Performance Curves

The following section will cover the theory of AICD performance curves, what they are, and how they're visualized. A brief look at performance curves as boundary conditions for simulators will also be included.

2.1.1 Performance Curve Basics and Visualization

An AICD performance curve characterizes the relationship between the fluid flow rate through an AICD and the pressure drop across it. The curve typically bears some similarity to the quadratic curve observed with flow through an orifice or a fixed geometry Inflow Control Device (ICD) [12]. Orifice flow for incompressible fluids is defined by Equation 2.1 (assuming very large diameter at the inflow), where ρ is fluid density, C_d is the geometry specific discharge coefficient, A_o is the cross-sectional area of the orifice, Q is fluid flow rate, and ΔP is the pressure drop across the valve.

$$\Delta P = \frac{\rho}{2C_d^2 A_o^2} Q^2 \tag{2.1}$$

This is a logical influence since AICDs are essentially ICDs with non-fixed geometry allowing the valve to control flow in a more aggressive and controlled manner. Figure 2.1 plots the performance curve for a fluid flowing through Tendeka's FloSure AICD[1], and the orifice flow equation optimized to best match the AICD performance curve. The two curves do follow the same general trend, but the AICD has a distinct curvature that does not match the quadratic function.

This performance plot is just a slice of the actual performance curve of interest for an AICD. Performance curves actually span multiple dimensions as a hyperplane, not just the two dimensions observed in Figure 2.1. The pressure drop over an AICD depends on additional fluid characteristics, such as the viscosity and density of component fluids, and not

[1] Tendeka generously provided 2 data sets of AICD performance curves for this thesis.

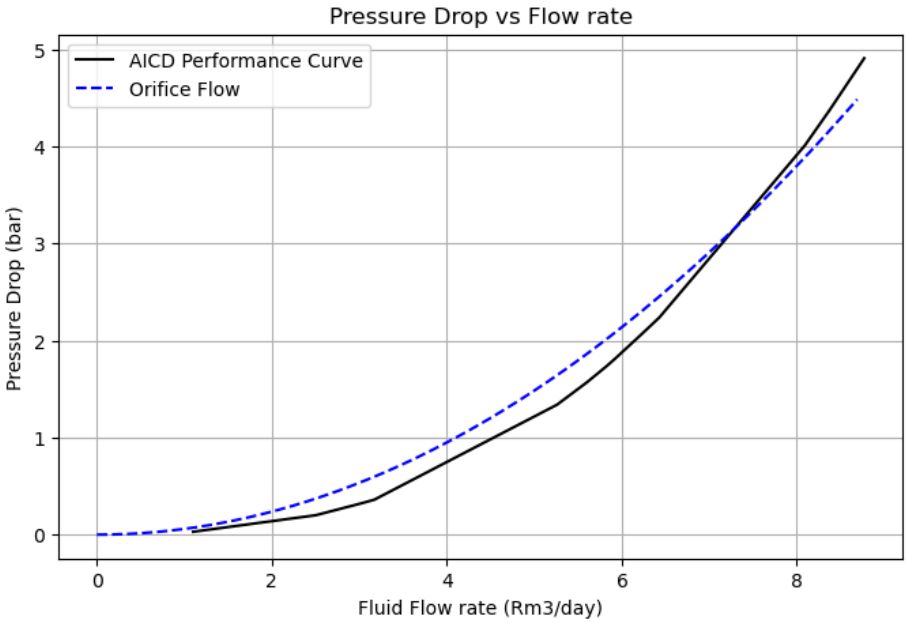

Figure 2.1: Plot showing Tendeka's FloSure AICD performance curve, and the incompressible orifice flow that best fits the AICD performance curve.

just the flow rate through the valve. Whether the fluids separate, form an emulsion, or mix perfectly, along with fluid compressibility also impact the performance of the AICD. In the most precise of descriptions, one would also account for the oil potentially passing through its bubble point, where some of the natural gasses dissolved in the oil form gas bubbles, again altering the characteristics of the emulsion. For the studied use case as a boundary condition in simulators, the performance curve is often simplified to a 4 dimensional hyperplane defining pressure drop as a function of flow rate, oil fraction, water fraction and gas fraction.

There is no method to directly visualize this relationship in two or three dimensional plots, however by using color some of the additional information can be conveyed. In this paper a direct correlation of oil, gas, and water volume fractions, $\alpha_o, \alpha_g, \alpha_w$, to RGB values using Equation 2.2, will be used.

$$\begin{aligned} R &= \lfloor \alpha_o * 255 + \frac{1}{2} \rfloor \\ G &= \lfloor \alpha_g * 255 + \frac{1}{2} \rfloor \\ B &= \lfloor \alpha_b * 255 + \frac{1}{2} \rfloor \end{aligned} \tag{2.2}$$

Using this, a performance curve spanning more fluid combinations can be visualized in a

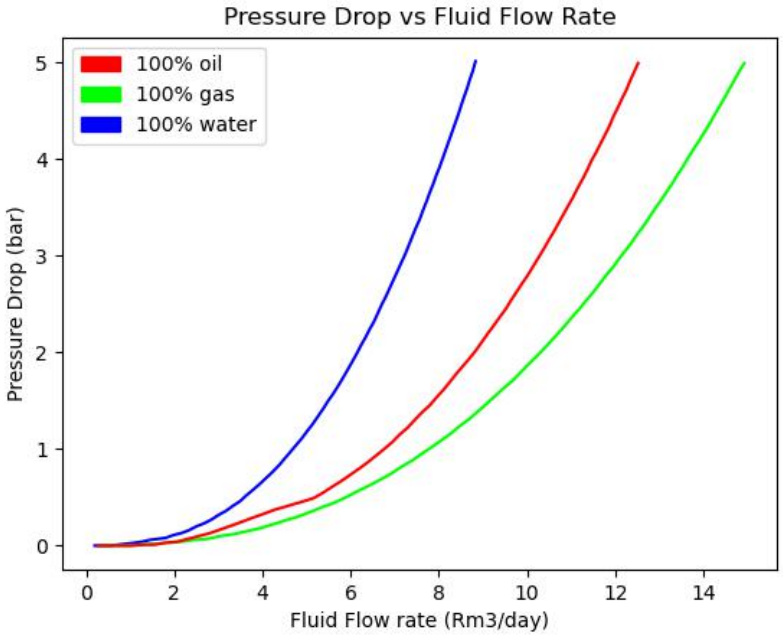

Figure 2.2: Example plot of an AICD performance curve, for Tendeka's FloSure technology, with pure oil, water, and gas on one plot. Additional combinations can be created by using color mixtures of red, green and blue.

single two dimensional plot, an example of which is shown in Figure 2.2.

2.1.2 Performance Curves as Simulation Boundary Conditions

Simulations of a well, below the wellhead, are largely separated into two broad categories: the well and the reservoir. Well simulations aim to examine what happens from when fluids enter the well itself, through an ICD or AICD if one is installed, and up to the wellhead. These simulations have a variety of uses, including ICD selection for long-term oil and gas production, and calculating flow along the well and determining what sorts of additional assistance, such as gas lift, are required to maintain production. Reservoir simulations aim to predict the behavior of a reservoir over its production lifetime to optimize well completion and placement, calculating how oil, water, and gas move through the reservoir under various conditions. Reservoir simulator depend on having a fluid sink at the well inflow points to remove fluid from the reservoir and simulate how the reservoir behaves while oil is being extracted. For wells with an AICD included in the completion, the AICD becomes the fluid sink necessitating a high-performance method to compute performance curves.

Both of these two systems are dependent on inflow points interfacing between the reservoir and the well. In the ideal world, the two simulation categories would be combined into one, large, completion simulator capable of combining the complexities of reservoir simulations with production performance under various well configurations. Today, this is infeasible, since both simulations are highly computationally demanding on their own and would likely need a systematic re-design for their solvers to be compatible.

Considering these limitations, it is only feasible to consider the inflow point as the boundary condition between the two simulators: they are single spatially isolated points whose behavior can be reasonably modeled by local variables that are not dependent on gradients in the surrounding reservoir or well.

2.2 AICD Equation

A basic AICD equation can be written as

$$\begin{aligned} \Delta P &= \left(\frac{\rho_{mix}^2}{\rho_{cal}}\right)\left(\frac{\mu_{cal}}{\mu_{mix}}\right)^y A_{aicd} \cdot Q^x \\ \mu_{mix} &= \alpha_o \mu_o^a + \alpha_w \mu_w^b + \alpha_g \mu_g^c \\ \rho mix &= \alpha_o \rho_o^d + \alpha_w \rho_w^e + \alpha_g \rho_g^f, \end{aligned} \tag{2.3}$$

where $\rho_{cal}, \mu_{cal}, A_{aicd}, x, y, a, b, c, d, e$, and f are variables to be optimized and held constant once fitted to an AICD's performance curve [7]. Q is the fluid flow rate through the AICD, α is an in situ volumetric fluid fractions, ρ is fluid density, μ is fluid viscosity, and ΔP is the pressure drop across the AICD. The fluid flow rate, in situ volumetric fluid fractions, fluid densities, and fluid viscosities are the dependent variables used to calculate the pressure drop across the AICD. This equation, when fitted to an experimental data set defining an AICD's performance curve, allows for quick predictions of the performance curve without requiring time-consuming multi-phase fluid dynamics simulations.

2.2.1 Origins of the AICD Equation

The structure of equation 2.3 can be broadly broken into three components: the density term, the viscosity term, and the flow rate term. The x and y exponents, over the density and flow rate terms, allow the function to adapt to variations in how the AICD responds to changes in fluid density and viscosity. The additional exponents, a, b, c, d, e, and f, in the mix expressions, allow the function to tune the response to variations in specific fluid components in cases where the valve is capable of doing so. However, the function forces the density response into a square relationship, and thereby does not allow for easy tuning to an AICD with a density response outside of the specified quadratic relationship. The

reason for excluding density response tuning makes a little more sense when the origins of the equation are taken into perspective.

The AICD equation was developed for Equinor's rate-controlled production (RCP) valve in 2012 by *Halvorsen et al.*[6]. The RCP valve adjusts flow according to the viscosity of the mixed fluid passing through the valve by changing its internal geometry. The RCP valve, however, is not designed to similarly respond to variations in density, so the equation fixes the density response to a constant quadratic relationship that is accurate for the RCP valve. Additionally, Equinor treated the viscosity and density of all the component fluids equally, assuming perfect mixing, so the exponents a, b, c, d, e and f from Equation 2.3 are all equal to 1. This leaves only five optimizable terms to fit the function to a performance curve: $x, y, \mu_{cal}, \rho_{cal}$, and A_{aicd}, as shown in Equation 2.4.

$$\begin{aligned} \Delta P &= \left(\frac{\rho_{mix}^2}{\rho_{cal}}\right)\left(\frac{\mu_{cal}}{\mu_{mix}}\right)^y A_{aicd} \cdot Q^x \\ \mu_{mix} &= \alpha_o\mu_o + \alpha_w\mu_w + \alpha_g\mu_g \\ \rho mix &= \alpha_o\rho_o + \alpha_w\rho_w + \alpha_g\rho_g \end{aligned} \tag{2.4}$$

The design of this function is logical for predicting the performance curve of the RCP valve specifically. The predictions were validated against experimental tests of the AICD at Equinor's Porsgrunn multi-phase flow test laboratory [12].

This version of the AICD equation has gained large traction in simulators globally. Considering the function was designed to specifically predict the performance of a specific AICD, and the consequent limitations of this function, however, various minor variations have been made over the years to better suit other technologies. Equation 2.3 is a variation designed to specifically enable better performance curve prediction with multiphase fluid mixes [7].

Because of the prominence of this version of the AICD equation in the industry, it will be the standard to which the work in this project is compared, and the standard-bearer for the performance of fitted fixed structure equations to predict performance curves.

2.2.2 Optimizing the AICD Equation

Along with having a viable AICD equation, which is capable of modeling the performance of an AICD, one also needs to tune the equation to optimally represent the performance curve of an AICD. For a given set of known data, the relevant tune-able variables in the AICD equation, $x, y, \mu_{cal}, \rho_{cal}$, etc., need to be optimized such that the function returns the best estimate of the data points possible. Typically, this is done with multi-variable nonlinear least-squares regression.

Briefly, nonlinear least-squares is a least-squares analysis method designed to approximate data points with a curve. The curve, $\hat{y}\left(\mathbf{x}, \boldsymbol{\beta}\right)$, is an arbitrary curve with properties similar to the behavior being modelled. The system has a selection of function tuning parameters, $\boldsymbol{\beta} = (\beta_0, \beta_1, ..., \beta_n)$, and dependent variables $\mathbf{x} = (x_0, x_1, ..., x_m)$.

How well the curve matches the data set can be determined by observing a sum of the residuals, r_i, squared, also known as the L2 norm squared.

$$F = ||\mathbf{r}||_2^2 = \sum_i r_i^2,$$
$$r_i = y_i - \hat{y}\left(\mathbf{x}_i, \boldsymbol{\beta}\right)$$

When F is at its minimum, an optimal fit has been found. When F is minimized the gradient of F will also be zero with respect to the tuning parameters $\boldsymbol{\beta}$. Therefore, to determine when F is minimized, it will be determined when the gradient of F is zero for each tune-able parameter β_j.

$$\nabla F_j = \frac{\partial F}{\partial \beta_j} = \sum_i r_i \frac{\partial r_i}{\partial \beta_j} = 0 \quad \text{for } \; j = 0, 1, ..., n \tag{2.5}$$

Note that in some cases Equation 2.5 might be written with a factor of two, generated when computing the gradient of r_i^2. However, this can be dropped since the system is set to zero. Equation 2.5 is typically re-written using the Jacobian matrix, $\mathbf{J}$, where $J_{ij} = \frac{\partial r_i}{\partial \beta_j}$.

$$\nabla F_j = \sum_i J_{ij} r_i \quad \text{for } \; j = 0, 1, ..., n$$

Performing a 1st order Taylor approximation on r_i, the nonlinear system is converted into a linearly approximated system to be solved iteratively by successive improvements on an initial guess to $\boldsymbol{\beta}$.

$$\nabla F_j \approx \sum_i J_{ij} \left(r_i - \sum_k J_{ik} \Delta\beta_k \right) \quad \text{for } \; j = 0, 1, ..., n$$

Where $\Delta\beta$, defined as the error in tuning parameters $\beta_k^{n+1} = \beta_k^n + \Delta\beta_k$, is the adjustment to β for the next iteration $n + 1$.

By collapsing this system into matrix form and restructuring the equation, one arrives at the system that needs to be solved.

$$\left(\mathbf{J}^T \mathbf{J}\right) \Delta\boldsymbol{\beta} = \mathbf{J}^T \mathbf{r}(\boldsymbol{x}, \boldsymbol{\beta}) \tag{2.6}$$

The system is solved by starting with a guess for $\boldsymbol{\beta}$, calculating the respective $\boldsymbol{r}$ vector and $\boldsymbol{J}$ matrix, and solving for $\Delta\boldsymbol{\beta}$. $\boldsymbol{\beta}$ is simply adjusted by $\Delta\boldsymbol{\beta}$ for the next iteration. This process is repeated until $\Delta\boldsymbol{\beta}$ is close enough to a zero vector that the desired accuracy is reached.

To use this system to fit a selected AICD equation to an AICD's performance curve, the data set and AICD equation simply need to be inserted into Equation 2.6. As an example, if Equation 2.4 is the chosen AICD equation, Equation 2.4 will take the place of the arbitrary nonlinear curve, $\hat{y}$, and the data points will be structured in the form $(\boldsymbol{x}_i, y_i) = (\alpha_{o,i}, \alpha_{w,i}, \alpha_{g,i}, \mu_{o,i}, \mu_{w,i}, \mu_{g,i}, \rho_{o,i}, \rho_{w,i}, \rho_{g,i}, Q_i, \Delta P_i)$. This leaves just the tuning parameters to be the remaining variables: $\boldsymbol{\beta} = (x, y, \mu_{cal}, \rho_{cal}, A_{aicd})$. After structuring the system, and iteratively solving the equations, the optimized parameters $\boldsymbol{\beta}$ will be the result and will allow the equation to be queried for a prediction of the performance.

2.2.3 Alternatives to the AICD Equation

One of the prominent issues with the AICD equation is that it was designed with single phase flow behavior at its core. While the AICD equation reflects the shape of single-phase flow quite well, the equation does not provide flexibility for multi-phase flows that are not perfectly mixed [4]. One particular variation, shown in Equation 2.7, attempts to overcome this structural weakness by restructuring the AICD equation to map an AICDs response to both viscosity and density with a single variable: x [5]. This new equation was also found to be more accurate describing single phase flow performance than the standard AICD equation when tested on the 3B Fluidic Diode AICD single phase data [4].

$$\begin{aligned} \Delta P &= \left(\frac{\rho_{mix}^{x-1}}{\mu_{mix}^{x-2}}\right) B_{aicd} Q^x \\ \mu_{mix} &= \alpha_o \mu_o + \alpha_w \mu_w + \alpha_g \mu_g \\ \rho mix &= \alpha_o \rho_o + \alpha_w \rho_w + \alpha_g \rho_g \end{aligned} \tag{2.7}$$

Even with this improvement, however, the system is still incapable of adapting to AICDs with geometries that physically change in a binary fashion (i.e. open or closed), or have a different relationship between density and viscosity response. A single general equation-based solution is unlikely to stand the test of time to model new and emerging technologies without further changing the equation.

In 2021, *Yavari et al.* created several machine learning models that outperform the standard equation-fitting-based solution [19]. Their work involved comparing a modified AICD equation, Equation 2.7, optimized with a genetic algorithm instead of the typical nonlinear least-squares solver, against a linear regression model, quadratic regression model, and four types of machine learning models. The four machine learning models tested include

an Artificial Neural Network (ANN), a support vector machine (SVM), an adaptive network fuzzy inference system (ANFIS), and a genetic algorithm augmented ANFIS model. Three of the Four machine learning architectures tested, the ANN, SVM, and ANFIS, were similar in performance and easily outperformed the fitted AICD equation. Additionally, one of the primary benefits of a machine learning approach is that the models are not designed to follow the specific behaviors of one single model: the same model can be trained on data from another AICD, and theoretically should not be disadvantaged.

The primary limitation of *Yavari et al.*'s study [19], as is frequently the case for machine learning projects, is a small training data set size. The data set was composed of 90 data points, 18 of which were isolated for testing purposes, leaving only 72 data points to test the models with. The impact of this data limitation is notable in the design of the models: the ANN was created with only eight hidden nodes over one hidden layer, dramatically limiting the maximum complexity of the model. It's also notable that while the study takes a broad look at machine learning as a whole by examining four different architectures, it does not examine alternative techniques to account for limited training data such as data augmentation or cross-validation.

3 Introduction to Machine Learning

3.1 Deep Learning and Artificial Neural Networks

Machine learning is a field in computer science focusing on computer processes that are capable of *learning* to perform a task better with training. Deep learning is a subsection of machine learning utilizing artificial neural networks (ANNs) to learn to perform a task. This section examines the design and function of ANNs, and what decisions need to be made when creating an ANN to perform a task. The discussion is limited to *supervised* deep learning models, where the training data set is 'labeled', meaning that it has the correct answer attached.

3.1.1 Artificial Neural Network Design and Function

Conceptualized to function similarly to a biological brain which is composed of many simple neurons forming a larger more complex system, ANNs are built up of layers of simpler nodes, often referred to as neurons, feeding into each other in a larger network. On its own, each node is highly simplistic, capable of performing only basic operations on its input signals. The nodes between the input and output nodes are known as hidden nodes since they aren't directly interacted with by a user, and the layers they form are called hidden layers. A larger system capable of performing advanced computations and analysis is then created by layering the neurons into a more complex network. A basic three-layer ANN, composed of two input nodes in the input layer, three hidden nodes in the hidden layer, and two output nodes in the output layer, is illustrated in Figure 3.1.

An ANN functions by first feeding a set of inputs into the input layer, this could be anything from pixel values of an image to discrete values characterizing a fluid. The input values are then multiplied by a weight and fed into the next layer, visualized using arrows in Figure 3.1. The weight controls how strong the dependence is between two neurons in two successive layers. For example, a weight of zero would mean there is no information passing between the two nodes while a weight of 1 implies a unitary dependence on the input. All the inputs, multiplied by their respective weights, leading to a node are summed up and assigned to that node with a bias, a node-specific constant added to the node value. An activation function, which will be explained in the next paragraphs, is then applied to the value. The node value is subsequently passed to the nodes in the next layer and the operation is repeated until the output layer is reached. Depending on the number

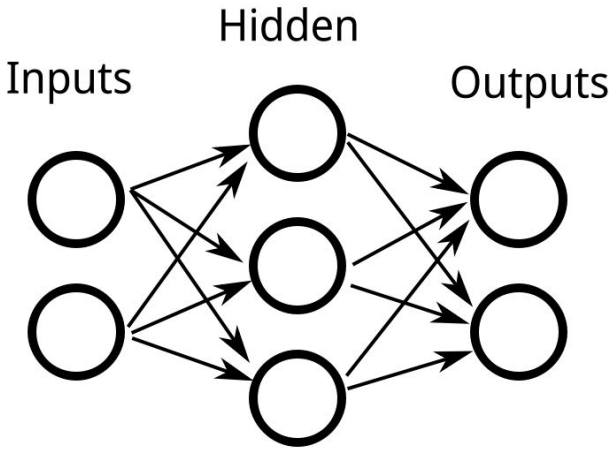

Figure 3.1: Diagram of a basic neural network with an input layer, one hidden layer, and an output layer.

of output nodes a network has, the network output is the value or values assigned to the nodes in the output layer of the network.

Each node can be mathematically represented as

$$a_j = f\left(\sum_i (x_i * w_i) + b_j\right),$$

where a_j is the node output value, f is an activation function, x is an input node, w is a weight, b is a bias, i is an iterator over the input node layer, and j is the iterator of the current node layer. Applying this equation to each node in a layer, Equation 3.1 shows how a layer can be written in matrix form.

$$\vec{a} = f\left(\mathbf{W} \cdot \vec{x} + \vec{b}\right) \tag{3.1}$$

The activation function, f, is central to the success of any ANN. For the ANN to work as a deep network, that is a network with one or more hidden layers, a nonlinear activation function is required. If the activation function is linear or omitted the entire network collapses down to a single linear layer, which is incapable of learning behavior more complex than linear relationships between input values. How this happens can be shown by examining what happens if two layers, represented by Equation 3.1, are placed in line with linear activation functions f_1 and f_2, which can be written in their matrix forms $\mathbf{F}_1$ and

$\mathbf{F}_2$. This is done in Equation 3.2 by directly feeding the output of the first layer, $\vec{a_1}$, into the input vector of the second layer, $\vec{x_2}$.

$$\begin{aligned}
\vec{a} &= f_2\left(\mathbf{W}_2 \cdot f_1\left(\mathbf{W}_1 \cdot \vec{x_1} + \vec{b}_1\right) + \vec{b}_2\right) \\
&= \mathbf{F}_2 \cdot \left(\mathbf{W}_2 \cdot \mathbf{F}_1 \cdot \left(\mathbf{W}_1 \cdot \vec{x_1} + \vec{b}_1\right) + \vec{b}_2\right) \\
&= \mathbf{F}_2 \cdot \left(\mathbf{W}_2 \cdot \left(\mathbf{W}'_1 \cdot \vec{x_1} + \vec{b}'_1\right) + \vec{b}_2\right) \\
&= \mathbf{F}_2 \cdot \left(\mathbf{W}'_{12} \cdot \vec{x_1} + \vec{b}'_{12}\right) \\
&= \mathbf{W}' \cdot \vec{x_1} + \vec{b}'
\end{aligned} \tag{3.2}$$

The linear activation functions multiply through to the weight matrices and bias vectors, reducing the entire system down to a single linear operation. Alternatively, if the activation functions f_1 and f_2 are nonlinear, they cannot be directly swapped to a matrix and the system will not to collapse to a single linear layer. This directly motivates the well-known alternating linear layer and nonlinear activation function structure to build deep ANNs.

There are many actively used nonlinear activation functions to choose between, with entire studies devoted to identifying new and better functions in specific applications [15]. When selecting an activation functions, it is recommended to consider the behavior of the function in conjunction with the intended task for the network. Two classic activation functions, sigmoid and tanh, both have structures that tend to sort values into two groups. This grouping can be seen in their plots in Figure 3.2, where unless the input value, x, is reasonably close to 0, the output value will either be close to one of the boundary values (0 or 1 for sigmoid and -1 or 1 for tanh). This can be a valuable property in binary classification models, where the model is tasked with sorting a set of inputs into two classes. However these nonlinear functions are less beneficial for models which are searching for continuous numerical outputs since many values that are too high or low will be overwhelmingly grouped around the values of 1, 0, or -1, depending on the activation function.

A common nonlinearity used today is the Rectified Linear Unit (ReLU) function, visualized in Figure 3.3 and defined in Equation 3.3. ReLU has been popularized for its consistent training performance and strong gradient calculations over a large domain [1].

$$ReLU(x) = f(x) = \max(0, x) \tag{3.3}$$

One of the most notable differences between ReLU and both sigmoid and tanh is the constant non-zero gradient for all values greater than zero. Why this is important will be broached in the next subsection. However, it is worth noting that non-zero gradients for

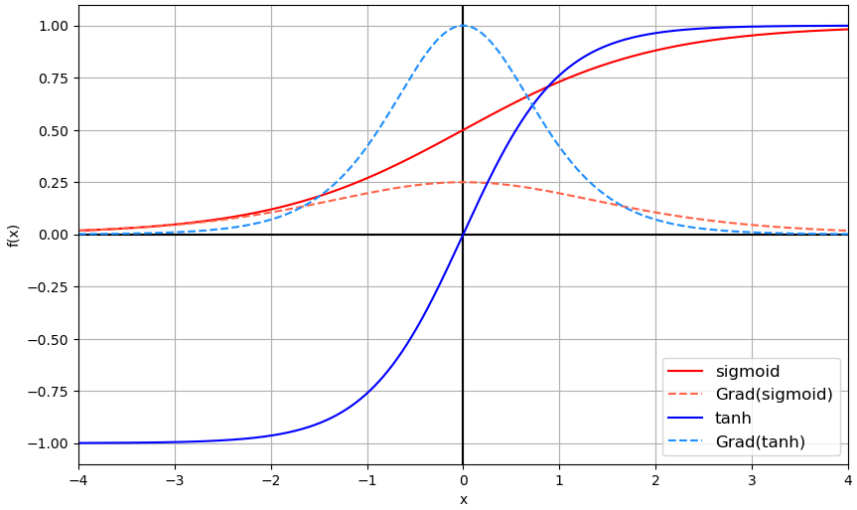

Figure 3.2: Comparison of the sigmoid and hyperbolic tangent activation functions. For input values not near 0, sigmoid tends to group output values to either 0 or 1, while the tanh function tends to group values to -1 or 1.

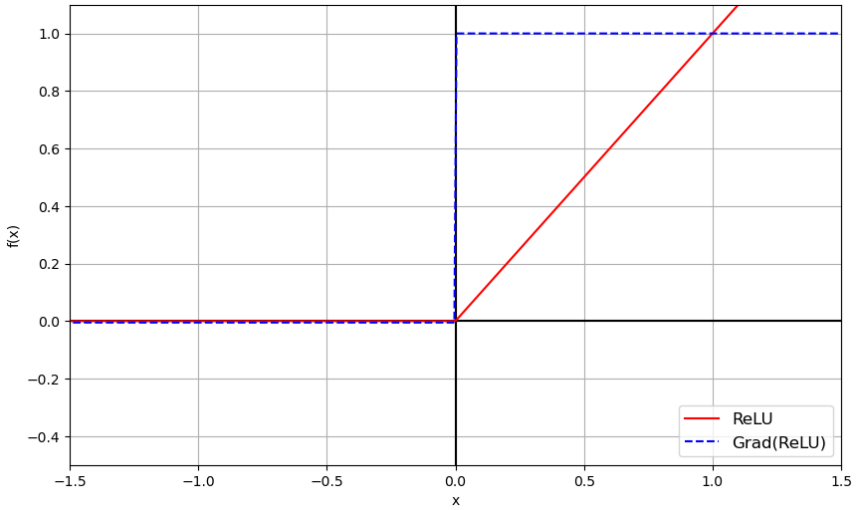

Figure 3.3: Visualization of the ReLU activation function with its non-zero gradient for all input values greater than 0.

more values of x can allow for more consistent training gain. For applications where non-zero gradients for x-values below zero are a concern, there are variations such as the leaky ReLU function which applies a typically small, positive, slope to x-values below zero.

3.1.2 Training an Artificial Neural Network

An ANN model does not know how to do something well until it is trained. Training involves finding the optimal weights and biases for an ANN that will allow the model to best perform a task. For a system predicting the performance curve of an AICD, the task is to predict the pressure drop across the AICD for given input fluid characteristics and flow rates. Training this model is therefore a matter of reducing the error in the predictions of the network output.

Broadly speaking, supervised training of an ANN is composed of the following steps:

1. Retrieve data from the training data set and have the model make a prediction from it.
2. Perform back-propagation on the model using the prediction error.
3. Have the optimizer perform one learning step on the model according to how well it performed.
4. Repeat until training is complete.

The system starts with the most important, and often most limiting, part of any machine learning project: the training data. At its core, an ANN works to replicate any relationships in the data set it is trained with. The implication of this is that the trained model will never perform better at a task than the data it is trained on. If there is not enough data to perform the task better than the model will be required to do, then a model trained with supervised learning will never be capable of performing the task well enough.

Yavari et al., as discussed in Section 2.2.3, used supervised machine learning to predict AICD performance curves[19]. It is interesting that the models outperformed the fitted AICD equation despite being trained with extremely limited data sets composed of only 90 data points, of which 20% were isolated and untouched for testing use. Admittedly, the data set was simplified to only three input variables (mix density, mix viscosity, and fluid flow rate), limiting how complex a network could be trained and thereby also reducing the training data quantity requirements. The final ANN produced used only eight hidden nodes in one hidden layer to produce its results. Increasing the amount of training data would allow larger networks with more input variables to be trained, and subsequently more complex relationships and physical phenomena to be predicted.

In step two of training an ANN, a loss function, such as a mean squared error, is used to assign a numerical value to how incorrect a model prediction is. When the loss function

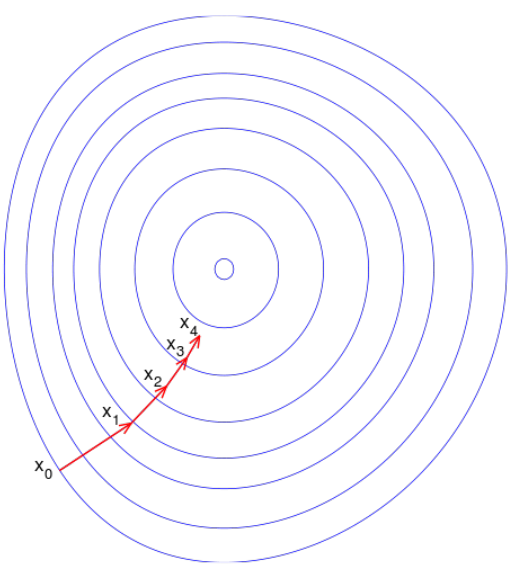

Figure 3.4: Visualization of the gradient descent algorithm on a surface. In each iteration, the algorithm moves opposite the direction of the gradient (i.e., *down*) by a distance determined by magnitude of the gradient. Figure source: [3].

is minimized the error should also be minimized, and the trained parameters are optimal. For step three, the trainer uses an iterative optimization algorithm to adjust the weights in the network to a more optimal state that reduces the errors of the network. Gradient descent, visualized in Figure 3.4, is an intuitive example of such an optimizing algorithm. The Adam algorithm is a more efficient stochastic optimization algorithm that is also commonly seen in training ANNs [9].

All these algorithms, however, depend on the gradient of the loss function with respect to all the weights and biases in the network to determine how to alter the weights. Backpropagation provides the required locally determined gradients for the optimization algorithms by calculating the gradient of the loss with respect to the weights across the network. This allows the optimizer to predict what happens in the neighborhood of the weights currently in use. For many ANNs, this step is a computationally expensive, which is why activation functions are designed with quickly computed derivatives.

Finally, the optimizer needs to know when the network is optimally trained so training can be terminated. Typically, assuming the neural network is complex enough that it can learn the desired rules, there is an 'optimal' point where the current system has learned all that can be extracted from the training data. If the optimizer keeps training the model, it overfits and starts learning individual data points instead of the desired overall behavior and

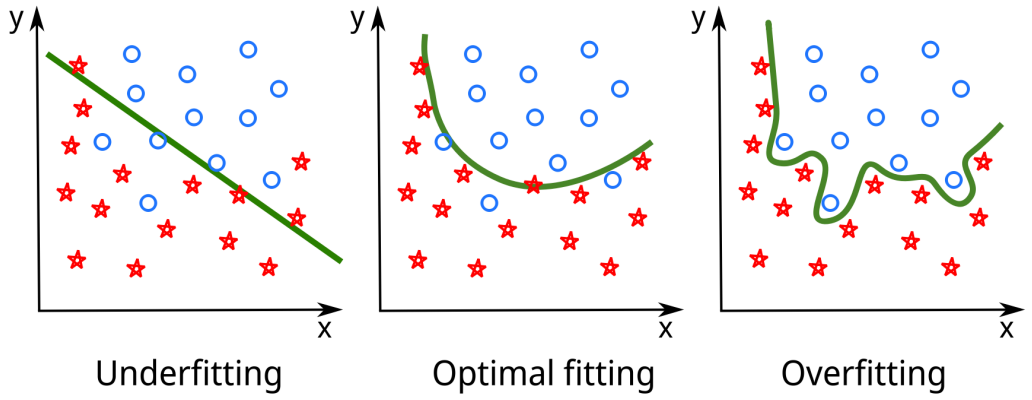

Figure 3.5: Illustration showing underfitting, optimal fitting, and overfitting of a simple classifying model. The left graph is underfitting the problem: the model has only learned a linear distinguishing line. The right graph is overfitting: the model is memorizing individual points. The middle graph is optimally fit: the model has learned the general structure of the data.

thereby reducing the performance of the model. Conversely, before the model reaches the optimal training, it is underfit and still has not learned the trends and rules it is supposed to learn from the data. At the point the model transitions from underfit to overfit, the model is optimally trained and training should stop. Figure 3.5 shows how a model that classifies two shapes on a simple 2-D grid can behave when overfit, underfit and optimally fit.

Tracking the validation loss during training is a method for determining when an optimally trained state has been reached. Validation loss is calculated by using the model to predict the labels of another data set that is not used in training, and then applying the loss function between the predictions and the labels. At the start of training, when the model is underfit, the validation loss will tend to drop with the training loss as the model learns the basic rules that apply to all the data – both validation and training. At a certain point, the validation loss will start increasing, while the training loss continues to decrease. This is where the model is starting to overfit and is learning individual data points in the training data set instead of the rules that would also apply to data that it wasn't trained on. A visualization of what could be seen in the training logs of a model is shown in Figure 3.6.

3.1.3 PyTorch Framework

A machine learning framework allows for easier and quicker use of machine learning. From the ground up, creating a deep learning model is an extremely time-intensive task; there are many moving and interacting parts with complex mathematics and hardware

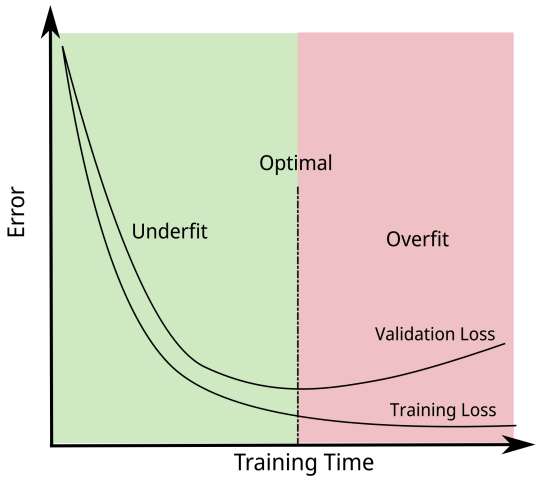

Figure 3.6: Visualization of training and validation loss during the training process. As long as the validation loss decreases, the model is learning to perform better on unseen data and is underfit. Once the validation loss starts increasing, the model is overfitting and starts performing worse. Training should be stopped at the point where validation loss starts increasing.

optimizations needed to create a machine learning model that is both accurate and high-performance. Machine learning frameworks allow a user to cut out the vast majority of the work, and focus only on the design decisions, such as the network structure, optimization algorithm choice, and data augmentation, that matter to their application.

The PyTorch framework [14] is an open-source machine learning framework used to create deep learning models. PyTorch is primarily used with its Python language interface, allowing for quick and more accessible model design and training with optional easy-to-use hardware acceleration. Even though the framework is steered towards the Python programming language, PyTorch also has a C++ API allowing high-performance implementations with a less user-friendly interface. A reasonable workflow could involve creating and training a model in Python, then loading it into a C++ application, such as a reservoir simulator, which has more stringent performance requirements.

3.2 Machine Learning for Performance Curve Prediction

This section will cover the primary issues facing machine learning in the application of predicting AICD performance curves, as well as some methods used to alleviate these problems.

3.2.1 Data Limitations

As is the case for many machine learning projects, data limitations are likely the largest concern for this application. There are generally two aspects to training data that are of concern for a machine learning model: data quality and data quantity. If the data is of insufficient quality, then the model will be trained to output equally bad quality results, and if the data is of insufficient quantity then the model won't be able to be trained on a sufficiently complex ANN to contain the variable relationships in the data.

Regarding AICD performance curves, data quality is perhaps the simpler, albeit not necessarily easier, of the two standards to tackle. *Yavari et al.*'s study of machine learning in AICD performance curve predictions was constrained to three independent variables (mix density, mix viscosity, flow rate) and one dependent variable (pressure drop)[19]. While this may be a benefit in some situations, particularly so for smaller, simpler, models with reduced training data quantities, this could prove disadvantageous for more advanced systems with more input variables and potential generalizability. When test rigs perform measurements on a new valve, they take detailed readings of all fluids and environmental characteristics in the test loop and around the AICD. Getting high-quality data is then a matter of retrieving the entire data set with individual component fluid characteristics and not simplified records where they are joined together.

Data quantity is perhaps the more difficult of the two elements to deal with. Machine learning needs a large quantity of training data that is spread out over all the variable axes that it is desired to work on. For the model to be able to perform over a variety of fluid combinations, training data is needed over the same variety of combinations. However, test rigs are large and expensive to use, so collecting more data points can be prohibitively expensive. Without these larger data sets, a smaller ANN is needed to avoid rapid overfitting, which limits the maximum complexity of the model.

In the search for data for this thesis, it was found that most test data sets are not ideal for ANN training. Often, experimental data is collected for only one- or two-phase flow, omitting the critical three-phase flow that is seen in real-world wells. Additionally, even when three-phase flow is included, only a couple of fluid combinations are tested so the full transient behaviors between the phases are not expressed in the data. Finally, even when all these elements are dealt with, the data sets are frequently very sparse in quantity with less than 100 data points, and without variation in water, oil, and gas density and viscosity.

In machine learning, limitations in training data sets are so pervasive and critical that a multitude of techniques have been developed to enable more effective training to be obtained from constrained training data sets. Data augmentation, perhaps the most common method seen in machine learning, particularly in the field of computer vision, is the process of creating more training data from the data that is already collected by tweaking the data in different ways. Common data augmentation methods include image cropping,

blurring, and mirroring. While extremely effective in image classification models, such direct data augmentation methods are not applicable to the data produced for AICD performance curves that have a small number of input values with very specific correct output values. Any modifications to the training data would reduce the accuracy of the training data, counteracting the goal of producing an accurate predictive model.

In the next two subsections, transfer learning and cross-validation will be discussed, which are techniques that are more likely to be beneficial to this application. Transfer learning and k-fold cross-validation provide benefits to the training process without degrading the quality of the training data.

3.2.2 Transfer Learning

Transfer learning is the process of training a model that is already optimized to perform one task to do another task. The idea is that a model which is already trained to do a similar task, for which there is plenty of training data, can be retrained to the desired task with less training data than it would take to train from scratch. An intuitive example of how this works can be found in the field of computer vision. When trying to create a model that can detect whether an image contains a dog, one can start by taking a model that knows how to detect other objects, such as cats, and retrain the model to detect dogs. Since the model already knows how to look for objects in the image, such as ears or a tail, it should require less additional training to learn how to identify dogs.

In the application of ANNs to AICD performance curves, there are no related models already available. However, data sets that have a similar performance curve can be easily created in the form of the curves produced by a fitted AICD equation 2.3. The curve can be fit to the data, and thousands of randomized data points produced, in less than one second. An ANN could initially learn the basic behavior of the AICD equation, which has a similar response, before the second round of training is performed with the raw experimental data.

The AICD equation is proposed in this case because of its simplicity to implement and test. In the ideal case, a more advanced model would be used to create the initial training data set. Computational fluid dynamics simulations could be a practical source of data, provided they can be produced in a time-effective manner.

In a previous publication [13], I was part of a group testing the efficacy of using simulated data to train a machine learning model to perform a task on experimental data. That project was focused on using convolution neural networks, a sub-type of ANNs including layers with convolutions, to detect defects in liquid crystal thin films. A key draw from the project was that as long as the simulated data phenomena of interest look *similar enough* to the desired target data, viable machine learning models can be produced from easily produced simulated data. Training with experimental data after training with simulated

data should further enhance this approach by correcting errors in the simulation.

3.2.3 K-Fold Cross Validation

In deep learning, data sets are typically split into three parts: training data, validation data, and test data. The test data is removed from the data set at the start of the process and is not touched or observed during the entirety of the training process. It's job is purely to evaluate the model with unseen data after all training is complete. Depending on the application, the test data set is typically anywhere from 10-20% of the data set – if more is used then there is less data remaining to train the model with, and if too little is used then there is less statistical significance to the measurement.

Validation data functions similarly to the test data set in that it is used as a measuring stick of training performance and does not influence weights assigned to the ANN by the optimizer. However, unlike test data, it can be accessed during the training process without biasing the measurement of how well the model performs on unseen data.

In k-fold cross-validation, the training and validation data sets are not held static. In k-fold cross-validation, the entire training data set, without the already isolated test data, is used to train the model. When a validation step is required, the training data set is split into k equally sized segments, referred to as folds. The model is then trained and validated k times, each time one of the folds is selected to be the validation data while the other $k-1$ data folds function as training data. At the end of each validation iteration, the newly trained validation model is discarded and the k validation errors are averaged. Figure 3.7 shows data folding and assignment for 3-fold cross-validation.

Selecting an appropriate k-value for k-fold cross-validation is a trade-off. For higher k-values there is more training data available for the algorithm for each epoch, which reduces bias, the variance of validation loss calculations is increased. Lower k-values, conversely, create larger validation data sets, has less variance in the validation loss but more biased training. A k-value of ten is often proposed as a reasonable compromise, although it is often tweaked on a per-instance basis depending on how consistent the validation loss measurements are [11].

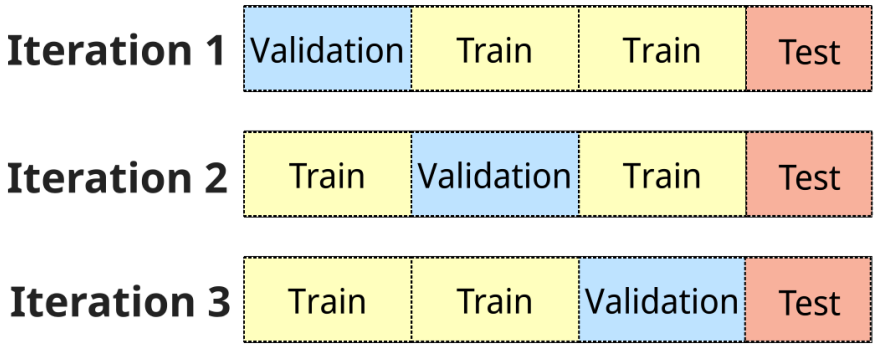

Figure 3.7: Illustration of 3-fold cross-validation. The training data set is segmented into three folds. The model is trained and validated on the omitted data three times. The performance score is then the average of all three iterations.

4 Methodology and Implementation

4.1 Experimental Design

The goal of this thesis is to design a machine learning model for predicting performance curves for AICDs that outperforms the standard fitted AICD equation, and, unlike the AICD equation, is not technology dependent. The model must be able to handle input data more complex than the simplified data set used in previous work employing machine learning to predict AICD performance curves [19]. Allowing more complex input data will make the model more applicable to future technologies that depend on additional variables.

The AICD equation will be used as a representation of the current industry go-to solution for AICD performance curve prediction. The AICD equation will be fit to the same training data, and tested against the same test data, as the machine learning models to ensure all models are limited to the same information.

Two separate machine learning models will be created and trained. The first model will be an artificial neural network (ANN), trained purely on the experimental training data. This functions as a 'baseline' machine learning model with no extra techniques used to accommodate the limited training data. A second model will be created with the same ANN but trained first with data produced by the AICD equation model and then with the same raw experimental data as the first model. This will test the benefit of transfer learning for training models with small data sets using simulated data. The choice to use ANNs, over alternative machine learning architectures, is primarily based on the extensibility of the architecture making it simple to increase or decrease model complexity by adding or removing hidden layers and nodes.

Ideally, a model should also be created to test the benefits of k-fold cross-validation. After substantial work on a k-fold cross-validation implementation, this was deemed too complex for the limited scope of this project since it lacks an easy implementation in the PyTorch framework, thus mandating re-writes of deep learning infrastructure.

Two data sets have been obtained from Tendeka, which will both be used by all the models to verify that any benefits are not specific to a single data set. Additionally, each data set will be run on all models twice: once with the full data set available and once with only 30% of the data available. Since the large data sets for this application can be prohibitively expensive, this will provide a second data point for the effectiveness of these machine

Data Set	%Data	Model
Tendeka-1	100%	AICD Model
		Baseline ML
		Transfer ML
	30%	AICD Model
		Baseline ML
		Transfer ML
Tendeka-2	100%	AICD Model
		Baseline ML
		Transfer ML
	30%	AICD Model
		Baseline ML
		Transfer ML

Table 4.1: Combinations of factors to be tested.

learning models with smaller data sets. Combining transfer learning with two data sizes will also provide insights to the benefits of transfer learning for data sets of different sizes.

Table 4.1 shows all the combinations of factors that will be tested.

Finally, the models will be run in a C++ environment to determine the computational efficiency of the models in a realistic environment where efficiency is a concern. Since the baseline and transfer learning models are using the same ANNs, they will have identical computational complexity. Consequently, only one of the machine learning models needs to be timed in C++ to determine the efficiency of the underlying ANN.

4.2 Data Acquisition and Processing

This section will examine the specific data requirements for this project, where the data is sourced from, and how the data is prepared and processed before use.

4.2.1 Data Requirements

To achieve the best results from this study, there are several desirable quality and quantity standards that will be used.

To obtain the desired data quality, data with multiple independent variables that are factors of the AICDs performance curve is needed. Having more independent variables allows for a more complex neural network to learn more complex behaviors, and ideally provide more accuracy and generalizability. This is particularly important if the model

Independent Variables	Dependent Variable
water fraction (α_w)	
water viscosity (μ_w)	
water density (ρ_w)	
gas fraction (α_g)	
gas viscosity (μ_g)	pressure drop (ΔP)
gas density (ρ_g)	
oil fraction (α_o)	
oil density (ρ_o)	
oil viscosity (μ_o)	
flow rate (Q)	

Table 4.2: Desired variables in training data set labels.

is required to predict performance curves of novel AICDs. As such, data with individual fluid component values is especially desired. Table 4.2 lays out the desired variables in a data set.

As well as having desired variables, the data must also have sufficient data quantity: the model can only learn how to respond to variables if there is sufficient data spanning them. For example, to learn how to respond to gas density, the model needs to train on data covering as many gas densities as possible. The model will in effect only be useful on variables that have sufficient data quantities spanning the variable in the training data set.

Model complexity is also constrained by the quantity of training data obtained. In order to properly train a model using ten independent variables, listed in Table 4.2, more than the 90 data points obtained by *Yavari et al.*'s study [19] is recommended. In reality, 'as large as can be found' will be used, and preferably data sets with at minimum several thousand data points.

4.2.2 Data Sources

Upon reaching out for data, Tendeka kindly provided two separate data sets, created using a multi-phase test rig, of performance curves for their FloSure valve that fulfilled the data requirements. Each data set is composed of 1125 data points, of which fewer than five were trimmed from each set to remove outliers. Each data point consists of all fields desired from Table 4.2. However, as each set is only composed of one set of oil, water, and gas density and viscosity, a model trained with this data cannot be used to test the models' ability to generalize over the respective component fluid properties. The data does, however, contain a large spectrum of three-phase flow mixtures, so the model should be responsive to variations in mixture fractions. A plot of one of the two Tendeka data sets is shown in Figure 4.1.

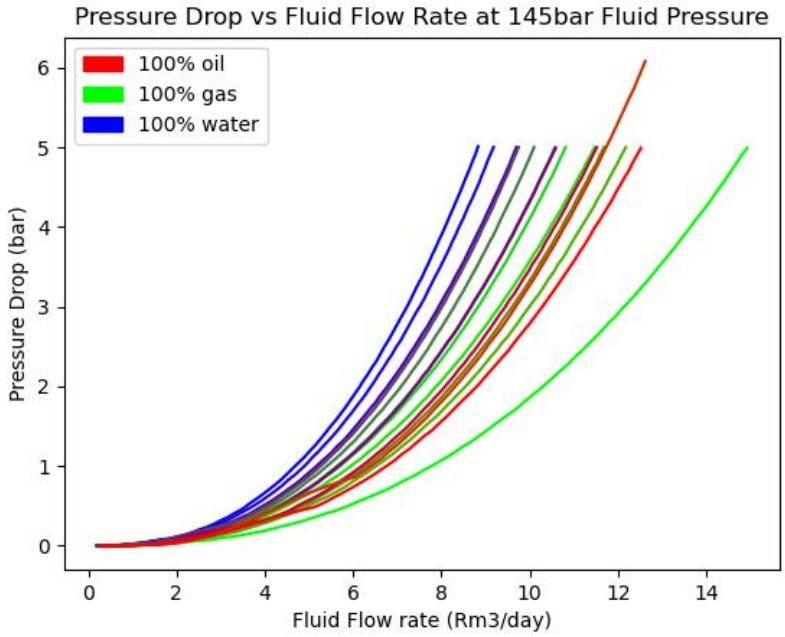

Figure 4.1: Visualization of one of the two Tendeka data sets.

#	ρ_w	μ_w	ρ_g	μ_g	ρ_o	μ_o	α_{water}	α_{gas}	α_{oil}	Q	ΔP_{total}
1	1.021	0.439	0.1118	0.0183	0.755	0.95	1.00	0.00	0.00	0.19	0.00003
2	1.021	0.439	0.1118	0.0183	0.755	0.95	1.00	0.00	0.00	0.37	0.00056
3	1.021	0.439	0.1118	0.0183	0.755	0.95	1.00	0.00	0.00	0.55	0.00267

Table 4.3: Sample of the Tendeka data set.

4.2.3 Data Processing and Partitioning

In preparation for use in the machine learning and AICD equation models, the data needs to be reformatted and partitioned for training and testing. Reformatting is largely a matter of properly separating the two Tendeka data sets, which can not be mixed together since they are retrieved from testing at different fluid pressures, and structuring them in a format that can be imported by the fitting and machine learning models. A snippet of the reformatted data is shown in Table 4.3.

Since there are two Tendeka data sets, only one will be used to create the models and workflow. The second data set will be isolated until a final network design has been determined, so that the results of the two data sets can be used to test performance on a data set that is unseen throughout the entire development process. The data set collected at 150

bar fluid pressure will be isolated, and all development will be done with the 145 bar data set.

After separating the two data sets, they are segmented into training and test data sets. 20% of the total data set is isolated for use as test data and remains untouched and unviewed until the accuracy of a trained model is to be measured. The machine learning models additionally segregates another 15% of the training data for validation, giving a 65-15-20 ratio for training, validation, and testing.

In practice, when fitting the AICD equation to real data, the entirety of the data set is typically used, and the average error between the points and the curve is measured. Since this study will compare the predictive power of the models, however, the models must both predict values for the same data points that have not been seen before. As such, the 20% test data set will also be excluded from the fitting of the AICD equation.

Finally, this entire process is repeated with only 30% of the base data. 30% is selected arbitrarily to bring the number of data points to approximately 300, a number of points that is more typical to be seen in multi-phase AICD measurements from test rigs. The data points that are omitted are chosen randomly before the data is split into training, validation, and testing data sets.

4.3 AICD Equation Implementation

This section examines a use of the AICD equation model for predicting AICD performance curves.

4.3.1 AICD Equation Selection and Optimization Strategy

Considering that data sets from Tendeka will be used in the machine learning models, the same version of the AICD equation that is used by Tendeka in their FloSure data sheet [17] will also be included in this project. Tendeka uses the standard AICD equation (Equation 2.4). Since this variation of the AICD equation is also the go-to version adopted in some reservoir simulators today, there is no need to add additional AICD equations to compare against.

As the variables ρ_{cal} and μ_{mix}^{y} in Equation 2.4 can be linearly absorbed into the constant A_{aicd}, it is common practice to set these variables to 1 when performing the fitting operation. The AICD equation is then formulated as written in Equation 4.1. This means that there are only three terms that need to be optimized: y, the mix viscosity exponent, A_{aicd}, a scaling parameter, and x, the flow rate exponent. To determine the fitted parameters, a nonlinear least-squares optimizer will be used from the SciPy optimization library [18].

$$\begin{aligned} \Delta P &= \left(\frac{\rho_{mix}^2}{\rho_{cal}}\right)\left(\frac{\mu_{cal}}{\mu_{mix}}\right)^y A_{aicd} \cdot Q^x \\ \mu_{mix} &= \alpha_o\mu_o + \alpha_w\mu_w + \alpha_g\mu_g \\ \rho mix &= \alpha_o\rho_o + \alpha_w\rho_w + \alpha_g\rho_g \end{aligned} \tag{4.1}$$

4.3.2 Code Implementation

The code for fitting the model will be entirely written in Python since this is not the implementation that will be timed to determine the performance of the model. As such, a naive implementation of the equation is adequate. The function is implemented as the *aicd_eq* in Source code 4.1, along with the optimization function that optimizes the parameters.

The Levenberg-Marquardt [10] algorithm will be used as the optimizer for the *curve_fit* method of the *scipy.optimize* library to find the fitted parameters of the AICD equation. The Levenberg-Marquardt algorithm is very popular for solving nonlinear least-squares problems because of a guaranteed rate of local convergence under reasonable assumptions, and an acceptable worst case for a bad initial guess [2]. This makes it a safe, high-performance, option for reliable convergence. The Python optimization code is shown in the *optimize* function in Source Code 4.1, which inputs the training data, in a pandas DataFrame object, and returns the optimized parameters.

4.4 Machine Learning Implementations and Structure

This section covers the two machine learning model implementations. The first is a basic bare-bones deep learning model, while the second is a slightly more advanced system employing transfer learning. Both models will only be trained and run on a CPU since a GPU is not available. Training time should not be prohibitively impacted by this limitation, due to the small model size.

4.4.1 First Network for Baseline Training

The first machine learning neural network will be trained solely on the training data set – i.e., 65% of the complete data set, and will not see the test data set until after training is complete. The model is created using PyTorch Lightning, a framework built on top of PyTorch to further automate the construction and training of PyTorch machine learning models.

There are no hard and fast rules for how many hidden layers a neural network should have, or how many nodes to have in each layer. Generally, node and layer counts are determined through experimentation to see what does or does not work. However, as a rule

```
from scipy.optimize import curve_fit
"""
aicd_eq(ind_vars, mu_exp, a_aicd, flow_exp)
AICD equation implementation.
ind_vars are the independent variables for a calculation
rho_cal and mu_cal are assumed as 1 g/cc and 1 cp respectively.

ind_vars: [q, a_o, rho_o, mu_o, a_g, rho_g, mu_g, a_w, \
  rho_w, mu_w]
mu_exp: viscosity exponent
a_aicd: scaling factor
flow_exp: flow rate exponent

return: pressure drop over the AICD
"""
def aicd_eq(ind_vars, mu_exp, a_aicd, flow_exp):
    rho_mix = ind_vars[1] * ind_vars[2] + ind_vars[4] * \
      ind_vars[5] + ind_vars[7] * ind_vars[8]
    mu_mix = ind_vars[1] * ind_vars[3] + ind_vars[4] * \
      ind_vars[6] + ind_vars[7] * ind_vars[9]
    f = rho_mix**2 * (1/mu_mix)**mu_exp
    return f * a_aicd * ind_vars[0]**flow_exp

"""
optimize(data)
Perform optimization of parameters for AICD equation on data.
data: pandas dataframe containing data to be optimized on

return: optimized [mu_exp, a_aicd, flow_exp]
"""
def optimize(data):
    x_data = data[['Rate', 'Oil Fraction', 'rhoo', 'muo', \
      'Gas Fraction', 'rhog', 'mug', 'Water Fraction',\
      'rhow', 'muw']].T
    y_data = data['DPtotal']
    params, _ = curve_fit(aicd_eq, x_data.to_numpy(), \
      y_data.to_numpy(), method='lm')
    return params
```

Source Code 4.1: Code creating a basic AICD equation, and the optimization function fitting the equation to a selection of data.

of thumb, the number of nodes in a layer can be approximated to be between the number of output nodes and double the number of input nodes [8]. For this ANN, this will be between one and twenty since there are ten input nodes and one output node. Determining the number of layers to use is a matter of estimating the expected complexity required, then tweaking the count until an appropriate balance of complexity and efficiency is found.

This network will be built using four hidden layers, with 16-16-12-8 hidden node counts. The ReLU function is used as the activation function, allowing strong gradients and avoiding grouping of values. Mean squared error (MSE), also known as the squared L2 norm, is used as the training loss function, putting a strong bias on fixing outlying errors to hopefully improve reliability. Validation loss is also calculated using the MSE loss function at the end of each epoch.

The learning rate hyper-parameter is determined using PyTorch Lightning's automatic learning rate method, which automates the typically manual process of selecting a learning-rate at the expense of some training cycles. Considering the small scale of the model and data set, this has a negligible performance overhead. This should also remove a bias of human error in selecting learning rates, making the model more consistent to train on new data sets. The trainer uses an Adam algorithm stochastic optimizer [9], to perform the parameter optimization during training.

The actual code generating the ANN, and selecting the Adam optimizer is shown in Source Code 4.2.

To avoid overfitting, the trainer tracks the validation loss at the end of each epoch, and when the validation loss has not decreased for 10 new epochs, training is halted and the model is selected from the epoch with minimal validation loss. This is implemented by providing the PyTorch trainer an *EarlyStopping* callback, with a patience of ten tracking the validation loss. The trainer configuration with early stopping and automatic learning rate selection is shown in Source Code 4.3.

4.4.2 Second Network for Transfer Learning

Transfer learning will be performed using a model trained using data created from the fitted AICD equation. The generated data will compute new data with varying flow rates, water fractions, oil fractions, and gas fractions – the same variables that vary in the training data. Oil, water, and gas density and viscosity are to be held constant since the training data is also constant across those variables.

Flow rates are randomized between zero and the max flow rate in the data sets so the full range of expected flow rates is equally represented. Oil, water, and gas fractions are also randomly generated to always sum up to one. The randomized values are inputs for the fitted AICD equation to produce simulated pressure drops. The simulated values and input values are then joined so the synthetic data is identical in structure to the real

```python
import torch
from torch import nn
import pytorch_lightning as pl

class Network(pl.LightningModule):
    # learning rate is set by auto_lr on call
    def __init__(self, learning_rate=1e-3):
        super().__init__()
        self.linear_stack = nn.Sequential(
            nn.Linear(10,16),
            nn.ReLU(),
            nn.Linear(16,16),
            nn.ReLU(),
            nn.Linear(16,12),
            nn.ReLU(),
            nn.Linear(12,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), \
                lr=self.learning_rate)
        return optimizer

    def training_step(self, batch, batch_idx):
        x, y = batch
        x_hat = self.linear_stack(x)
        loss = F.mse_loss(x_hat, y)
        self.log("training loss", loss, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        x_hat = self.linear_stack(x)
        loss = F.mse_loss(x_hat, y)
        self.log("Validation loss", loss)
```

Source Code 4.2: A snippet of the code defining the neural network class using the PyTorch and PyTorch Lightning frameworks. The *configure_optimizers* method selects the desired optimizer. *training_step* is called at the end of each pass of each batch and *validation_step* is called at the end of each epoch.

```
import torch
import pytorch_lightning as pl

# Create and configure the trainer
trainer = pl.Trainer(
        max_epochs=1000,
        default_root_dir = model_path,
        logger = logger,
        callbacks = [EarlyStopping(monitor="Validation loss", \
            mode="min", patience=10)],
        auto_lr_find = True
        )

# Actually train the model
trainer.fit(model, training_dl, validate_dl)
```

Source Code 4.3: Code configuring and creating a trainer with early stopping and automatic learning rate detection, which is then used to train a model with a training data loader.

experimental data. 10000 data points are produced in under a second and are then sorted into training and validation sets in an 80-20 ratio.

The ANN is identical in design to the one used in the baseline machine learning model and is created by the same class definition. A snippet of this was provided in Source Code 4.2.

Unlike the baseline model which requires only one trainer, this model uses two separate trainers to accommodate the two training cycles. Aside from having unique loggers and data sets, the trainers are configured identically to the trainer for the baseline model. Transfer learning is performed by simply calling the *fit* method of the trainers on the model in succession with their separate data sets. Figure 4.2 shows a flow diagram of the training process for the transfer learning model.

4.5 Performance Comparison and Timing

This section covers the methodology used to measure the performance of the model. Performance of a model includes the error in prediction, the time required to train and fit the model, and finally how fast the model can produce a prediction.

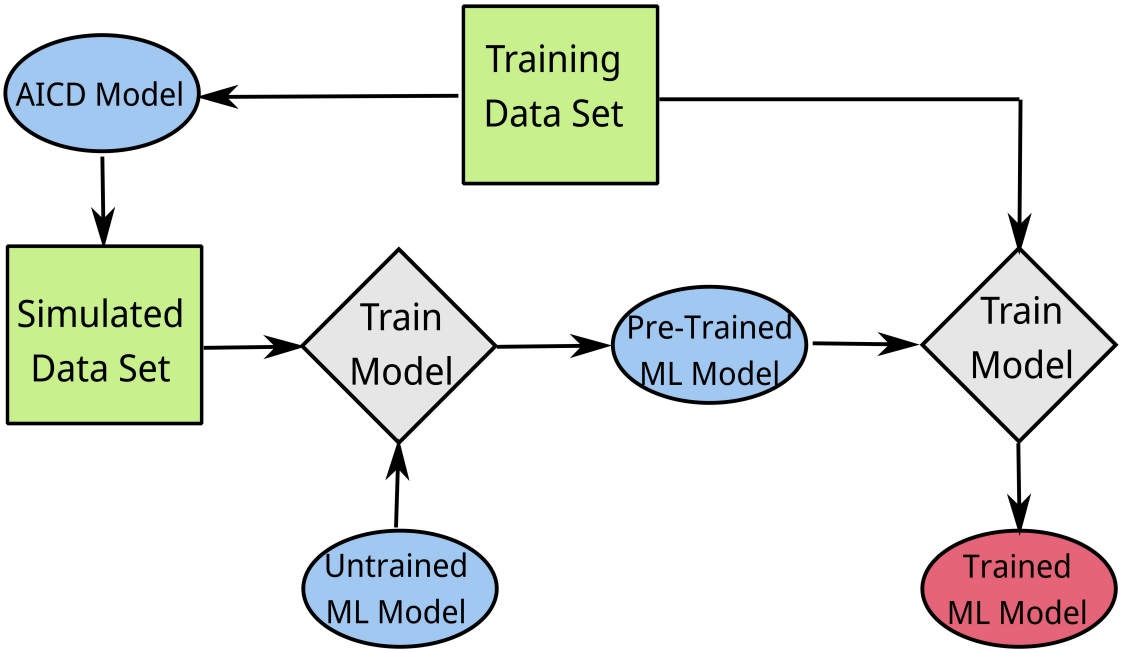

Figure 4.2: Flow diagram showing the training process for the transfer learning machine learning model. The training data is used both to optimize the AICD equation, which produces the simulated data set, and to do the final training on the machine learning model. The untrained model is the same ANN that is used for the baseline machine learning setup.

```
import torch.nn.functional as F
import pytorch_lightning as pl

class Network(pl.LightningModule):
    def test_step(self, batch, batch_idx):
        x, y = batch
        x_hat = self.linear_stack(x)
        loss = F.mse_loss(x_hat, y)
        self.log("Test loss", loss)

model = Network()

# Train the network in here
do_training()

# Test the network on the test data set
pl.Trainer().test(model, test_data)
```

Source Code 4.4: Code defining the testing behavior of the PyTorch neural networks.

4.5.1 Model Prediction Error

The model prediction error is determined by comparing predicted results with experimental values from the test data set. The test data isn't seen by any of the models during training, or fitting for the AICD equation model, so no model will have an advantage in predicting the test data set labels.

The PyTorch Lightning trainer object used to train the models with the training data also has a *test* method, which uses a loss function defined in a model's *test_step* method to calculate the test loss. Test loss is calculated with the same loss function as training and validation loss: i.e., the MSE loss function. The model's *test_step* method, and the call to the trainer *test* method, are shown in Source Code 4.4.

The fitted AICD equation model will have a prediction error calculated using the same loss function. Converting the predictions and test labels to Torch tensors, which are required to interact with the PyTorch framework, the same PyTorch MSE loss function implementation is used for the AICD equation. The function casting the data to a Torch tensor and calculating the error on the test data is shown in Source Code 4.5.

```
import torch
import model_plot

"""
test_equation(test_data, optimized_parameters)

test_data: the test data set as a pandas DataFrame
optimized_parameters: the parameters that optimize the AICD equation

return: MSELoss of the AICD equation
"""
def test_equation(test_data, optimized_parameters):
    predictions = test_data.apply(lambda x: \
            f(model_plot.retrieve_independent(x), \
            *optimized_parameters), axis=1)
    predictions = torch.tensor(predictions)
    targets = torch.tensor(test_data['DPtotal'].values.tolist())
    loss = torch.nn.MSELoss()
    return loss(pred, target).detach()
```

Source Code 4.5: Code defining the testing behaviour of the AICD equation.

```cpp
/*
* AICDfunc(std::vector<double>& params, double mu_exp,
*       double a_aicd, double flow_exp)
*
* params: vector containing {rate, oil_fraction, rhoo, muo, gas
*          fraction, rhog, mug, water_fraction, rhow, muw}
* mu_exp: viscosity exponent
* a_aicd: aicd factor
* flow_exp: flow exponent
*
* return: float, Pressure drop over AICD
*/
float AICDfunc(std::vector<double>& params, double mu_exp,
        double a_aicd, double flow_exp) {
    auto rho_mix = params[1] * params[2] + params[4] * params[5] +
        params[7] * params[8];
    auto mu_mix = params[1] * params[3] + params[4] * params[6] +
        params[7] * params[9];
    auto f = pow(rho_mix,2) * pow(1/mu_mix, mu_exp);
    return f;
}
```

Source Code 4.6: C++ implementation of the AICD equation with basic optimization.

4.5.2 C++ Implementation of the Models

Despite many Python libraries and algorithms being implemented in C, FORTRAN, or another high-performance language under the hood, Python remains a slower language overall for high-intensity computing tasks. As such, most simulators are written in a language other than Python, C++ being a common alternative. To properly determine the performance of these models in real situations, all the models will need to be converted to C++ versions to analyze their performance in a more typical working environment.

The AICD equation is easy to convert by simply rewriting the function in C++. Absolute peak optimization for this is not highly relevant, since function implementations in simulators is proprietary Therefore, only some basic optimization are performed. For example, independent variables are members of a vector passed by reference instead of being passed individually, and the computationally costly terms, ρ_{cal} and μ_{cal}, are combined into A_aicd to reduce the required calculations. The C++ implementation is shown in Source Code 4.6.

The PyTorch machine learning models are more involved processes to convert. While PyTorch does have a beta version of a C++ API, PyTorch lightning does not, meaning a

```
import os.path as osp
import utils

import torch

# Trace and export torchscript model
data_path = osp.join(DATA_DIR, "TrainingData.csv")
example_data = utils.getOneValue(data_path)
traced_model = torch.jit.trace(model, example_data)
traced_model.save(osp.join(model_path, f"{RUN_NAME}_script.pt"))
```

Source Code 4.7: Python code tracing a trained model to Torch Script, and saving the traced model to a .pt file.

C++ rewrite will also involve rewriting much of the control algorithms, most notably early stopping and automatic learning rate selection. On top of the neural network algorithms that need to be rewritten, the infrastructure involving data loaders and loss logging woill need to be rewritten.

PyTorch does, however, have a method for converting trained models from Python to C++ through Torch Script, a representation of PyTorch models that can be read and compiled by both the Python and C++ PyTorch APIs. The method of exporting a model to Torch Script used in this code involves tracing an operation of the code using arbitrary input data, then saving the trace into a Torch Script file which can be read into C++. Source Code 4.7 shows how some source data is selected, run through the model to be traced, and then saved to a file in Torch Script.

The Torch Script model can then be loaded by PyTorch C++ API's *load* function. Using the model is easy as is is callable as a regular function, requiring only the input values as an argument formatted as a vector of PyTorch IValues. The loading and use of a model is shown concisely in Source Code 4.8.

While the performance of these models may be constrained by the fact that only the CPU is used, GPU acceleration is not necessarily beneficial to this application. Neural Networks are typically well suited to GPU hardware acceleration [16]. To what extent GPUs benefit machine learning models is not straight forward as the benefits are highly dependent on a plethora of factors, including memory size, memory access frequency, how parallelizable the calculations are, and the size of the network. For a small model with small data input sizes, the latency of loading data into and out of the GPU might take longer than simply solving it on the CPU.

```cpp
#include <string>
#include <torch/script.h>

/*
* loadModel(string path_to_model)
* Load a torch script model into the PyTorch C++ environment
*
* path_to_model: string path to the model.pt file exported from Python
*
* return: torch::jit::script::Module containing the model
*/
torch::jit::script::Module loadModel(string path_to_model) {
    torch::jit::script::Module mod;
    mod = torch::jit::load(path_to_model);
    return mod
}

// Load the Torch Script model into C++
model = loadModel("model.pt");

// The model can then be used by calling the model with input values
std::vector<torch::jit::IValue> inputs;

auto vals = torch::tensor({4.63,0.25,0.755,0.95,0.0,0.1118,
        0.0183,0.75,1.021,0.439});
inputs.push_back(vals);

auto result = model.forward(inputs);
```

Source Code 4.8: C++ implementation of the AICD equation with basic optimization.

4.5.3 Timing Script

The C++ models are timed with a simple timing script using the *high_resolution_clock* from the C++ chrono library. A model is run once before timing begins in order to load the relevant program code into the CPU cache and thereby remove any ambiguity from some models loading faster into the cache than others. Since simulators are proprietary, and do not advertise how and when they call AICD performance curve prediction functions, it is reasonable remove model loading times from the efficiency metric. Additionally, the models are all called with the same input arguments, arbitrarily selected from the source data set. The timing of each model is then taken as an average of 100,000 calls.

The source code for measuring the time of a machine learning model is shown in Source Code 4.9. The AICD equation model timing is performed identically aside from the input variables being in the structure of a C++ vector of doubles instead of a C++ vector of PyTorch tensors, as is required by the machine learning model.

```cpp
/*
* double time_ml(int count, string path_to_model)
* Time how long the ml implementation takes to run on average over
* count iterations
*
* count: number of runs to average over, 50 for the experiment
* path_to_model: path to the Torch Script prediction Model
*
* return: double containing the average runtime in milliseconds
*/
double time_ml(int count, string path_to_model) {
    double total_time;
    std::vector<torch::jit::IValue> inputs;
    inputs.push_back(torch::tensor({4.63,0.25,0.755,0.95,0.0,
            0.1118,0.0183,0.75,1.021,0.439}));

    total_time = 0;

    //Import the model
    model = loadModel(path_to_model);

    // Initial run of the function to cache program
    auto result = mod.forward(inputs);

    // Start the actual timing of the model
    for (int i =0; i < count; i++){
        auto t1 = std::chrono::high_resolution_clock::now();
        result = model.forward(inputs);
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::milli> ms_double = t2 - t1;
        total_time = total_time + ms_double.count();
}

return total_time / count;
```

Source Code 4.9: C++ code measuring the average run time of a machine learning model. The process is identical for timing the fitted AICD equation, except the AICD equation is called instead of model.forward(), and the input variables are structured as a vector of doubles instead of a C++ vector of PyTorch tensors.

5 Results and Conclusion

5.1 Results

5.1.1 Training Performance

All models were trained using an Intel Core i7-8550U CPU, with a base clock of 1.8 GHz and a boost clock of 4GHz. The training times and respective minimal validation losses of all the models are shown in Table 5.1. The training time for the transfer learning models is the sum of the time needed to train with the simulated data and the experimental data. It is notable that all models were trained in under ten minutes on one core of the CPU, meaning models can be created and trained within a reasonable time with no specialized GPU or CPU requirements.

The results show that a model takes more time to train on the Tendeka-1 data set than on the Tendeka-2 data set, even if they are approximately the same size. This could be because the Tendeka-1 data set is inherently harder for this neural network to learn how to replicate and therefore requires more iterations to train a model on. This is not a consistent phenomenon as the model trained on 30% of the Tendeka-2 data set takes 38% longer than the same model trained on 30% of the Tendeka-1 data set. Considering the 30% Tendeka-2 transfer learning model takes the same amount of time as the same model with 100% data, however, it is possible that it had a poor randomized starting weight which slowed the training of the model.

Overall, the automatic learning rate selection appears to be effective. The validation loss follows a typically desired downward curve, shown in Figure 5.1, indicating an appropriately scaled learning rate. If the learning rate was too high, the training loss would become jumpy and plateau instead of smoothly increasing. This means that manual learning rate control and configuration to train a model on new data is not necessary, removing another barrier for use by automating the process and making the system more accessible to people not experienced with machine learning. This figure also shows a good example of how the early stopping condition works. Minimal validation loss was attained at epoch 15, and when the validation loss didn't reduce again by epoch 25, training was stopped and the model was selected from epoch 15.

The output Torch Script files, which need to be imported into another simulator to make use of the AICD prediction model, are exceptionally small at approximately 20KB. While

Data Set	%Data	Model	Training Time	Validation Loss
Tendeka-1	100%	AICD Model	<1s	–
		Baseline ML	82s	0.031
		Transfer ML	582s	0.005
	30%	AICD Model	<1s	–
		Baseline ML	30s	0.088
		Transfer ML	268s	0.095
Tendeka-2	100%	AICD Model	<1s	–
		Baseline ML	58s	7.04
		Transfer ML	369s	0.80
	30%	AICD Model	<1s	–
		Baseline ML	25s	9.54
		Transfer ML	369s	6.25

Table 5.1: Time required to train the models.

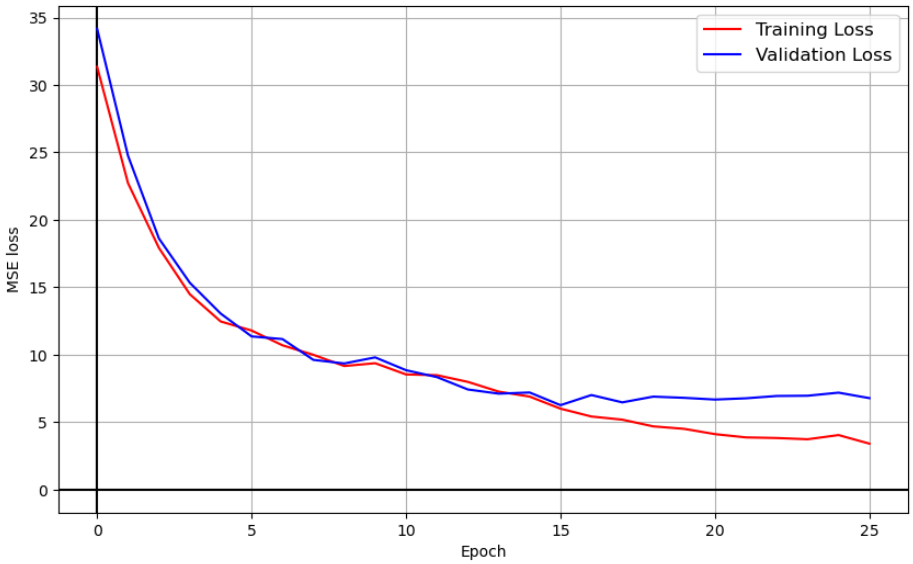

Figure 5.1: Plot of the training and validation loss. This is the training of the transfer learning model with 30% of the Tendeka-2 data set.

Data Set	%Data	Model	Test MSE Loss
Tendeka-1	100%	AICD Model	1.068
		Baseline ML	0.036
		Transfer ML	0.010
	30%	AICD Model	1.664
		Baseline ML	0.105
		Transfer ML	0.081
Tendeka-2	100%	AICD Model	31.31
		Baseline ML	6.98
		Transfer ML	1.13
	30%	AICD Model	41.46
		Baseline ML	7.35
		Transfer ML	6.30

Table 5.2: MSE loss of the models on their respective test data sets.

this is substantially larger than the AICD equation parameters, composed of three doubles requiring only 24 bytes, it is still small for modern computer systems.

The training memory requirements are modest because of the small size of both the training data set and the artificial neural network. The machine learning model, including the python interpreter, and PyTorch framework, peaked at 460 MB during training. The AICD equation fitting script, not requiring the large PyTorch library to operate, has a smaller memory requirement of only 300 MB while fitting. All of these models are well within the capabilities of any modern consumer PC, which typically start at 8 GB of memory.

5.1.2 Model Error Analysis

The model error, measured as mean square error (MSE), on respective test data sets is shown in Table 5.2. Since all models are tested on the same test data set, the errors are directly comparable within the same data set.

Overall, it was found that machine learning, both with and without transfer learning, outperformed the AICD equation in all tested cases. This shows the primary benefit of machine learning over a fixed equation: it is adaptable and can learn to emulate phenomena without requiring an engineer to explicitly formulate the desired response to be optimized. It is possible that a new equation, derived and written specifically for the FloSure AICD, could outperform these machine learning models. However, this is a highly demanding task, requiring large time investments to determine and mathematically represent the specific physics defining the fluid flow through the AICD.

Transfer learning was found to benefit all the tested scenarios. What is surprising about the

Model	Time (μs)
ML model	48
AICD Equation	0.083

Table 5.3: Run time of C++ models averaged over 100,000 runs.

transfer learning accuracy results is that transfer learning provided a larger benefit to the models trained with the entire data sets than it did to the models trained with just 30% of the data. Specifically, when using the entire Tendeka-1 data set there was a 72% reduction in MSE loss, but only a 23% reduction was observed with 30% of the data set. Similarly, the Tendeka-2 data set observed 84% and 14% MSE loss reductions for 100% and 30% of the data sets available for training, respectively. This is opposed to the expectation that transfer learning is more beneficial to training with smaller data sets. One possible reason for this behavior is that pre-training was performed with simulated data that was created from the same reduced data set, so the simulated data was also of reduced quality when transfer learning was performed. By performing the experiments again with simulated data that is not dependent on the training data, such as a fluid dynamics simulation, this hypothesis could be tested.

Figure 5.2 shows the predictions produced by the baseline machine learning model and the optimized AICD equation on seven fluid combinations from the Tendeka-2 test data set. The actual experimental values from the test data are plotted alongside the predictions for comparison. Visually, it is apparent that the machine learning model has matched the shape of the performance curve better than the fitted AICD equation. The baseline machine learning model tends to follow the test data along the entire test space, while the AICD equation quickly diverges away. The AICD equation appears to be incapable of changing the slope of the curves it predicts to the same degree as the machine learning models, and therefore can't produce as accurate predictions.

5.1.3 Model Runtimes

Table 5.3 provides the average time required by the C++ implementations of the models performing predictions 100,000 times.

The machine learning model with a 10-16-16-12-8-1 node structure using the ReLU activation function performed a forward pass in 48 μs – a respectably fast computational time. Compared to the regular AICD equation model, which averaged 0.083 μs, this is a 500–600 times reduction in computational efficiency.

If the simulator requires only around a hundred thousand function calls, about 5 seconds for the machine learning model or 0.01 seconds for the AICD equation model, then this could situationally be a beneficial trade-off. Additional prediction accuracy and precision is provided by the machine learning model in exchange for more computing time.

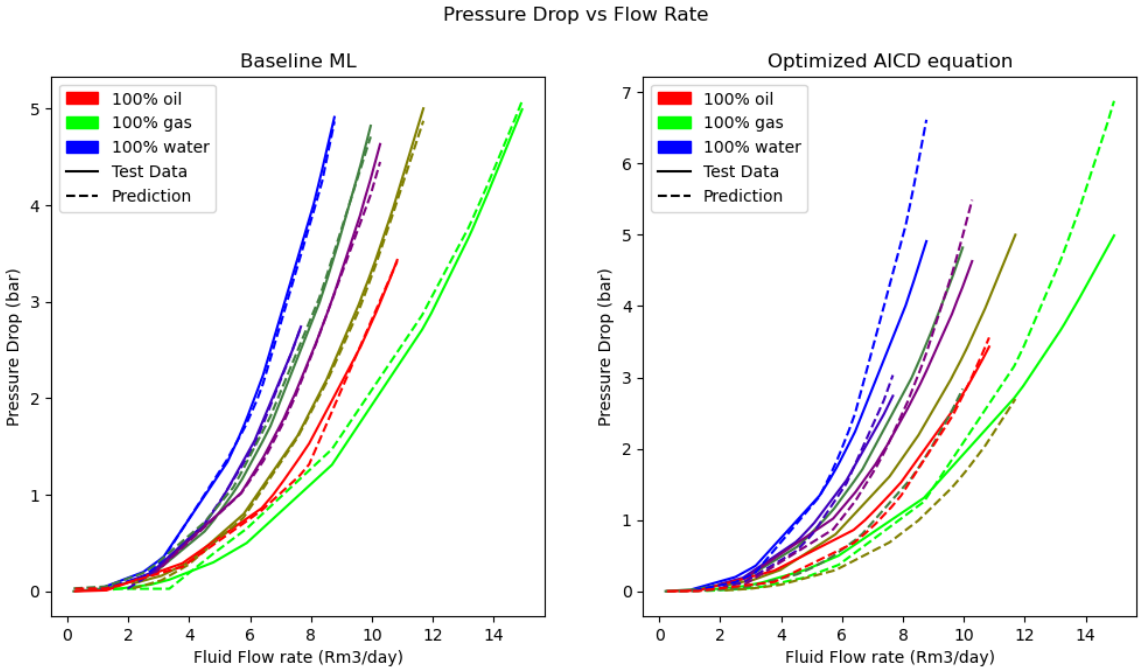

Figure 5.2: Comparison of the baseline machine learning model and the AICD equation. The models are trained/fitted with 100% of the Tendeka-2 training data set. Predictions are performed on the corresponding test data set. Only a subset of the fluid combinations in the test data set are shown to reduce clutter.

In longer simulations, requiring hours of computing time, a handful of extra seconds or minutes is a negligible expense for more reliable results.

5.2 Conclusion

With the goal of creating a more generalized and capable model to quickly predict AICD performance curves, deep learning neural networks were found to be a viable solution. Overall, the primary objectives of the project were completed: improve model accuracy over the AICD equation with a machine learning model, maintain AICD technology generalizability, and minimize human interaction and skill requirements to train and use a machine learning model.

The two trained machine learning models both outperformed the AICD equation in all tested data sets and required no AICD-specific tweaking. Improvements over a fitted AICD equation ranged from one to two orders of magnitude in reduced mean squared error, with the worst case achieving a 78% mean squared error reduction. The AICD performance curves predicted by the machine learning models followed the curve of the experimental data labels for the entire span of the test data set and were capable of adapting the curvature when required while the AICD equation diverged from the experimental results.

Some of the primary issues that are common in machine learning projects, such as long training times and specialized hardware dependency, are not a major concern for this application due to the small system size. All models were trained exclusively on a laptop CPU in under 10 minutes from start to finish. Additionally, all models were trained with a peak memory usage below 500 MB, including the python interpreter and PyTorch library, placing the training and usage of these models well within the capabilities of any modern consumer-grade computer.

As anticipated, substantial improvements were observed when the machine learning models were pre-trained using simulated data, produced by the AICD equation model, for all four data set and size combinations. While the actual benefit varied between the data sets and data quantities, the benefits were universal at the expense of several minutes of training time. While it was hypothesized that the smaller data sets would benefit more from transfer learning than the larger data sets, it was found that the larger data sets had more improvements in accuracy than the smaller data sets. It is hypothesized that this is because the simulated data that was used is dependent on the training data set itself.

Future work could test whether simulated data that does not depend on the training data set could provide further benefits to reduced data set sizes. The unexpected transfer learning results still leave the long-standing issue of training data set size: while the performance is dramatically improved over the AICD equation, data set size still has a dominating impact on the quality of the results.

The impact of simulated data quality used for transfer learning was not tested in this thesis. However, it stands to reason that more 'correct' simulated data that better emulates the behavior of the AICD performance curve could provide even better results than were observed in this project. A source of simulated data other than the AICD equation could prove to be of increased value for AICD technologies that have more characteristics that are not represented by the AICD equation.

Another aspect of transfer learning that wasn't explored in this work is their ability to create general extrapolations over axes with only one, or very few, data points on it. This work was limited to test data sets with variables spanning the same axes as the experimental training data. A model trained with simulation data that includes the expected behavior on additional axes could, theoretically, make extrapolated predictions along these variables following the rules contained in the simulated data.

It is important to note that this study is limited to only two data sets, each of which had only four variables spanned by the data. Initially, the models were intended to be trained using data that had variations across all ten input variables in the AICD equation (oil, water, and gas viscosity, density, and volume fraction as well as fluid flow rate). Additionally, both data sets were from the same technology, so the generalizability of the models to more technologies could not be tested in this project.

The use of an automated search function for the learning rate hyper-parameter at the start of the training process, means that the use of these models requires little to no specialized knowledge about machine learning or computer science. Similarly, automated early stopping conditions based on tracking the validation loss enables the model to automatically pick its top performing version. The use of larger data sets, with more statistically significant validation loss calculations, will make this tactic more reliable and consistent than it currently is. Further tweaking of a model is also not restricted, as the PyTorch Python and C++ interfaces are agnostic to the design of the network between the input and output layers once exported to Torch Script.

Finally, the run times of the machine learning models were found to be reasonably short. Each call to the models takes on average 48 μs using the CPU exclusively. While this is substantially slower than the 0.083 μs observed when using the AICD equation, it still allows over 100,000 predictions in under five seconds. This makes the machine learning models computationally viable for simulations that are not bottle-necked by AICD performance curve predictions.

Bibliography

[1] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), February 2019. arXiv:1803.08375 [cs, stat].

[2] El Houcine Bergou, Youssef Diouane, and Vyacheslav Kungurtsev. Convergence and Complexity Analysis of a Levenberg–Marquardt Algorithm for Inverse Problems. *Journal of Optimization Theory and Applications*, 185(3):927–944, June 2020.

[3] Wikimedia Commons. File:gradient descent.png — wikimedia commons, the free media repository, 2022. [Online; accessed 7-November-2022].

[4] Eltazy Eltaher, Khafiz Muradov, David Davies, and Peter Grassick. Autonomous flow control device modelling and completion optimisation. *Journal of Petroleum Science and Engineering*, 177:995–1009, June 2019.

[5] Eltazy Eltaher, Khafiz Muradov, David Davies, and Ivan Grebenkin. Autonomous Inflow Control Valves - their Modelling and "Added Value". In *All Days*, pages SPE–170780–MS, Amsterdam, The Netherlands, October 2014. SPE.

[6] Martin Halvorsen, Geir Elseth, and Olav Magne Nævdal. Increased oil production at Troll by autonomous inflow control with RCP valves. In *All Days*, pages SPE–159634–MS, San Antonio, Texas, USA, October 2012. SPE.

[7] Martin Halvorsen, Martin Madsen, Mathias Vikøren Mo, Ismail Isma Mohd, and Annabel Green. Enhanced Oil Recovery On Troll Field By Implementing Autonomous Inflow Control Device. In *Day 1 Wed, April 20, 2016*, page D011S006R001, Grieghallen, Bergen, Norway, April 2016. SPE.

[8] Jeffrey Heaton. *Introduction to Neural Networks with Java*. Heaton Research, 01 2008.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[10] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[11] Bruce G. Marcot and Anca M. Hanea. What is an optimal value of k in k-fold cross-validation in discrete Bayesian network analysis? *Computational Statistics*, 36(3):2009–2031, September 2021.

[12] Vidar Mathiesen, Haavard Aakre, Bjørnar Werswick, and Geir Elseth. The Autonomous RCP Valve - New Technology for Inflow Control In Horizontal Wells. In *All Days*, pages SPE–145737–MS, Aberdeen, UK, September 2011. SPE.

[13] Eric N. Minor, Stian D. Howard, Adam A. S. Green, Matthew A. Glaser, Cheol S. Park, and Noel A. Clark. End-to-end machine learning for experimental physics: using simulated data to train a neural network for object detection in video microscopy. *Soft Matter*, 16(7):1751–1759, 2020.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[15] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for Activation Functions, October 2017. arXiv:1710.05941 [cs].

[16] D. Steinkraus, I. Buck, and P.Y. Simard. Using GPUs for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120 Vol. 2, Seoul, South Korea, 2005. IEEE.

[17] Tendeka. FloSure Autonomous ICD. Datasheet, Tendeka, 2022.

[18] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[19] Hossein Yavari, Rasool Khosravanian, David A. Wood, and Bernt Sigve Aadnoy. Application of mathematical and machine learning models to predict differential pressure of autonomous downhole inflow control devices. *Advances in Geo-Energy Research*, 5(4):386–406, December 2021.