Machine Learning for Industrial Processes: Prediction, Monitoring, and Adaptive Control

by

Rui Nian

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Process Control

Department of Chemical and Materials Engineering

University of Alberta

Abstract

Machine learning (ML) has shown great potential to create tremendous value and growth to all sectors around the world, enhancing productivity, health, and longevity of humanity. ML differentiates itself from all previous methods through its adaptive and self-learning capabilities. In recent years, the energy sector experienced significant setbacks due to collapsing commodity prices and increasing pressure from environmental groups. As such, the sector is now actively seeking new innovative strategies to improve their bottom line. One such avenue is to centralize and leverage historical data for process optimization and enhanced business decisions, a concept known as *Industry 4.0*. This thesis aims to explore and demonstrate the capabilities of ML in process modelling, monitoring, and control.

Central to Industry 4.0 is the ability and necessity to create value for stakeholders. As technology continues to progress and data continues to accumulate, it becomes increasingly difficult for engineers to fully understand and optimize modern processes. By leveraging ML, whose performance is highly correlated with the amount of training data, highly multi-variate relationships within modern processes can be identified. Through their discovery, multi-variate optimizations can be leveraged to further enhance process performance and push the bounds operating efficiency. In Chapter 2, a comprehensive process for identifying the multi-variate relationships and optimization step was shown and applied onto an industrial pipeline.

Just as important to process optimizations is the topic of process safety. Currently, the strongest line of defense against process upsets is proactive risk management, where the hazards are eliminated or isolated before they escalate. If this fails,

industrial alarms will warn plant operators of the potential dangers. Unfortunately, many industrial alarm systems are poorly designed, resulting in thousands of flooding alarms during process upsets. Here, ML was first used to construct proactive anomaly prediction tools for passive risk monitoring. To tackle alarm floods, a ML-based alarm management system was introduced to mitigate redundant alarms and prioritize safety critical alarms.

Lastly, process control and optimal control are perhaps the most important subjects in the modern process industry for safety and operation excellence. Traditional optimal control used methods such as model predictive control (MPC) where a model of the process is identified and leveraged to perform multi-variate optimization. Such methods were widely demonstrated on small systems; however, their application in large, multi-variate systems are still limited due to computation constraints. Furthermore, the identification of such processes may not be feasible. In Chapter 5, a ML-based optimal control algorithm, known as reinforcement learning (RL), was leveraged instead to perform optimal control. The two main advantages of RL are its unreliance on a process model and cheap online computation cost, making it a convincing method for processes with un-identifable and/or fast dynamics.

Each application presented was then applied onto an engineering system to validate its effectiveness and feasibility in a physical process. Pipelines, distillation towers, and wastewater treatment plants were selected as the engineering systems due to their importance to society, making them prime targets for optimization. By leveraging ML and RL, the pipelines and wastewater treatment plants undergone significant cost savings while still meeting strict government regulations. Moreover, safety and reliability were greatly enhanced on the distillation tower through a RL fault-tolerant control system. To explore the current progress of RL, this thesis was concluded with a literature review of its current applications in the process industry.

Preface

This thesis is an original work by Rui Nian. Some of the research conducted was an collaborative effort between different organizations. For Chapters 2 and 3, the machine learning project was partially funded by Mitacs through the Mitacs Accelerate Program. Additionally, the project was a joint effort between the University of Alberta, an unnamed large energy company [1], Willowglen Systems Inc., and Alberta Machine Intelligence Institute (AMII). While all the algorithms were developed by myself, the unnamed large energy company and Willowglen's management team provided feedback regarding the complexity and accuracy of the algorithms desired. Additionally, AMII provided the optimization packages and code used in the optimization portions of the product. As of the completion of this thesis, the original tool was cloned onto a second project for similar optimization purposes; thus, the developed optimization toolbox is now applied onto two separate industrial pipelines. In Chapter 4, the waste water treatment plant and alarm management projects were conducted in collaboration with NTwist, a start-up AI company based in Edmonton, Alberta. All algorithms were also developed by myself, but NTwist provided weekly feedback regarding what is feasible and infeasible in industry. Furthermore, the fault-tolerant control system project was completed in collaboration with Imperial Oil. The work here was also completed by myself, but the original research topic was provided by Imperial Oil. Both the Natural Sciences and Engineering Research Council (NSERC) and Faculty of Engineering funded the first year as well as the last several months of my two-year MSc. program.

[1] The company management team preferred to stay anonymous within this thesis, and would like to be referred to as "an unnamed large energy company".

Parts of Chapters 1, 2, and 5 of this thesis was submitted as R. Nian, J. Liu, and B. Huang, *"A Review on Reinforcement Learning: Introduction and Applications in Industrial Process Control", Computer & Chemical Engineering.* I obtained layout suggestions from my supervisors, Dr. Liu and Dr. Huang. The manuscript was written by myself. The alarm management section in Chapter 3 was presented at the INFORMS Data Mining conference as R. Nian, J. Liu, B. Huang, and C. Meenavilli, *"An Adaptive Alarm Prioritization, Reduction, and Optimization Algorithm for Enhanced Process Safety", INFORMS 2019.* The illustrative reinforcement learning example shown in Chapter 4 was submitted as Y. Hong, R. Nian, O. Dogru, K. Velswamy, F. Ibrahim, and B. Huang, *"Reinforcement Learning for Control of a Pilot-scale Centrifugal Pump Process", IFAC 2020.* Additionally, the fault-tolerant control work was published in *SICE 2019* by R. Nian, J. Liu, B. Huang, and T. Mutasa titled *"Fault-tolerant Control System: A Reinforcement Learning Approach".*

Acknowledgements

All the material presented in this thesis certainly could not have been achieved without the countless hours of undivided attention and focused guidance from my professors, colleagues, family and the online machine learning community. First of all, I would like to thank Dr. Fadi Ibrahim and Dr. VInay Prasad, the former being my previous co-op supervisor and the latter being a great mentor during my undergraduate studies, for being the catalyst which ultimately led me down the graduate studies path and introduced me to Dr. Biao Huang and Dr. Jinfeng Liu. Without Dr. Ibrahim and Dr. Prasad, I certainly would never have considered graduate school and would never have had the opportunity to learn as much as I have in the past two years.

Next, I would like to gratefully thank both Dr. Jinfeng Liu and Dr. Biao Huang for the opportunity of pursuing my graduate studies here at the University of Alberta. None of this work could have ever been possible without your continued trust, guidance, and never-ending support. The learning opportunity was also greatly enhanced due to the vast amount of industrial sponsors that both Dr. Huang and Dr. Liu had due to their cutting edge research. Additionally, the presentation experience I have obtained through to the weekly meetings and the IRC meetings are genuinely priceless.

And of course I cannot forget about my research groups for all the continued guidance and patience, accompanying me along my two year journey. They are Ranjith Chiplunkar, Su Liu, Benjamin Nelson, Xunyuan Yin, Soumya Sahoo, Song Bo, An Zhang, and all other lab members who I did not get a chance to work with

directly, but were definitely there to provide me with incredible feedback. Without Ranjith's mathematical expertise, I would never have been able to understand many derivations and other highly theoretical topics. Su helped me tremendously by providing initial recommendations to kick-start my first industrial project. For coding help or theory regarding model predictive control, there was nobody better than Ben, who was always open to helping in any task. The waste water treatment plant project would have been impossible without Xunyuan and An's initial input and help on the model and other performance related topics. And finally, Song was always available when I had very simple questions or simply wanted to chat! Everyone was definitely a critical piece for my overall success.

Another critical part that made my experience as a graduate student much more complete was the chance to work directly with large industrial sponsors through NTwist and Mitacs. Here, I would like to express my thanks to Chowdary Meenavilli, NTwist, and Mitacs. Because of them, I had the opportunity to work directly on real industrial issues and formulating innovative new ways to solve them.

Additionally, I would like to extend a warm appreciation to Professor John Cocchio and Dr. Lisa White, and Dr. Xunyuan Yin for allowing me to be the teaching assistant for ENGG 404 and Ch E 358, respectively. Being a teaching assistant allowed me to see the other side of the table; the hardships and difficulty in teaching courses fairly and effectively.

But not all collaborators helped me in the technical aspect. Some of my most critical collaborators supported me emotionally and provided for my social needs. These individuals are, of course, my mom and dad, Ling Yan Gai, Hong Hui Nian. My grandpa, Dou Shu Gai, my younger brother, Sunny Nian, and my girlfriend, Vivian Tran. Without them, my mental aptitude would not have been resilient enough to allow me to push towards my limits.

And finally, to the unsung heroes of the machine learning community. I would like to express my sincerest gratitude for spending so much time in crafting *excellent* free, and easy to understand content so aspiring machine learning practitioners can learn

the basics without running into a wall of technical jargon. These materials posted online truly serve as a gateway to a new, untapped world. A special thanks goes out to Dr. Andrew Ng and his team for starting Coursera, an economically-smart and value-heavy online course platform. In the future, I will take the community's guidance and pass it forward to the next generation of upcoming machine learning engineers and data scientists.

Contents

List of Figures

List of Tables

Nomenclature

ADAM	Adaptive momentum gradient descent
ADASYN	Adaptive Synthetic
ADP	Approximate dynamic programming
AI	Artificial intelligence
BGD	Batch gradient descent
CEMPC	Centralized economic model predictive control
CMDP	Constrained Markov decision process
CMPC	Centralized model predictive control
CSTR	Continuously stirred tank reactor
DDPG	Deep deterministic policy gradient
DEMPCE	Distributed economic model predictive control (centralized model)
DEMPCS	Distributed economic model predictive control (subspace model)
DPG	Deterministic policy gradient
DP	Dynamic programming
DQN	Deep Q-learning network
EMPC	Economic model predictive control
FOMDP	Fully observable Markov decision process
FTC	Fault tolerant control
GAN	Generative Adversarial Networks
HVAC	Heating, ventilation, air conditioning
LPV	Linear parameter-varying
MAE	Mean absolute error

MAE	Mean absolute error
MC	Monte Carlo
MDP	Markov decision process
MIMO	Multiple-input multiple-output
MRP	Markov reward process
ML	Machine learning
MP	Mathematical programming
MPC	Model predictive control
MSE	Mean squared error
NP	Non-deterministic polynomial time
OU	Ornstein-Uhlenbeck
PDP	Partial dependence plot
PID	Proportional Integral Derivative controller
POMDP	Partially observable Markov decision process
RL	Reinforcement learning
RNN	Recurrent neural network
RTO	Real time optimization
SGD	Stochastic gradient descent
SISO	Single-input single-output
SMDP	Semi Markov decision process
SMOTE	Synthetic Minority Over-sampling Technique
sp	Set-point
ss	Steady state
TD	Temporal difference
TITO	Two-input two-output
UCB	Upper confidence bound
VFD	Variable frequency drive
WWTP	Waste water treatment plant

Chapter 1

Introduction

1.1 Motivation

The non-existent price recovery of the Western Canadian Select crude index since its collapse in 2015 has forced many Canadian energy companies to shift their operating strategies from expansion to optimization [1]. Typically, existing processes in the oil and gas sector have been operating in a similar regime for many years. In doing so, vast amounts of data have been collected for the current operating regime. Through rapid advancements of computer hardware, this data can now be leveraged as a gold mine for modern data hungry machine learning algorithms. Firstly, the data can be used for predictive applications such as forecasting, digital twinning, soft sensing, and even training purposes. The data can also be leveraged to create ML-assisted safety applications similar to driver assistance in the automotive industry. For example, process monitoring and process forecasting ML models can be built to *proactively* manage operational risk by identifying hazards well in advance of actual incidents. Modern optimal control methods (i.e., maximizing profits of a plant or minimizing operating cost) can also benefit greatly through the assistance of ML algorithms. Currently, a common optimal control method in industry is model predictive control (MPC); however, the method assumes the availability of a (often times, accurate) process model. In any industrial scale process, an accurate process model is nearly impossible to identify due to the vast amount of non-linear interac-

tion effects. Even after identification, the model would require re-tuning after several months due to process drifts and other changes. Furthermore, for large processes, the dimension of the states and actions may be too large for online optimization to be feasible. One field of study known as distributed MPC aims to solve this computational hurdle by decomposing the system into smaller sub-systems; however, distributed MPC performance is typically subpar compared to its centralized counterpart due to communication issues [2]. Through RL, such large problems may be computationally feasible as a centralized algorithm by pre-computing the optimal control policies offline. Moreover, process drifts can be naturally handled by RL through its direct adaptive optimal control nature [3]. For traditional optimal control, adaptive characteristics are typically indirect and require re-identification of the system models. In the case of RL, the policy is adapted directly through interactions with the environment. There exist numerous *big data* machine learning success stories in the technology sector such as deep learning for highly effective targeted advertisement. However, applications involving highly complex and nonlinear models in the process industry are still severely limited even though there exist large archives of data. One main reason for the absence of recent big data ML progress in the process industry is the lack of a workforce skilled in both ML *and* process control.

Many technology companies and ML engineers specialized in the big data have attempted to fill the gap; however, process control data is exceedingly different compared to the typical user or transactional data used by technology companies. The data in process control is typically unintuitive and is ofte unreliable or noisy. As such, traditional ML engineers lacking engineering expertise will struggle to build models that translate to true value. Indeed, most of the effort in machine learning projects is not spent on the algorithms, but rather, the data pre-processing step. That is, even simple algorithms such as linear regression can achieve neural network performance if the data is properly de-noised and the proper features are selected. Without fundamental engineering expertise, feature selection and data de-noising

could be almost impossible. For example, an engineer trying to predict the heat released by a reaction would intuitively selects the mass and temperature features due to his/her knowledge of thermodynamics while a ML engineer would start with arbitrary initial features out of the thousands available. There also exist many time delays in chemical processes and feature engineering is difficult without proper fundamentals of process engineering. Comparatively, the data in the technology sector is often very intuitive and easy to understand. For example, building a classification algorithm for facial recognition is easier to understand compared to predicting when a pump will fail. The former only requires an image of the individual or some 3D spatial data corresponding to the individual's facial features. In the latter, there may be thousands of interactions affecting the ultimate outcome of the pump, most of which are impossible to identify through intuition alone. Due to these differences, engineers not specialized in the process sector faced great challenges when attempting to create value in the process industry.

More recently, there has been a surge of ML innovations made by research scientists and AI start-up companies catered towards the process industry. However, most were never commercialized because the mentality between industry and the engineers is vastly different. In industry, the ultimate objective is to create shareholder value through risk-managed products; it may be traditional methods or it can be ML. For the research scientists, the focus is more on the elegance and novelty of the algorithm, regardless of the complexity. For industry, such algorithms are difficult to explain to a non-technical audience, have a high cost of ownership for the customer, and are difficult to understand without a team of subject matter experts (which themselves cost a significant amount of money).

Throughout this thesis, the main theme is to introduce easy, cost effective solutions that explicitly consider the following four customer focused values required for successful commercial products [4]:

- **Functional value:** Describes the overall usefulness of the product compared to other available products. For example, a ML anomaly detection algorithm

may be far superior compared to other methods if sufficient data is present.

- **Monetary value:** The cost savings generated from this product (e.g., amount of money saved through using an optimization algorithm or preventing a loss incident).

- **Social value:** Ability for the product to enhance your brand or product awareness is especially important for sales focused enterprises. For example, after an individual goes to Disneyland, they may tell many people how great it was without any incentive from Disney. In the process industry, operators and/or engineers will recommend great products that helped them in their jobs and/or become more productive without external incentives.

- **Psychological value:** Ability to make the company feel superior compared to the competition. For example, a firm may believe it has better chances at winning contracts if its products contain state-of-the-art ML technology needed for big data applications.

Ultimately, the goal is to create organic growth for the local industry through new, innovative ways. This thesis introduces novel techniques to cater machine learning to the local industry, ranging from commodities transportation to automation.

1.2 Introduction to AI

Artificial intelligence (AI) has set off a change in perspective in the various sectors around the globe, ranging from health care to manufacturing. The previously arcane topic is now spreading wildly across countless academic and industrial minds alike. Quick progressions in computing power and declining prices in data storage combined with AI's self-learning abilities have transcended AI to become the go-to algorithm for many difficult worldwide problems such as natural language processing, predictive analytics, and computer vision. PwC projected AI to contribute well over $15 trillion USD to the global economy by 2030, while elevating GDP of local markets

by 26% [5]. Generally speaking, the field of AI is ever-expanding and contains many goals.

Figure 1.1 shows the six major goals of AI. Out of all the goals, machine learning (ML) is currently the most influential topic in industry. The field of ML can be described as the study that develops algorithms to give machines explicit abilities to learn different tasks without being pre-programmed to do so [6]. ML can be further decomposed into supervised learning, unsupervised learning, semi-supervised learning (a combination of supervised and unsupervised learning), and reinforcement learning.

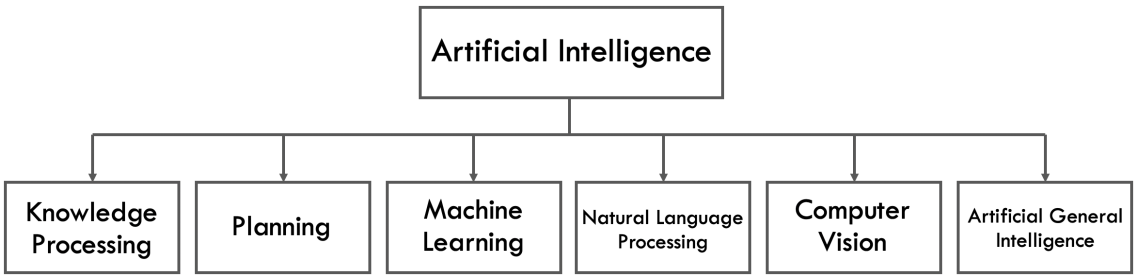


Figure 1.1: The major goals of artificial intelligence.

The sub-fields of ML are shown in Figure 1.2. In supervised learning, the algorithm learns the optimal input-output mapping, called the model, from a training data set pre-labeled by an external supervisor [7]. Be aware that not all labels provided are guaranteed to be correct. In fact, it is not uncommon to have mislabeled data caused by noise in the original data set. For example, imagine trying to transcribe an interview with the audio playback heavily corrupted by noise. In the process industry, the supervisor is typically a sensor measuring the current condition of the process (pressure, temperature, flow rate, etc.) and is often unreliable. In the end, the performance of the supervised learning model is *upper bounded* by the quality of the labels provided by the supervisor. In the ideal case, the model can exactly replicate the right *and wrong* labels of the supervisor. In unsupervised learning, the algorithms are typically used to optimally segregate data based on their similarity or to identify the principal components within large data sets [7], [8]. Objectively, unsupervised learning identifies hidden patterns within data sets through feature

extraction and dimensional reduction. Semi-supervised learning is a hybrid between supervised and unsupervised learning where the models are trained on a small data set of labeled data and refined using features extracted from the unlabeled data set. For example, in the process industry, tasking an engineer to manually label data sets is a costly but required endeavor. In many applications such as fault detection or root cause analysis, a well labeled data set is required to materialize any useful applications. Using semi-supervised learning in these scenarios, the model can learn from the small labeled data set and extract additional helpful insights from the remaining unlabeled data to fine tune performance. In this case, the final algorithm is vastly superior compared to its supervised or unsupervised learning counterpart [9]. Unfortunately, all the above methods exhibit one critical flaw: *the inability to transcend the supervisor in terms of performance.* Although these methods may provide great cost reductions and/or greatly speed up production through automating trivial tasks, the methods fail to expand the current capabilities of modern methods.

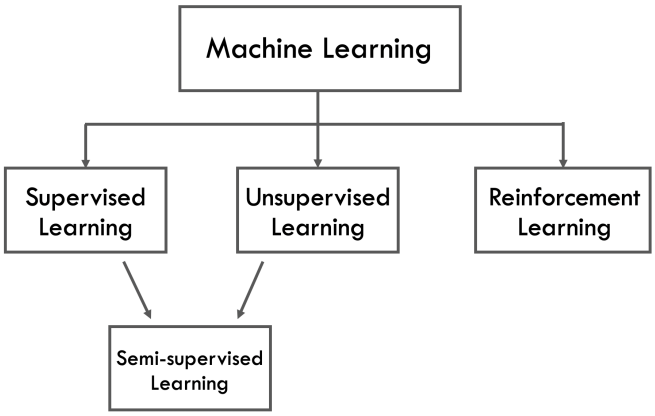


Figure 1.2: The sub-components of machine learning.

Reinforcement learning (RL) aims to overcome this dilemma by providing machines the ability to *surpass all known methods.* More specifically, reinforcement learning *agents* learn the optimal actions to perform in different situations (also called optimal policy) through self-interaction with the environment. After each interaction, the agent is provided feedback via a scalar reward signal; large positive rewards follow good actions while negative rewards follow bad actions. In challenging circumstances, actions affect both the immediate reward signal and the subsequent

rewards there-forth. In an intuitively context, pursuing an University degree may yield negative immediate rewards; however, rewards years down the line may become significantly more positive due to the newly equipped knowledge. These two characteristics—delayed feedback and guided trial-and-error search—differentiate RL from all other types of algorithms and ultimately permit RL to push the existing boundaries of known science [7].

1.3 Thesis Outline and Contributions

The thesis is organized as follows: First, basic concepts of RL and MPC will be introduced. In Chapter 3, applications of ML algorithms in prediction applications will be explored on an industrial pipeline. Following that, ML for process safety applications will be shown in Chapter 4. Safety applications include topics such as anomaly detection, anomaly prediction, and alarm management. Up until Chapter 4, the projects will mostly use traditional supervised, unsupervised, and semi-supervised learning methods because the applications are predictive in nature. Towards the end of Chapter 4 until the end of the thesis, RL methods will be introduced because these applications are more control oriented. Chapter 5 contains various different RL applications in process control. Applications here include the optimal control of a waste water treatment plant, set point tracking control of small scale systems, and fault-tolerant control of an industrial distillation tower. Additionally, RL is also compared to MPC on simple small-scale systems in this chapter. Finally, this thesis is concluded in Chapter 6. A comprehensive project report for the pipeline optimization project introduced throughout this thesis is shown in Appendix A.

The contributions of this thesis are as follows: In Chapter 3, methods for identifying representative process models in an industrial settings are introduced. Additionally, a new adaptive modelling method was formulated here to significantly reduce the cost of ownership of the machine learning models for the industrial partner. The adaptive method also overcomes catastrophic interference and can be retrofitted onto

all model structures. Chapter 4 introduces novel data pre-processing approaches to anomaly detection and prediction in the process industry. Additionally, a new RL-powered alarm management method is introduced for filtering of nuisance alarms, alarm reduction, and alarm prioritization. Chapter 5 provides various comparisons between traditional optimal control methods and RL on many different systems. Furthermore, a new easy-to-implement continuous non-linear RL method is also shown here. The last contribution in Chapter 5 is the extension of RL into a fault-tolerant control where RL is used for both the fault detection algorithm and the fault tolerant controller. Chapter 6 provides a literature review of all the renowned applications of RL as well as RL agents that have potential to materialize value in a process control environment.

Chapter 2

Preliminaries and Tutorials

2.1 Preliminaries to Reinforcement Learning

Reinforcement learning is a goal-directed learning algorithm which continually improves its own performance through interactions with the environment [7]. The main objectives of reinforcement learning are to identify hidden structures within the environment and to find the optimal policy (i.e., optimal state to control action mapping) through guidance from an internal scalar reward (feedback). Two distinct characteristics that deviate reinforcement learning from other methods are its trial & error search to find the optimal policy, and its ability to identify delayed reward signals. Modern reinforcement learning methods combine principles of optimal control and learning methods together to solve for the optimal control trajectory in an environment. In the remaining sections of this chapter, fundamental reinforcement learning concepts will be introduced. Then, tabular based RL methods will be shown. However, due to the curse of dimensionality of high dimensional problems, tabular based approaches struggle in large multi-variate scenarios. To overcome these issues, deep neural networks will be leveraged for function approximation, and deep reinforcement learning will be introduced.

2.1.1 A Historical Overview

Reinforcement learning is a combination of two fields of research: **optimal control** through extremizing an objective function through dynamic programming and **animal psychology** inspiring trial-and-error search. Originally, the **optimal control** problem was proposed for designing controller to maximize or minimize the objective function of a dynamical system over time [10]. By the 1950s, Richard Bellman extended on the works of Hamilton and Jacobi to develop a novel approach to solve the optimal control problem. This approach, known as dynamic programming, optimizes a system's input trajectory by using the functional equation (a function where the unknowns are also functions) generated from the system's state information together with a value function [11]. The functional equation, now called the Bellman equation, is mathematically represented as:

$$V(x) = r(x) + \gamma \sum P(x'|x, u) \cdot V(x') \tag{2.1}$$

where $V(x)$ represents the value function of x. Here, γ denotes the discount factor to incorporate future uncertainty. $r(x)$ is the reward signal obtained as a function of the system's desired performance. $P(x'|x, u)$ is the dynamics function describing the transitional probability of arriving at state, x', given x and u. $V(x')$ is the value function of x'. Intuitively, the value function describes how good or how bad being in particular state is, assuming optimal behaviour thereafter; high values represent good states and low values for bad. True dynamic programming is cursed by dimensionality (i.e., computational cost increases exponentially with the dimensions of the states and actions); thus, approximate dynamic programming (ADP) methods were developed to bypass this hurdle [12]. In reinforcement learning, many ADP methods are leveraged to solve for the optimal policy. The concept of a feedback oriented learning system in RL originated from **animal psychology**. More specifically, the original concept was introduced in the early 20^{th} century, named the *Law of Effect*. The law stated that animals tend to repeat actions resulting in good outcomes, vice

versa for actions with bad outcomes [13]. Initially, the agent explores the environment in which it exists to identify the outcomes corresponding to different actions, then only repeating the actions resulting in good outcomes thereafter. By unifying dynamic programming from optimal control and trial-and-error search from animal psychology, the modern field of RL was developed. For a more comprehensive overview of the history of RL, see [7].

The development of RL is shown in Table 2.1. Reinforcement learning takes its roots from the k-armed bandit problem that has been extensively studied in engineering, psychology, and statistics. This problem disregards state information, and only worries about solving the optimal actions for *one* specific situation [14]–[17]. As a natural extension, Barto, Sutton and Brouwer expanded the idea to multi-situation systems [18] through associative search, also known as *contextual bandits*. The main objective of this algorithm was to find an optimal policy, $\pi^*(x)$, for each situation. However, it only concerns the immediate rewards and not the long term consequences. Reinforcement learning was then developed to find the optimal policy for different situations based on immediate reward and the onward trajectory there-forth.

Table 2.1: From left to right, the evolution of reinforcement learning.

k-armed bandits	**Contextual bandits**	**Reinforcement learning**
Optimal action	Optimal action	Optimal action
One situation	Many situations	Many situations
Immediate consequence	Immediate consequence	Long-term consequence

k-armed Bandit

The k-armed bandit problem provides the fundamentals to understanding modern reinforcement learning. Here, an agent is present and must choose action u from $\mathcal{U}$, where $\mathcal{U}$ has k choices. After each action, a scalar reward from a stationary distribution will be returned to the agent as feedback. Favorable actions yield positive rewards, while unfavorable actions return negative rewards. The objective of the

agent is to ultimately maximize reward over N steps. For each action, there is an expected reward called *value*, given by Equation 2.2.

$$q_*(u) = \mathbb{E}[R_t|U_t = u] \tag{2.2}$$

where u is the action taken at time, t. R_t is a scalar reward returned to the agent after action u was performed at time t. R_t is drawn from a stationary distribution, $R_t \sim N(q_*(u), \sigma^2)$. Finally, $q_*(u)$ is the expected reward of taking action, u.

The real value is unknown, however, an estimation can be computed and is denoted as $Q_t(u)$. Given all $Q_t(u)$ is maintained, at any time, one $Q_t(u)$ will be greater than all others. Picking the action that corresponds to the maximum $Q_t(u)$ is known as *greedy*, and the agent is said to be *exploiting*. If a non-maximum action is picked, the agent is *exploring* [7].

Action selection based on estimating the value of actions are called **Action-value methods** [19]. At time t, the estimate of the value is given by Equation 2.3 [7].

$$Q_t(u) = = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{U_i=u}}{\sum_{i=1}^{t-1} \mathbb{1}_{U_i=u}} \tag{2.3}$$

where $\mathbb{1}$ equals 1 if the condition is true, else 0. R_i is the reward obtained at the i^{th} episode through selecting action, U_i. Intuitively, the numerator is the sum of rewards when action, u, was taken prior to t. Likewise, the denominator is the number of times action, u, was taken prior to t. As $t \to \infty$, $Q_t(u) \to q_*(u)$. Action selection is based on Equation 2.4.

$$U_t = \arg\max_u Q_t(u) \tag{2.4}$$

However, initial successful episodes may cause the agent to be stuck at local minimums. To overcome this, a semi-stochastic action selection method called ϵ-greedy can be introduced to promote exploration. In this method, the agent will perform a random action with ϵ probability (greedy action can be performed). Higher ϵ results

in more exploratory moves. Consequently, all $u \in \mathcal{U}$ will be picked many times and by the law of large numbers, $Q_t(u) \to q_*(u)$ [20]. Figure 2.1 shows the effect of ϵ on the performance of the agent.

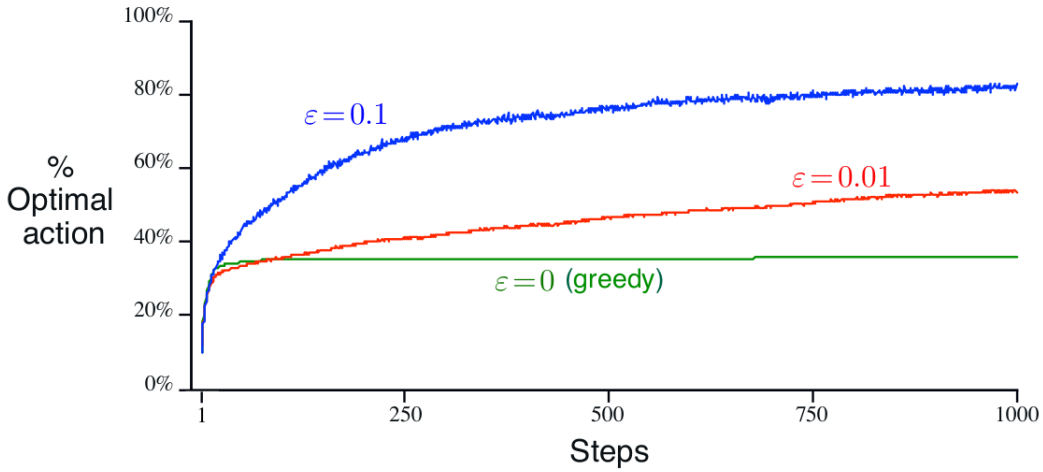


Figure 2.1: Average performance of three agents using different ϵ. The data is averaged over 2000 runs. Figure from *Reinforcement Learning: An Introduction* by Sutton and Barto (2018).

During implementation, ϵ should decay out as $Q_t(u)$ approaches $q_*(u)$ to ensure knowledge of the agent is being adequately exploited. For non-stationary problems where the Q values change, ϵ must be greater than 0 for all t to ensure continued exploration.

Algorithms to solve the k-armed bandit problem are easily applied to situations where the concept of state is inert and only the actions are of concern; a near impossibility in the real world.

Contextual Bandit

A natural extension of the k-armed bandit is associative search. In associative search (sometimes called contextual bandit), different policies are associated with different situations [18]. Equation 2.5 is the extension of Equation 2.2 in the associative search problem.

$$q_*(x, u) = \mathbb{E}[R_t | X_t = x, U_t = u] \tag{2.5}$$

Associative search is known as the method between k-armed bandits and reinforcement learning. In associative search, the objective is to associate optimal policies to different situations, but only maximizing the *immediate* reward. Often times, near term sacrifices are required to initiate the trajectory to a large lump sum reward at the terminal state. For example, heavy capital and time investment is required for University in the short term. However, the long term gain is so great that it outweighs the short term losses, making going to University an optimal policy for many individuals.

In order to find the true optimal policy (i.e., policy that returns the greatest rewards over a long time period), the topic of reinforcement learning is developed. In reinforcement learning, sequential decision making is explored to identify the delayed reward signals from different actions and to ultimately find the optimal policy, π^*.

2.2 Markov Decision Processes

In the presence of uncertainty, the agent's *sequential* decision making is formalized in the Markov decision process (MDP). The general MDP framework is shown in Figure 2.2 and contains two components: the **agent** and the **system**. The **agent** is a continuously learning decision maker and is mathematically represented by the RL algorithm. Objectively, the agent will undergo numerous meaningful interactions with the system to ultimately learn the optimal policy, π^* (i.e., the optimal decisions given different situations). Conversely, the **system** contains all elements the agent cannot arbitrarily control. In process control, the ambient temperature, actuators, and even the wires transporting the control signals are all part of the system because the agent cannot *deterministically* manipulate them.

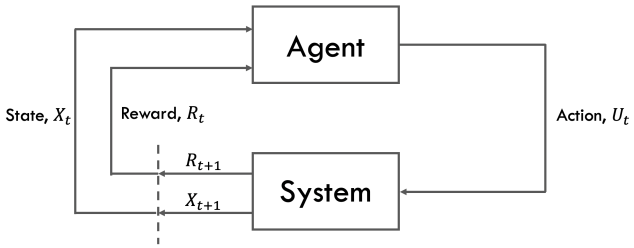


Figure 2.2: The general Markov decision process framework. Original image from [7].

Mathematically, the MDP is a discrete representation of the stochastic optimal problem and a classical formulation of *sequential* decision making where both the immediate and long term consequences are explicitly considered [11], [21]. Many definitions of the MDP exist and are equivalent up to small alterations of the process. One comprehensive definition is that a MDP is a tuple $\mathcal{M} = (\mathcal{X}, \mathcal{U}, p(x', r|x, u), \gamma, R)$ where [22]:

- $x \in \mathcal{X}$: **States** of the system at each time step. Common states in industrial processes include temperatures, valve positions, pressures, flow rates, etc.

- $u \in \mathcal{U}$: Bounded **action** space of the agent, (comprised of at least two elements). In traditional control, this is the **bounded input signals** sent to the actuators.

- $R \in \mathbb{R}$: Expected **reward** signal after performing action u at state x. Reward functions are designed based on a desired performance metric. In control theory, the reward function is known as the **objective function**. Typically, $|R| \leq \mathcal{R}$ for convergence guarantees, where $\mathcal{R}$ is some upper bound of the reward.

- $p(x', r|x, u)$: System **dynamics function**. Formally, it is the probability of transitioning to x' and receiving r, given states $x \in \mathcal{X}$ and performing action $u \in \mathcal{U}$. Mathematically, it is described by the following:

$$p(x', r|x, u) \doteq Pr\{X_t = x', R_t = r|X_{t-1} = x, U_{t-1} = u\} \tag{2.6}$$

where p describes the system **dynamics** and Pr denotes the probability operation [7]. Additionally, p satisfies the following equality:

$$\sum_{x' \in \mathcal{X}} \sum_{r \in \mathcal{R}} p(x', r|x, u) = 1, \forall x \in \mathcal{X}, u \in \mathcal{U} \tag{2.7}$$

Notice here that p is only a function of the *immediate past*, thus assuming that x_{t-1} and u_{t-1} captures the complete history. This is known as the Markov property and its underlining assumptions are critical for successful process control applications using RL. Additionally, note that when the state and actions are formulated as augmented past information: $x_{t-1} = [s_{t-1}, s_{t-2}, ...s_{t-N}], u_{t-1} = [a_{t-1}, a_{t-2}, ..., a_{t-N}]$, where s_{t-N} and a_{t-N} denotes the past states and actions, the system is still Markov because decisions can be made exclusively using x_{t-1} and u_{t-1}.

- γ: **Discount factor** associated with uncertainty of the future, $(0 \leq \gamma \leq 1)$. $\gamma < 1$ is also a requirement for continuous processes to guarantee eventual convergence.

There exists three different MDPs: fully observable MDP (FOMDP), partially observable MDP (POMDP), and semi MDP (SMDP). Table 2.2 shows a general guideline on the different MDPs.

Table 2.2: A comparison of different Markov decision processes.

FO-MDPs	**S-MDPs**	**PO-MDPs**
All states observable	All states observable	Some states observable
Discrete time	Continuous time	Discrete time

2.2.1 Fully Observable Markov Decision Processes

Fully observable Markov decision processes are the simplest and serves as the foundational framework. They are mainly applied to discrete systems with fixed sampling times where transition dynamics are unimportant and all states are ob-

servable (measurable in control literature). Here, the agent starts in some initial states, x_0. At each time t, the agent maps x_t to some u_t corresponding to its policy, π_t. Given x_t and u_t, the system will then transition to some new states x_{t+1} dictated by Equation 2.6 while outputting reward signal R_{t+1} based on the reward function. In regulation and set-point tracking problems, this reward function is typically the squared tracking error between x_t and x_{sp}. By repeating this cycle many times, the agent is able to traverse through some sequence, $x_t, u_t, R_{t+1}, x_{t+1}, u_{t+1}, R_{t+2}, x_{t+3}, ...$ and accumulate [7]:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}... \tag{2.8}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.9}$$

where G_t denotes the cumulative discounted return at time t and γ is the discount factor to capture the future uncertainty. MDPs can represent both finite or infinite systems; the former describes episodic tasks with explicit terminal states while the latter describes tasks that continue forever. Intuitively, most two-player board games such as Checkers, Chess, or Go are finite MDPs where the game is terminated after one player is defeated. Contrarily, an infinite MDP system could be the control system in an industrial process. For infinite MDP systems, $\gamma < 1$ is a necessary condition to keep G_t bounded. Ultimately, the agent is tasked with finding the optimal policy, π^*, that maximize G_t, and subsequently the value function, over N steps. The value function for each state is given as [7]:

$$v_\pi(x) \doteq \mathbb{E}_\pi[G_t | X_t = x] \tag{2.10}$$

$$= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | X_t = x \right] \tag{2.11}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | X_t = x] \tag{2.12}$$

where $v_\pi(x)$ is the value function of x under policy π. Theoretically, the existence and uniqueness of v_π is guaranteed for continuous systems where $\gamma < 1$ or in systems

with guaranteed termination. Compared to Equation 2.2, Equation 2.12 takes the expectation of G_t; therefore, explicitly optimizing the long term returns rather than only the immediate rewards. The action-value formulation of Equation 2.12 is:

$$q_\pi(x,u) \doteq \mathbb{E}_\pi[G_t|X_t = x, U_t = u] \tag{2.13}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|X_t = x, U_t = u\right], \forall x, u \in \mathcal{X}, \mathcal{U} \tag{2.14}$$

FOMDPs work well for discrete systems where all states are observable. However, system states in industrial processes are often unobservable (unmeasurable in control) due to limited hardware or engineering limitations. In such systems, the Markov property no longer holds resulting in sub-optimal decision making of the agent.

2.2.2 Partially Observable Markov Decision Processes

Partially observable Markov decision processes (POMDPs) extend upon the concepts of FOMDPs and represent systems with unobservable states. In RL literature, observability is equivalent to measurability in control; thus, the two terms are used interchangeably here-forth. In FOMDPs, the current state x_t at each time t is fully observable. In the more general setting of POMDPs, the entire state vector describing the agent's current situation is no longer available. Instead, the agent only has access to a set of possible observations $\mathcal{O}$. At each time t, the agent sees observation o_t which corresponds to probability distributions over states. Using o_t, the agent can infer the states it *might* currently be in [22]. Relating to a process control setting, existing sensors typically only measure a subset of the current states; however, by using available measurements, one can infer the remaining unmeasurable states using probabilistic approaches.

Generally, finding π^* in a POMDP setting is significantly harder compared to FOMDPs. Even finding a near-optimal policy is at least NP-hard (non-deterministic polynomial time) [23]. Furthermore, even agents with access to all the system's true

value functions are unable to behave optimally in a POMDP setting because the current states are unknown [22].

Belief states is one method for agents to behave optimally in POMDPs. On a high level, belief states transform the POMDP setting into its FOMDP counterpart through a probabilistic approach. Specifically, belief states, b, are probability distributions over states deduced using previous observations and actions. The probability distributions represent what the agent thinks its current state is. Using these probabilities, the agent can compute scalar value functions of each state-action pair and use these to act optimally. Note here that the agent's behaviour is optimal given the available information, and not optimal with respect to the system. An quantitative example is provided below:

> Suppose an agent exists in a two-input two-output (TITO) POMDP setting with two unobservable states (x_1 and x_2) and two actions (u_1 and u_2) and suppose the problem is only concerned with the immediate consequences (for longer horizons, the agent must also consider the long term rewards, making the example less intuitive). In this system, there are four value functions, one for each state-action pair. Suppose u_1 earns a reward of 2 in x_1 and 0 in x_2. Similarily, u_2 earns a reward of 0 in x_1 and 1 in x_2. Given $b_t = [0.2, 0.8]$ (probabilities of being in x_1 and x_2, respectively), then $Q(b_t, u_1) = 0.2 \cdot 2 + 0.8 \cdot 0 = 0.4$ and $Q(b_t, u_2) = 0.2 \cdot 0 + 0.8 \cdot 1 = 0.8$, resulting in u_2 being the optimal action.

In control theory, observers, such as soft sensors, are used to estimate unmeasurable states. Observers are typically 1^{st} principles, data driven, or probabilistic models. The concept of belief states is very similar to observer design in control theory. Traditionally, Kalman filter is a widely used observer design. Conversely, recurrent neural networks (RNNs) are widely used for belief state estimation in RL. The performance of RNN was compared with Kalman filter in [24], drawing similarities of the two methods' objective, theory, and performance.

System representations using FOMDPs and POMDPs work well in discrete tasks

where transition times are constant and transition dynamics are disregarded; however, both topics are paramount for continuous optimal control.

2.2.3 Semi Markov Decision Processes

Semi-Markov decision processes (SMDP) extend the concepts of MDPs to continuous time and can represent unknown transition times and system dynamics. In SMDPs, the transition dynamics of the system are explicitly captured using reward function [25]:

$$R(x_t, x_{t+1}, u_t) = \int_0^\infty \int_0^t e^{-\beta s} \rho(x_t, \pi(x_t)) ds dF_{x_t, x_{t+1}}(t|\pi(x_t)) \tag{2.15}$$

where $R(x_t, x_{t+1}, u_t)$ is the reward when transitioning from x_t to x_{t+1} after performing action u_t, adjusted for the unknown transition time. Here, $\rho(x_t, \pi(x_t))$ represents the mean reward during the transition following policy, π. To obtain ρ, intermediate rewards are calculated at each time step in the transition period to explicitly capture transition information. $F_{x,x_{t+1}}(t, u)$ denotes the probability distribution of the transition time from x_t to x_{t+1}. Finally, $\beta > 0$ is the *constant* discount factor in SMDPs. High β results in short-sighted agents. In SMDPs, the transition time is no longer constant. Thus, the discount factor is corrected for transition time during each update step. The corrected discount factor is:

$$\gamma(x_t, x_{t+1}, u) = \int_0^\infty e^{-\beta t} dF_{x_t, x_{t+1}}(t|\pi_t) \tag{2.16}$$

where $\gamma(x_t, x_{t+1}, u_t)$ is the transition time adjusted discount factor. The value function for SMDPs is obtained through combining Equations 5.39 and 2.12:

$$v_\pi(x_t) = \frac{1 - e^{-\beta\tau}}{\beta} R(x_t, x_{t+1}, \pi(x_t)) + e^{-\beta\tau} v_\pi(x_{t+1}) \tag{2.17}$$

where τ denotes the unknown transition time. The action-value variant is given by:

$$q_\pi(x_t, u_t) = \frac{1 - e^{-\beta\tau}}{\beta} R(x_t, x_{t+1}, \pi(x_t)) + e^{-\beta\tau} q_\pi(x_{t+1}, u_{t+1}) \tag{2.18}$$

By representing control environments as SMDPs, policies resulting in large overshoot, inverse response, or other undesirable dynamics will be minimized. Additionally, SMDPs can handle unknown transition times. An intuitive example of SMDPs in process control is as follows:

> Suppose a CSTR in a refinery must maintain a temperature of 200° C. The temperature is regulated using cooling water via a heat exchanger. A RL agent was tasked with maintaining the temperature set point. Suppose the CSTR is initiated at 220° C. Agents using FOMDP representations may be overly aggressive and send large control actions because the reward is calculated *right before* the next evaluation step. Therefore, input signals resulting in large overshoot or inverse response may be missed during the reward calculation. Contrarily, SMDPs consider the average reward accumulated throughout the transition to provide feedback to the agent, allowing the undesirable dynamics to be captured. Furthermore, the sampling time of SMDPs are not fixed (traditional representations evaluate after a set time period), enabling re-evaluation during the transitional period if unexpected events occur. In such scenarios, the discount factor will also be adjusted in accordance to the elapsed time from last evaluation.

2.2.4 Optimal Solution of the MDP

The optimal solution to the RL problem refers to identifying a policy that generates the highest long term returns. Such a policy may not be unique; there may exist many optimal policies, where $v_{\pi_1^*} = v_{\pi_2^*} = ... = v_{\pi_N^*}$. Formally, the optimal policy must satisfy the **principle of optimality**: the optimal policy π^* is optimal

if and only if $v_{\pi^*}(x) \geq v_{\pi \neq \pi^*}(x)$ for all $x \in \mathcal{X}$ [26]. Mathematically, the optimal value function is:

$$v^*(x) \doteq \arg\max_{\pi} v_\pi(x), \forall x \in \mathcal{X} \tag{2.19}$$

with its action-value variant being:

$$q^*(x,u) \doteq \arg\max_{\pi} q_\pi(x,u), \forall x,u \in \mathcal{X},\mathcal{U} \tag{2.20}$$

In a more explicit form, the optimal value function and action-value function written in terms of Equations 2.12 and 2.14 are given, respectively, by [7]:

$$v^*(x) = \arg\max_{u} \mathbb{E}[R_{t+1} + \gamma v^*(X_{t+1}) | X_t = x, U_t = u] \tag{2.21}$$

$$q^*(x,u) = \mathbb{E}\left[R_{t+1} + \gamma \arg\max_{u_{t+1}} q^*(X_{t+1}, u_{t+1}) | X_t = x, U_t = u\right] \tag{2.22}$$

where the *max* operation denotes that the optimal action will be taken for the remaining of the trajectory. Theoretically, all optimal value functions can be explicitly solved using Equation 2.21; however, such a task would require unreasonable amounts of computation power for even simple systems. In the following section, three popular methods will be introduced to estimate the value and action-value functions in reinforcement learning.

2.3 The Reinforcement Learning Problem

In general terms, reinforcement learning in an industrial setting is simply an agent undergoing meaningful interactions with the process to learn an optimal operating policy. For added intuition, Figure 2.3 shows the information flow of an agent in process control. First, the agent observes some states, $x_t \in \mathcal{X}$, from the environment (some states may be unobservable). Given x_t, the agent performs some controls actions, $u_t \in \mathcal{U}$ and receives a scalar reward signal, $r_{t+1} \in \mathcal{R}$. Finally, the process will transition to some new states, x_{t+1}, given probability $P(x_{t+1}, r_{t+1} | x, u)$.

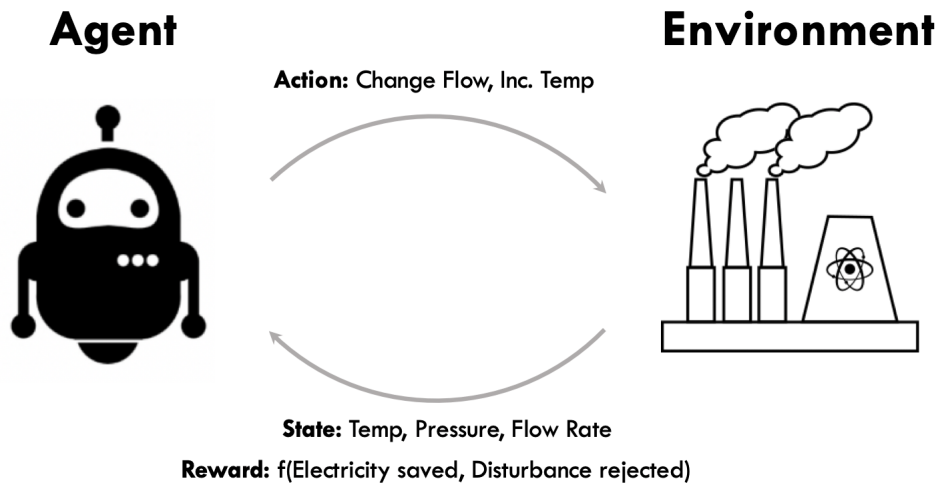


Figure 2.3: Basic setup of reinforcement learning where an agent interacts with the system.

The three main branches of reinforcement learning solutions are shown in Figure 2.4. Starting from the left, dynamic programming (DP) methods can identify the exact value functions, but require a *perfect system model* and is extremely computationally expensive, even for trivial tasks. Comparatively, both Monte Carlo (MC) and temporal difference (TD) methods are approximate DP methods. As such, they are less computationally demanding. Additionally, MC and TD methods do not assume the presence of a system model and identifies the value functions through interactions with the environment. MC methods find the value functions through averaging the returns generated over many sampled trajectories of states, actions, and rewards. One drawback is the significant variance in the sampled trajectories. Consequently, this may lead to poor reproducability in highly noisy systems. TD methods combine the best characteristics of DP and MC methods into one unifying approach. Like MC methods, TD learn from sampled data. Like DP methods, TD performs update steps after each step. However, TD methods typically exhibit large bias (especially during initial learning episodes) due to estimating values through previously estimated values (known as bootstrapping). The general details of each method will be shown throughout this section. For a comprehensive introduction to each algorithm, see [7].

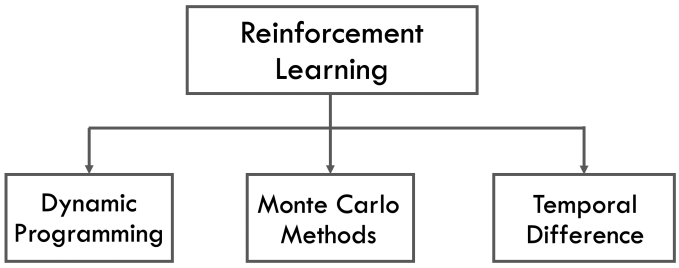


Figure 2.4: The sub-components of machine learning.

2.3.1 Dynamic Programming Methods

Dynamic programming algorithms identify the exact value functions through an iterative procedure using the system dynamics function. In real life applications, DP algorithms are rarely used due to their unreasonable computational cost for even trivial problems. Nevertheless, the ideas of DP serve as the fundamentals for modern approaches. Policy iteration and value iteration are two common techniques in DP.

As an overview, policy iteration searches for the optimal policy by iterating through infinitely many policies, $\pi \in \Pi$, storing only the policy corresponding to the highest cumulative returns. The optimal policy is assumed to be found when G_π can no longer be improved. Policy iteration is comprised of two phases: policy evaluation and policy improvement. **Policy evaluation** computes the value functions and cumulative returns of the system under π through an iterative approach. Value functions are initialized as 0, and are solved iteratively using:

$$v_{k+1,\pi}(x) = \mathbb{E}_\pi[R_{t+1} + \gamma v_{k,\pi}(x_{k+1})] \tag{2.23}$$

$$v_0(x) = 0,\ \forall x \in \mathcal{X}$$

where k denotes the k^{th} update. Here, $v_{k+1,\pi}(x)$ is the predicted value function for x under policy π after $k+1$ update steps. As $k \to \infty$, $v_k(x) \to v_\pi(x)$ for all $x \in \mathcal{X}$ (i.e., the value functions converge to the true value functions under π). However, there often exists a π' where $v_{\pi'}(x) \geq v_\pi$. **Policy improvement** identifies such

situations. Once identified, current policy π will violate the principle of optimality, hence deeming it ineligible for being the optimal policy. Then, the value functions of π' will be identified in the next policy evaluation. This procedure will continue iteratively and infinitely until a policy where $v_{\pi^*}(x) \geq v_{\pi \neq \pi^*}(x)$ for all $x \in \mathcal{X}$ is found. After such a policy is identified, it is regarded as the optimal policy.

Figure 2.5 shows a visualization of the policy iteration algorithm. It can also be described using the following [7]:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} ... \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} v^* \tag{2.24}$$

where $\xrightarrow{\text{E}}$ and $\xrightarrow{\text{I}}$ denotes the policy evaluation and policy improvement steps, respectively. From Figure 2.5, the agent starts with some arbitrary policy and performs policy evaluation. Initially, a large gap exists between V_π and π. As the iterative procedure proceeds, the gap is continuously reduced until $V_\pi, \pi \rightarrow V^*(x), \pi^*$. In industrial applications, the required iterative procedure for each policy evaluation is far too expensive for any non-trivial tasks.

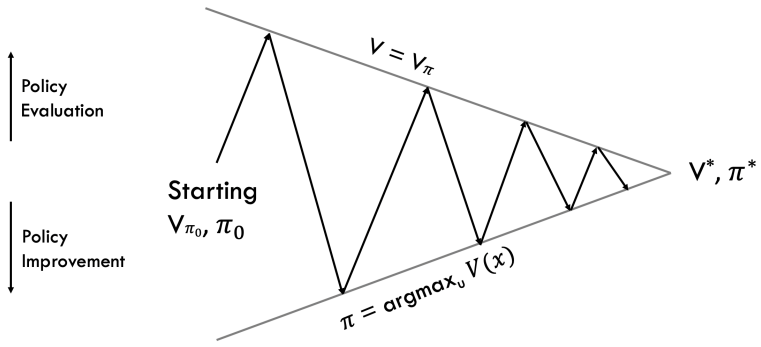


Figure 2.5: A visualization of the policy iteration algorithm. Original image from [27].

To improve upon these computational issues, value iteration was proposed. Value iteration finds the optimal policy through identifying the optimal value functions instead. Intuitively, value iteration is a special case of policy iteration where the policy evaluation is terminated after one step. From the value functions of each state, π^* can be found by traversing through the states corresponding to the highest values. Note that the optimal policy can only be found using $V(x)$ if a dynamics

equation of the system is provided. Without it, $Q(x, u)$ must be identified instead to behave optimally. The policy evaluation for the value iteration algorithm is given as:

$$v_{k+1}(x) = \max_u \mathbb{E}[R_{t+1} + \gamma v_k(x_{t+1})] \tag{2.25}$$

$$q_{k+1}(x, u) = \mathbb{E}[R_{t+1} + \gamma \max_{u_{t+1}} q_k(x_{t+1}, u_{t+1})] \tag{2.26}$$

Here, the *max* operation ensures that each $v_k(x)$ is updated using only the maximizing action so the *optimal* value function can be identified. After all $v^*(x)$ are identified, an agent can behave optimally starting in any state assuming the agent takes the maximizing action at each time. Note that both policy and value iteration are *bootstrap* methods. Bootstrapping in RL increases data efficiency while capturing long-term trajectory information; however, the method also introduces unintended biased.

In industry, both policy and value iteration have limited utility because their updates are far too computationally expensive. In high dimensional settings, even one iterative step may be intractable; therefore, even with value iteration's reduced computational complexity, it is still infeasible for most complex problems. Asynchronous dynamic programming methods further reduces computational complexity by only updating frequently visited states. However, agents are rendered hopeless in states that are rarely encountered. Although, such a methodology mimics human behaviour where encounters can be handled effectively and efficiently and more exotic situations may catch us by surprise. Nonetheless, such methods still require system models to be explicitly provided, an extremely rare case in industry.

2.3.2 Monte Carlo Methods

Monte Carlo methods no longer require explicit system models (a characteristic known as *model-free*). Instead, MC methods *estimate* the average returns for different policies through sampling infinitely many sequences of states, actions, and rewards. As the samples increase, $v_k(x) \to v_\pi(x)$ for all $x \in \mathcal{X}$. Learning-wise, the

average returns are updated at the end of each trajectory. Due to this, the finite tasks with explicit terminal states are typically solved using MC methods. For example, discrete manufacturing is an episodic task in process control. The system is reset after the assembly of each object (cars, toys, ...). In episodic tasks, the value functions are updated naturally after each episode. However, most tasks in process control are continuous. Training a continuous agent using MC methods require additional modifications. One method is to pre-specify a length of time. After the time has elapsed, the agent will pause and update its value functions.

Policy search in MC methods is similar to policy iteration. There exists three differences: 1) only visited states are updated; 2) updates use *sampled* data instead of a model; 3) $q_\pi(x, u)$ is required and identified instead of $v_\pi(x)$. In MC methods, the action-value functions are identified because a model is not provided to the agent. Hence, the agent cannot behave optimally using only the value functions because the actions required to transition to the high value states are not known. Instead, action-values contain explicit information on the expected returns for each action in each state. The iterative procedure of MC to compute the cumulative returns is given by:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} ... \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} q_{\pi_*} \tag{2.27}$$

Intuitively, the agent is initiated in an unknown system and follows a certain policy, π, to traverse throughout the state space while collecting rewards after each decision. Eventually, the agent will reach a terminal state and conclude the episode. Upon termination, a sequence of returns $G_1, G_2, ..., G_{n-1}$ can be generated using the received reward signals:

$$G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + ... + \gamma^{n-1} R_n$$

$$G_2 = R_2 + \gamma R_3 + \gamma^2 R_4 + ... + \gamma^{n-2} R_n$$

$$G_3 = R_3 + \gamma R_4 + \gamma^2 R_5 + ... + \gamma^{n-3} R_n$$

$$\vdots$$

$$G_{n-1} = R_n$$

or:

$$G_m = \sum_{i=0}^{n} \gamma^i R_{m+i} \tag{2.28}$$

where G_m denotes the discounted cumulative return received on the m^{th} step. Using G_m, the action-values can be computed for each step by:

$$Q_{k+1}(x,u) = Q_k(x,u) + \frac{1}{k}\left[G - Q_k(x,u)\right] \tag{2.29}$$

where $Q_k(x,u)$ represents the k^{th} action-value update and G corresponds to the returns received after performing action u in state x. Notice that as $k \to \infty$, $\frac{1}{k} \to 0$; therefore, this set-up is ineffective in non-stationary settings because the updates get infinitely small. To extend Equation 2.29 to non-stationary problems, $\frac{1}{k}$ is changed to a constant given by α:

$$Q_{k+1}(x,u) = Q_k(x,u) + \alpha\left[G - Q_k(x,u)\right] \tag{2.30}$$

where $\alpha \in (0,1]$ is known as the learning rate (also called step size). The lower bound prevents α from approaching 0; therefore, allowing for continually adaptation in non-stationary problems. After each update of Equation 2.30, a new episode starts and the procedures are repeated. As k, # of episodes $\to \infty, Q(x,u) \to q(x,u)$. Once $Q(x,u)$ converge, online action selection can be conducted by:

$$\pi^*(x) = \arg\max_u q(x,u) \tag{2.31}$$

That is, π^* is performing the greedy action in each state.

Exploration in MC

Notice that bootstrapping is not used in MC methods. In fact, all value functions are estimated independently. As such, MC methods do not suffer from bias issues; however, MC methods may suffer from large variances instead caused by the noise in each sampled trajectory [7]. Moreover, exploration is mandatory in MC methods because the dynamics of the system are unknown to the agent. Through exploration, the agent can discover the dynamics of the system and the value functions for each state. Typically, exploration in MC methods is conducted by initiating the agent in a random state at the beginning of each episode. After infinite episodes, all states will be visited infinitely many times.

MC methods allow the agent to learn solely from sampled data; however, the action-values are updated only after each episode. Such a procedure is unnatural in continuous systems (most systems in process control), disadvantageous long episode systems, and is not intuitive to human behaviour. For example, humans learn immediately after feedback, not in pre-set increments. Temporal difference methods combine the best features of DP and MC methods into one unifying algorithm.

2.3.3 Temporal-Difference Methods

Temporal difference (TD) methods are mathematically simple and cheap computationally compared to MC and DP methods. TD methods learn from experiences (like MC methods) and bootstraps (like DP methods). Furthermore, a dynamics model is not required in TD methods. Instead, the agent learns the dynamics from interactions. Moreover, TD methods update their value functions immediately after x_{t+1} and R_{t+1} are received. The TD update algorithm for value and action-value functions are given in Equations 2.32 and 2.33, respectively [28]:

$$V(x_t) \leftarrow V(x_t) + \alpha \left[R_{t+1} + \gamma V(x_{t+1}) - V(x_t)\right] \tag{2.32}$$

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha \left[R_{t+1} + \gamma Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)\right] \tag{2.33}$$

where $\leftarrow$ is the update operator. At each update, the old value function is corrected as a function of the *TD error* by a fixed amount determined by α. The TD errors are at each time t is given as:

$$\delta_t = R_{t+1} + \gamma V(x_{t+1}) - V(x_t) \tag{2.34}$$

$$\delta_t = R_{t+1} + \gamma Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t) \tag{2.35}$$

The first two terms, $R_{t+1} + \gamma V(x_{t+1})$, denote the predicted value function for x in accordance with the last interaction. $V(x_t)$ is the previously predicted value function for x. After infinitely many interactions with the system, $V(x_t) \rightarrow v(x_t)$ (i.e., the estimated values converge to the true values). The action-values follow the same procedure. After convergence of the values and/or action-values, the optimal action selection is given in Equation 2.31.

Exploration in TD

Like MC methods, TD methods are also *model-free*; therefore, action-values are required for the agent to act optimally and exploration is mandatory. A simple and common exploration method used in TD methods is the ϵ-greedy action selection. Here, the agent performs the greedy action with a $\epsilon \in [0, 1]$ probability of performing a random action. During training, ϵ is typically decayed throughout training. At the beginning, ϵ starts at a high value because agent knows nothing. Eventually, ϵ decays to a low value when training is almost complete.

Unfortunately, random exploration is sample inefficient and may require the agent to undergo thousands of interactions before learning anything meaningful. Learning can typically be significantly accelerated through a heuristics function $\mathcal{H} : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$ [29]. One such heuristics approach is the upper confidence bound (UCB) action selection algorithm [30]. Here, exploration is promoted on states that

have high potential to be optimal and is given by:

$$U_t = \arg\max_u [Q_t(x, u) + \mathcal{H}] \tag{2.36}$$

The heuristics function here is given by:

$$\mathcal{H} = c\sqrt{\frac{ln\ t}{N_t(x, u)}} \tag{2.37}$$

where c is the degree of exploration. Large c values promote greater degrees of exploration. Furthermore, N_t is the number of times action u was selected prior to time t. As $N_t(x, u) \to \infty$, the corresponding $Q(x, u)$ has been updated many times and becomes very accurate. Hence, the heuristics function $\mathcal{H} \to 0$.

Popular TD Algorithms

The two most popular TD algorithms are **SARSA** and Q**-learning**. **SARSA** is an *on-policy* algorithm. In such algorithms, the *behaviour policy* and *target policy* are identical. Target policy refers to the goal policy of the agent. Typically, this is the optimal policy. Conversely, the behaviour policy, $b(u|s)$, is the policy used by the agent for decision making. In cases where the target and behaviour policy are identical, the agent is *on-policy*. One flaw with *on-policy* agents (assuming the target policy is the optimal policy) is that during training, the agent may quickly converge to a local optimum and never explore (since any policies containing exploration is not the optimal policy). Ultimately, this results in a sub-optimal solution. Contrarily, *off-policy* agents, like Q**-learning**, typically follow exploratory policies during training to conduct deep exploration. Then in online applications, the policy is swapped to the optimal policy. Moreover, *off-policy* agents are *guaranteed* to find the optimal policy assuming each state-action pair is visited infinite times and $b(u^*|s) > 0$ (i.e., probability of picking the optimal action under the behaviour policy is not 0) [28].

Since SARSA is *on-policy*, the action-value function are updated using Equation

2.33 using the quintuple $(x_t, u_t, R_{t+1}, x_{t+1}, u_{t+1})$. Q-learning updates use only four parameters $(x_t, u_t, R_{t+1}, x_{t+1})$ through Equation 2.38:

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha \left[R_{t+1} + \gamma \arg\max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t) \right] \tag{2.38}$$

In Q-learning, u_{t+1} is not required because the action taken might follow a different policy compared to the target policy since the algorithm is *off-policy*. Instead, Equation 2.38 uses the *max* operation to ensure Q-values are still updated towards the optimal policy. Ultimately, TD methods unify DP and MC methods, allowing the agent to learn from experiences and perform inter-episode updates to exploit the most recent learnings.

A detailed numerical example is provided in Chapter 4 where a tabular Q-learning algorithm was applied onto an industrial VFD system to conduct set-point tracking control.

2.4 Summary of DP, MC, and TD

The main features of DP, MC and TD methods are summarized in Table 2.3. Overall, DP requires a dynamical model of the system to compute the value functions while both MC and TD methods can learn directly from interactions with the system. Both DP and TD methods use bootstrapping to estimate value functions; that is, they estimate the current value function based on previously estimated values. Bootstrapping is data efficient, but introduces large biases to the estimated values, especially in the early episodes. Conversely, MC methods estimate the value functions of each state independently through sampling many system trajectories. However, this method, instead, introduces high variance. For extremely noisy systems, the reproducability of the results may be low. Comparing the computational cost, DP methods require much more compared to MC or TD since all value functions are simultaneously solved. In MC methods, only the value functions that were visited in the sampled trajectories are updated. Additionally, updates are con-

ducted at the end of each episode and not after each step. Similar to DP methods, TD methods update the value function immediately after an experience; however, only the value function corresponding to the last visited state is updated. In terms of exploration, DP methods do not explore the system model (both transition probabilities and expected reward) is explicitly provided. MC methods explore by being initiated in a random state after each episode termination. In TD methods, agents explore by occasionally performing a random action.

Table 2.3: A comparison of DP, MC, and TD methods.

	Dynamic Programming	**Monte Carlo**	**Temporal Difference**
Requires model	Yes	No	No
Estimate bias	High	Low	High
Estimate variance	Low	High	Low
Computational cost	High	Medium	Low
$v(x)$ update	All states simultaneously	After a trajectory	After an experience
Exploration	Not needed, all states update	Random initialization	Performing a random action

2.5 Reward Design for Process Control

The design of the reward function for process control applications is similar to MPC. For regulation or set-point tracking problems, the MSE reward function can be used and is given by:

$$r(x, u) = -(x_i - x_{sp})^2 \tag{2.39}$$

However, the agent may find it difficult to distinguish between small off-sets using this reward function. For example, when the tracking error is 10, the reward is -100. However, if the off-set is only 0.25 or 0.1, the agent would find it difficult to distinguish between the small rewards because the difference is miniscule compared to an error of 10. To enhance this distinction, a Huber loss can be used [31]:

$$r(x, u) = \begin{cases} x_t - x_{sp} & if\ |x_t - x_{sp}| > 1 \\ (x_t - x_{sp})^2 & otherwise \end{cases}$$

In this case, large errors are not squared, significantly reduce their magnitude. In the tabular cases, this error works exceptionally well; however, not so much in deep RL. Typically, the inputs to neural networks are normalized for sufficient learning [32]. When normalizing the rewards, small errors will once again become indistinguishable. In such a case, the following reward function typically works better:

$$r(x,u) = \begin{cases} x_t - x_{sp} & if\ |x_t - x_{sp}| > 1, \\ (x_t - x_{sp})^2 & if\ 1 \geq |x_t - x_{sp}| > \eta, \\ +1 & otherwise \end{cases}$$

where η is the maximum acceptable tracking error. Here, as the agent achieves states within η, the rewards are significantly increased. Such an idea is similar to zone MPC, where the objective of the controller is to guide the trajectory within a zone [33]. Another flaw with deep RL comes from the noisy action signals. For example, a normal human would not go and continuously change the air conditioning set-point if the temperature inside a house varies between 22.1° C to 22.2° C because the two temperatures are relatively the same. In the case of deep RL, these are seen as two completely different states, and correspond to (slightly) different actions. One could design a filter to remove such small actions from being sent to the system; however, adding a cost to the change in inputs is a more natural way to mitigate this:

$$r(x,u) = -[(x_t - x_{sp})^2 + \nu \Delta u_t^2] \tag{2.40}$$

where Δu_t is the change in input in the last sampling time. The coefficient, ν, is used to tune the effect of the action on the reward. For example, if the system's input signals are typically small, a large ν would be used so the tracking error does not dominate the entire reward function.

In exotic scenarios where the optimal input is known, the reward function can become:

$$r(x,u) = -[(x_i - x_{sp})^2 + (u_i + u_{ss})^2] \tag{2.41}$$

Lastly, the rewards are sometimes clipped to avoid large TD errors causing numerical issues during bootstrapping [34]. From Equation 2.38, it can be seen that if $R(x,u) >>> Q(x,u), Q(x_{t+1}, u_{t+1})$, then the updated $Q(x,u)$ would be completely dominated by $R(x,u)$. Additionally, any future updates bootstrapping off $Q(x,u)$ would subsequently become dominated by its value. As such, rewards are clipped (bounded) within a range to prevent such issues. Unexpectedly large rewards may originate from incorrect sensor readings, which consequently leads to inaccurate reward signals being sent to the agent. Reward clipping is conducted by:

$$r(x,u) = min(max(r_t, \mu^-), \mu^+) \tag{2.42}$$

where μ^- and μ^+ denotes the minimum and maximum rewards, respectively.

2.5.1 Reinforcement Learning vs. Other "Learnings"

Reinforcement learning is a unique class of machine learning. An ideal supervised learning model can only be as good as the subject matter expert providing the labels to the data set, which may not be 100%. For example, in a complex control task, the control law is usually highly non-linear. Control experts can try to provide control strategies for such systems, but optimality may not be guaranteed for highly non-linear systems. Also, supervised learning is used to generalize responses for occurrences not present in the data [7]. Reinforcement learning works by directly interacting with the environment *without labels*. Through adequate exploration, reinforcement learning will identify peculiar features to optimally control such problems [citation required]. Reinforcement learning is *similar* to unsupervised learning in terms of identifying hidden structures within the environment. However, reinforcement learning tries to maximize an internal scalar reward signal, rather than purely data mining.

Evolutionary methods, a family of optimization algorithms such as genetic algorithm, are most similar to reinforcement learning. For a control problem, such

methods can apply multiple static policies for different operating regimes [7]. Policy search is conducted by first initiating k random input trajectories of length N, generating input matrix $\mathbb{U}_{[k,N]} \in \pi$. Subsequently, the loss, J_U, of each U is calculated based on the objective function. Input trajectories with the lowest loss move onto the next generation and generates new pseudo-random input trajectories. This process is repeated until optimal policy, π^* is found for each operating regime [35].

Evolutionary methods work well when the policy space is sufficiently small, easy to find, or a lot of time is available for optimization. The biggest advantage of such methods compared to reinforcement learning is that the whole state does not need to be known. However, such methods does not capture the reinforcement learning fundamentals of mapping $X \rightarrow U$. Unlike evolutionary methods, reinforcement learning keeps memory of each individual interaction making it a more data efficient approach [7].

2.6 Function Approximation

2.6.1 Introduction to Function Approximations

Prediction models have wide applications in all sectors of the economy. To obtain the highest possible accuracy, one can simply have an infinitely large repository of previous examples. When given any input, a suitable output can be generated by finding the exact solution in the repository. For example, if the task is to predict the model of an automobile based on a picture and the dimensions of a car, one could obtain 100% accuracy so long as every single car specification exists in a repository. This idea sounds good in theory, but is only possible in real life if there exists infinite memory. Function approximation aims to solve this problem by generating a model to generalize across a massively large repository of historical data. Intuitively, the model stores the information at a much lower space complexity, for a cost of some reproduction error. There is also a trade-off between the reduced space complexity and the reproduction error. Large, complex models have increased space complexity

but reduced error while small, simple models exhibit the opposite. This section focuses on complex neural network models with high predictive capabilities. In RL, function approximation is typically used to approximate the (action-) value functions or the policy itself.

2.6.2 Neural Network Basics

Neural networks are highly non-linear models that explore the individual and interaction effects of each variable with all other variables [32]. The general structure of a neural network is shown in Figure A.16. Neural networks are comprised of an input layer, some hidden layer(s), and an output layer. The input layer consists of the input data, while the hidden layer(s) and output layer consists of fitted weights and biases, $W_{n_x \times n_b}$ and $b_{n_b \times 1}$, respectively. Here, n_b and n_x denotes the batch size and the dimension of the input layer, respectively. In Figure A.16, x_m denotes the m^{th} input variable. The superscript and subscript of a denotes the hidden layer number and the node number in the corresponding layer, respectively. Subscript m_1 to m_r denotes the number of nodes in hidden layers 1 to r, respectively. Finally, superscript o denotes the output layer.

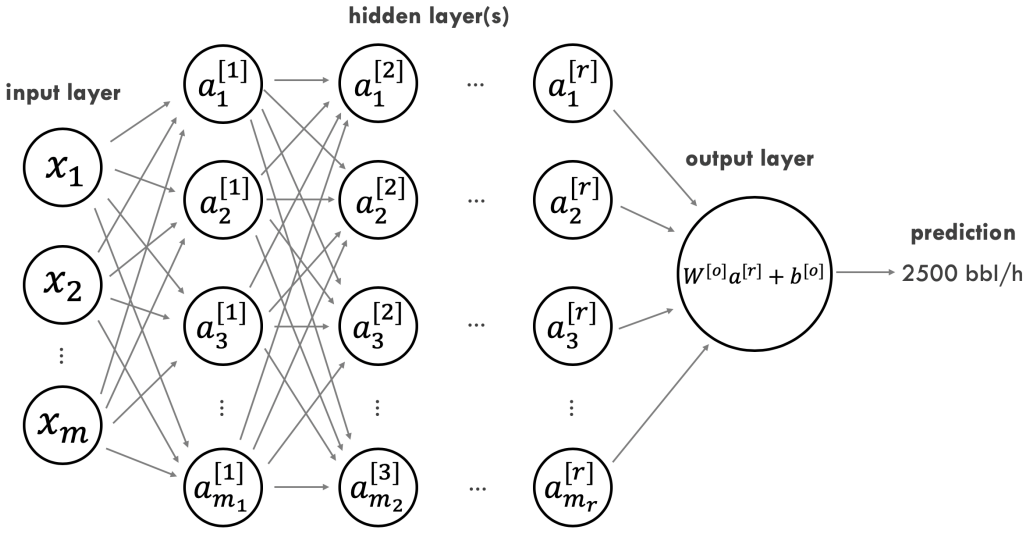


Figure 2.6: Structure of a general neural network.

The details within a hidden layer's node is shown in Figure A.17. First, the outputs from the previous layer's nodes are inputted and multiplied by the weights

of the current node. The current node's bias is then added. If the current node is in the first layer, the outputs from the previous layer is replaced with the input variables. Afterwards, the output is sent to an action function to provide the non-linearity for any neural network model. In this thesis, the rectified linear unit (ReLU) activation function is typically used and is given by:

$$a_j^{[i]} = \begin{cases} y, & \text{if } y \geq 0. \\ 0, & \text{otherwise.} \end{cases} \tag{2.43}$$

where i and j denotes any hidden layer and any node number, respectively. Two other popular activation functions are sigmoid and tanh given in Equations 2.44 and 2.45, respectively.

$$a_j^{[i]} = \frac{1}{1 + e^{-z}} \tag{2.44}$$

$$a_j^{[i]} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.45}$$

where e denotes the exponential operator and $z = Wx + b$. The sigmoid and tanh activation functions have lost popularity in recent years because they create the exploding/vanishing gradient effect. This effect occurs because the derivatives of both the sigmoid and tanh functions are zero outside of a small section. Additionally, the derivative at the inflection point is infinity. Since neural networks are trained using backpropagation, often times, the gradient of the loss function becomes zero as it is backpropagated through the neural network during training. Ultimately, this leads to significant difficulties in training neural networks (especially deep networks) [36].

Mathematically, for one example with input vector x:

$$\begin{aligned} z_j^{[1]} &= W^{[1]}x + b^{[1]} \\ a_j^{[1]} &= ReLU(z_j^{[1]}) \\ z_j^{[2]} &= W^{[2]}a_j^{[1]} + b^{[2]} \\ a_j^{[2]} &= ReLU(z_j^{[2]}) \end{aligned}$$

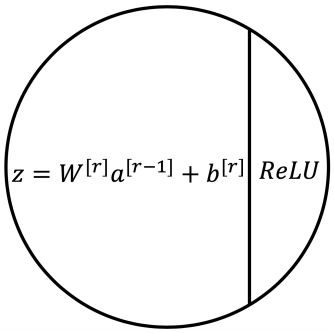


Figure 2.7: Inside a hidden layer's node.

$$\vdots$$
$$z_j^{[r]} = W^{[r]}a_j^{[r-1]} + b^{[r]}$$
$$a_j^{[r]} = ReLU(z_j^{[r]})$$
$$y = W^{[o]}a_j^{[r]} + b^{[o]}$$

Neural Network Initialization

Neural networks can be initiated in many ways. Although neural networks can be initiated as all zeros, such an approach is not symmetry-breaking resulting in all neurons performing the same calculations [32]. Ultimately, this results in all neurons outputting the same values rendering the whole network useless. Therefore, a primitive approach to overcome this was to initialize the neural network weights as random near-zero values. This method was symmetry-breaking, but such networks required long training times, especially in deep learning [37]. In 2010, Xavier and Bengio published one of the first papers to explicitly study neural network initialization.

In [37], the the Xavier initializer was proposed to equalize the variance of the outputs of each layer with the variance of its inputs. More specifically, the biases of each layer was initialized as zero, but the weights were initialized as:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{2.46}$$

where $U[-a, a]$ represents an uniform distribution bounded between $(-a, a)$. Here, n_j and n_{j+1} denotes the size of the previous and current layers. The derivation of

Equation 2.46 assumed linear actions. When tested using sigmoid activation functions, the Xavier initialized neural networks showed substantially faster convergence times. Unfortunately, sigmoid activation functions were considered obsolete as time went on due to the exploding/vanishing gradients problem [36].

By 2015, He et al. proposed a new initialization method specialized for ReLU activation functions, known as the He initializer [38]. He extended upon previous work by assuming a ReLU activation function instead of a linear one and obtained the weight initialization function given by:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_j}\right) \tag{2.47}$$

where $\mathcal{N}$ denotes the Gaussian distribution and $\frac{2}{n_j}$ denotes its standard deviation. Like in [37], the He initialization showed substantially faster convergence times for neural networks compared to previous methods when using the ReLU activation function.

The advantages of each initialization are summarized in Table 2.4

Table 2.4: Comparing different neural network initialization methods.

Zero init.	**Random init.**	**Xavier init.**	**He init.**
Does not work	Simple but slow	Ideal for sigmoid activations	Ideal for ReLU activations

2.6.3 Cost Function for Neural Networks

MSE is the typical cost function for regression tasks and is given by [32]:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.48}$$

where J represents the loss. Here, n denotes the number of samples in the current optimization step. $\hat{y}_i$ and y_i are the i^{th} predicted and actual labels, respectively. The MSE cost function is typically selected due to its convex nature [39].

Gradient Descent

Given the cost function, the model parameters are updated using gradient descent. The general gradient descent formulation is given by Equation A.6.

$$\theta_j^{m+1} \leftarrow \theta_j^m - \alpha \frac{\partial J}{\partial \theta_j} \tag{2.49}$$

where θ_j denotes the j^{th} parameter (parameter includes both weights and biases) of the model. Here, m represents the m^{th} update of gradient descent and α is the learning rate. Unfortunately, gradient descent can optimize quite slowly, especially for neural networks where the solution is highly non-convex. There are many different enhanced gradient optimization methods such as momentum gradient descent, AdaGrad, RMSprop, etc; however, adaptive momentum gradient descent (ADAM) method will be used for the remainder of this thesis [40]. Mathematically, ADAM combines momentum gradient descent and RMSprop into one unifying algorithm. ADAM improves upon Equation A.6 by computing an adaptive learning rate for each parameter [40]. To do so, the exponentially decaying average of the past gradients and squared gradients of the weights and biases are computed and stored using Equations A.7 to A.10.

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW \tag{2.50}$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \tag{2.51}$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2 \tag{2.52}$$

$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \tag{2.53}$$

where V and S are the estimates of the gradient and squared gradients, respectively. V and S are typically initiated as zero vectors and are heavily biased towards zero at initial steps. Hence, the biases (numerical bias, not the neural network parameter

bias) for the initial terms are corrected using:

$$V_{dW}^{corrected} = \frac{V_{dW}}{1 - \beta_1^t} \tag{2.54}$$

$$V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t} \tag{2.55}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1 - \beta_2^t} \tag{2.56}$$

$$S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t} \tag{2.57}$$

Combining the above equations, the weights and biases are updated by:

$$W_j \leftarrow W_j - \alpha \frac{V_{dW}^{corrected}}{S_{dW}^{corrected} + \epsilon} \tag{2.58}$$

$$b \leftarrow b - \alpha \frac{V_{db}^{corrected}}{S_{db}^{corrected} + \epsilon} \tag{2.59}$$

where ϵ is a small scalar to avoid division by zero. The authors proposed values of 0.9, 0.999 and 10^{-8} for β_1, β_2, and ϵ, respectively [40]. Next, the amount of data that will be used to compute the loss gradient will be explored.

Mini-batch Gradient Descent

Classically, the gradient of the loss function was computed using all data available. Furthermore, this method (called batch gradient descent) guarantees monotonic improvements in performance after each update step [39]. However, BGD suffers from space complexity and is infeasible in big data applications. Thus, mini-batch gradient descent was used for the work in this thesis. Mini-batch gradient descent fits between stochastic gradient descent (SGD) and BGD, where small batches of data sampled from the original data set are used to perform stochastic updates at each step [41]. Mini-batch gradient descent offers three benefits over the previous methods: i) less computationally demanding compared to batch gradient descent; ii) more accurate loss function gradient for parameter updates compared to SGD;

iii) requires less steps compared to SGD.

Data Segregation

The data set was split into three sections for machine learning: training, validation, and testing. The partition and description of each section is shown in Table A.6. The training data set was used to identify the machine learning model(s). Then, the model was validated on unseen data via the validation data set (sometimes called development data). The error of the model on the validation data set, $e_{validation}$, was then evaluated and compared to the training data error, e_{train}. If the difference is large, the model was rebuilt using different data pre-processing techniques and features. This step was repeated until $e_{train} \approx e_{validation}$ to ensure that the model did not overfit to the training data. Finally, the model was tested on the testing data to explore the performance of the model in live production. Testing data was always the last 5% of the data set.

Table 2.5: Description of each data partition.

	% of Data	Description
Training	90%	Identify the ML model
Validation	5%	Tune ML model performance on unseen data
Testing	5%	Test ML model performance on proxy live data

Regularization

Objectively, supervised learning models attempt to generalize the learnings obtained from the training data set to predict for situations not seen before. For example, suppose there exists a data set that contains the height and weight of a species of dogs. Objectively, the model must predict the weight of the dog given its height. After the model is trained, it should have sufficient capability to predict for the weight of a dog even if the exact height provided was not in the training data set. Often times, the training data set is small and does not represent the whole population of the data set. This ultimately leads to the model overfitting the training data set, resulting in poor generalization characteristics. In machine learning

literature, the model error is often called the **bias**. Similarily, the difference in the modelling error between the training and validation data set is called the **variance** [32]. Models exhibiting high variance are typically overfit to the training data, and does not predict well in production. Regularization aims to significantly reduce variance at only a slight cost to bias. Generally speaking, regularization reduces the likelihood of learning a complex model by penalizing large weights through the objective function. One common method is called the L1 regularization (sometimes called Lasso regularization) where a linear penalty is applied to weights and is given by:

$$J(W) = \frac{1}{n}\left[\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{p} |W_j|\right] \tag{2.60}$$

where λ is a hyper parameter to determine the aggressiveness of the penalty and p denotes the number of parameters inside the model. The L2 regularization is another popular regularization technique and applies a quadratic instead:

$$J(W) = \frac{1}{n}\left[\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{p} W_j^2\right] \tag{2.61}$$

Figure 2.8 shows the optimal solution space of the L1 and L2 regularizations for a two parameter model. Overall, L2 regularization is typically the preferred choice because of its unique, stable solution and invariance under rotation [42]. Another key difference is that L1 regularizations cannot be used for gradient based approaches because it is not continuously differentiable [43].

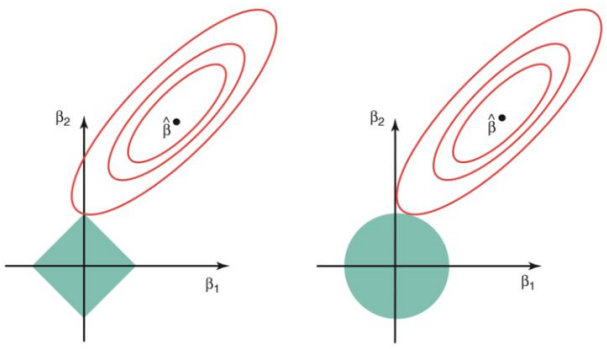


Figure 2.8: Solution space of the lasso (left) and ridge regularization (right). Original image from [44].

Two other popular, but specialized regularization techniques catered towards neural networks are drop-out and batch normalization [45], [46]. Figure 2.9 shows a neural network with and without drop-out. On a high level, drop out randomly disable neurons during training to prevent major "co-adaptation" between adjacent neurons. Intuitively, the drop-out process introduces (significant) pseudo noise into the training step, forcing neurons to learn a more probabilistic mapping. Ultimately, the drop-out method was able to achieve state-of-the-art performance when tested on various data sets in computer vision, natural language processing, classification, and computational biology. A more detailed explanation of drop-out can be found in [45].

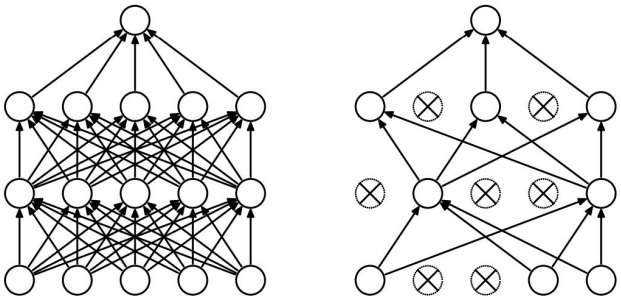


Figure 2.9: A neural network with (right) and without (left) drop-out [45].

Batch normalization is another recently popularized regularization method in deep learning. Neural networks are trained through backpropagation. In this method, the accuracy of the neurons in the later layers are paramount for proper parameter updates in the earlier layers. If not, the backpropagated errors are significant incorrect. Specifically, the distribution of different layer's inputs change during training due to the weight changes of the subsequent layers. This characteristic, called internal covariate shift, contains a significantly negative effect on the training time of neural networks. Batch normalization aims to overcome internal covariate shift by normalizing layer inputs. In doing so, deep neural networks are less sensitive to initializations and much larger learning rates can be used. Additionally, the method was shown to contain regularization effects and often times eliminate the need for drop-out. In experiments, the authors found that neural networks using batch normalization can achieve previous accuracies with 14 times fewer training

steps. For more information on batch normalization, see [46].

2.7 Deep Deterministic Policy Gradient

Deep deterministic policy gradient was introduced as one of the first RL architectures to handle both continuous states and continuous actions [47]. Additionally, it was shown to also work in massively large state and action space systems (one such system was $x \in R^{102}$ and $u \in R^9$). DDPG contains four neural networks and employs an actor-critic framework. The actor is the deterministic policy gradient (DPG) algorithm and maps states to actions. Similarily, the critic is the deep Q-learning network (DQN) algorithm and approximates the action-values of the state-action pairs. Intuitively, combining DPG and DQN into one unifying algorithm overcomes several shortcomings exhibited by each algorithm individual. For example, policy gradients were traditionally trained using MC methods and cannot update mid-episode; however, DDPG trains the DPG using the gradient of the DQN, allowing for inter-episode updates. Furthermore, DQN cannot output continuous actions, but DDPG can by leveraging DPG to select actions. Another advantage provided by DDPG is the mitigation of large variances in the DPG through the use of DQN. Previously, DPG experiences high variance because similar action sequences may return different outcomes in stochastic environments; however, evaluating the action using the DQN (a deterministic function) will result in an unbiased estimate of performance [47].

2.7.1 Actor - Deterministic Policy Gradient

The DDPG leverages the DPG algorithm to deterministically map *continuous* states to *continuous* control actions. Classically, policy gradient algorithms represent the policy as a probability distribution $\pi_\theta(u|x) = \mathbb{P}[u|x;\theta]$ which stochastically maps states to actions [48]. In DPG, *deterministic* policies are considered instead and are given by $u = \mu_\theta(x)$. Comparatively, deterministic policies are more ad-

vantageous because only $Q^{\mu}(x, \mu_{\theta}(x))$ is required during updates steps compared to $\sum_{u} \pi(u|x) Q^{\mu}(x, u)$. Here, $\pi(u|x)$ denotes the probabilities of picking different actions u in x.

Like all experience driven RL algorithms, exploration is required in DPG; thus, requiring some stochastic behaviour policy (ironically making it non-deterministic). To this end, DPG can be trained using an *off-policy* actor-critic method where a deterministic policy is identified while following a separate exploratory policy. Such a concept is analogous to Q-learning, where a deterministic greedy policy is identified while following a noisy behaviour policy (typically ϵ-greedy) during training. For a detailed explanation of DPG, please refer to [48].

2.7.2 Critic - Deep Q-learning Network

In DDPG, DQN is used to reduce variance and provide off-policy training to the DPG. DQN is a deep Q-learning approach to map from states to action-value functions [49], [50]. Historical methods to train deep Q-learning were unstable and data inefficient. Authors of DQN introduced two important concepts in the DQN algorithm—the experience replay and target networks—to significantly improve convergence rate. The *experience replay* is a dictionary of tuples (x, u, r, x'). During training, random mini-batch of experience tuples are sampled to enhance data efficiency (same experiences used many times) and to provide the agent with temporally de-correlated training examples. The target network solves the "moving target problem". In all deep Q-learning approaches, supervised learning models are used to predict for the action-values. Initially, the model is trained using:

$$y_i(x, u, r, x') = r + \gamma \max_{u} Q_{\theta}(x', u') \tag{2.62}$$

$$J(\theta) = \mathbb{E}_{(x,u,r,x') \sim U(R)} \left[(y_i - Q_{\theta}(x, u))^2 \right] \tag{2.63}$$

where Q_{θ} denotes the predicted Q value given model parameters θ. Additionally, $(x, u, r, x') \sim U(R)$ represents sampling experience tuples from the experience replay

following an uniform distribution and y_i denotes the "target" Q value (i.e., the label to the model). From Equation , y_i is also partly calculated from the Q value prediction model. As the model updates, y_i will consistently change, creating an ill-posed minimization problem ultimately resulting in poor learning. In DQN, a *target network* is introduced to prevent this problem. Architecturally, the target network is an exact copy of the original model; however, the model weights are a time-delayed version of the "online" model. That is, the weights of the target network are kept constant for a period of time and are used to compute y_i. By doing this, the target (although inaccurate during initial episodes) remains stationary during the optimization step. After a period of time, the target network copies the weights from the online model and the procedure is repeated until accurate Q-values can be predicted. Typically, this occurs when the target and online network are sufficiently similar.

A more detailed explanation of experience replay is provided below. For complete details, see [51], [52]. For more details the target network or DQN, see [49], [50].

2.7.3 Exploration in DDPG

Traditionally, exploration in continuous action spaces are difficult because classical approaches, such as ϵ-greedy, work only in a discrete action space. DDPG explores through corrupting the action with exploratory noise [47]. Throughout RL literature, many researchers conduct exploration using white noise. The noise corrupted action is given by:

$$u'(x_t) = u(x_t|w_t) + \mathcal{N} \tag{2.64}$$

where $u'(x_t)$ is the action corrupted by some noise, $\mathcal{N}$.

Exploration using White Noise

In Equation 2.64, $\mathcal{N}$ can be white noise drawn from $\mathcal{N}(0, \sigma^2)$; however, white noise is de-correlated and is ineffective for "deep" exploration (i.e., traversing far from the current state) due to the zero averaging effect [53]. Intuitively, white noise simply introduces oscillation into the process and does not create displacement in any particular direction. Therefore, it is more effective to corrupt the action using a temporally correlated process such as the Uhlenbeck-Ornstein (UO) process.

Ornstein-Uhlenbeck Exploratory Noise

The UO process is given as [54]:

$$dx_t = \theta x_t dt + \sigma dW_t, \tag{2.65}$$

where $\theta > 0$, $\sigma > 0$, and W_t denotes the Wiener process. Mathematically, the Wiener process is a special case of a continuous time stochastic process. Detailed information regarding the Wiener process and its properties can be found in [55]. The UO process is ideal for exploratory noise in RL because of its time correlated feature.

Intuitively, actions u_t from the RL agent can be understood as exerting an external force upon physical bodies and is given by [56]:

$$u = m\ddot{x} \tag{2.66}$$

where m and $\ddot{x}$ denotes mass and acceleration, respectively. To obtain displacement (i.e., movement in the state space), the force must be integrated twice:

$$x = \frac{1}{m} \int \int u \tag{2.67}$$

Interestingly, integration operators are low-pass filters and will remove high frequency noise contained in u that are generated by the Wiener process [57]. Con-

sequently, this results in smooth displacements in temporally correlated processes, such as the UO process. Additionally, the displacement will typically stay in the same direction for long durations, allowing for deep exploration the state space. For example, Figure 2.10 shows the trajectory of a randomly generated OU process, and its corresponding effect on the displacement of the agent inside the state space. It can be seen that the displacement is smooth and is heavily biased towards one direction, ultimately promoting deep exploration.

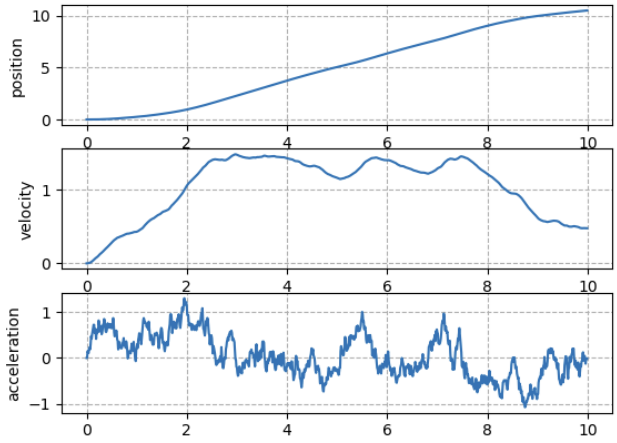


Figure 2.10: Change in displacement caused by a randomly generated OU process.

2.7.4 Stabilization of Training

Architecturally, DDPG contains two interacting neural networks that are trained upon each other. To successfully train such a complicated system, careful parameter initialization and proper weight updates are paramount. Although some initialization techniques were introduces in the above sections, the authors of [47] initiated the last layer of the networks with weights uniformly drawn from $[-3\times10^{-3}, 3\times10^{-3}]$ for low dimensional problems. Such an initialization ensures initial policy and value estimates were near zero [47]. The other layers were initialized using uniform distributions $[-\frac{1}{\sqrt{n_j}}, \frac{1}{\sqrt{n_j}}]$, where n_j was the size of the previous layer. Regularization-wise, batch normalization was used [46]. For exploration, the UO noise given in Equation 2.65 with θ and σ as 0.15 and 0.2 was used.

Experience Replay

DDPG also uses experience replay (sometimes called replay buffer) to enhance data efficiency and prevent catastrophic interference during training. Experience replay was first introduced in [51] to provide temporally de-correlated training samples to agents in time-series settings. In DDPG, tuples of:

$$(x_t, u_t, r_{t+1}, x_{t+1})$$

are memorized and stored in the experience replay. During updates, random mini-batches of previous experiences are sampled from the replay buffer to update the agent. Consequently, the agent obtains the ability to learn the same experiences many times, a concept similar to cycling through many epochs in deep learning. Correlating to humans, experience replay is similar to hippocampal replay, where memories are sub-consciously replayed over and over. Indeed, that is one theory explaining the efficiency of human learning [58]. However, human memories are rarely replayed randomly. Instead, only the most important or unexpected memories are replayed. Prioritized experience replay mimics this concept and biases sampling to experiences with large TD errors [52]. Intuitively, such experiences are *shocking* since the outcome was significantly different than what was expected. Using prioritized experience replay, the agent learned faster in 41 out of 49 ATARI games compared to the original experience replay.

2.7.5 Input and State Constraints

As with all RL methods, input constraints can be handled quite trivially; however, state constraints are much more difficult. Typically, *soft* state constraints are implemented in RL by introducing large negative rewards when the agent arrives at an undesired states. Indeed, humans learn state constraints in such a way where guardians provide negative consequences when we venture into troubling situations. In literature, constrained Markov decision processes (CMDPs) and safe RL are two

fields that explore how RL can handle constraints explicitly; however, most modern methods either require explicit system models or are difficult to implement in industry.

2.7.6 Training Algorithm

The DDPG algorithm is trained as follows [47]:

1. Initialize replay buffer and the actor and critic network weights corresponding to the previous subsection.
2. Observe some states from the system
3. Map the states to some exploratory actions via the online actor network:

$$\mu'_t = \mu(x_t|\theta^\mu) + \mathcal{N}_t$$

4. Implement μ_t to the system, observing transition to x_{t+1} and obtaining r_{t+1}
5. Store tuple $(x_t, \mu_t, r_{t+1}, x_{t+1})$ into the replay buffer
6. Sample a mini-batch of N experiences $(x_t, \mu_t, r_{t+1}, x_{t+1})$ from the replay buffer
7. Using the target critic network, compute $y_t = r_t + \gamma Q'(x_{t+1}, \mu'(x_{t+1}|\theta^{\mu'})|\theta^{Q'})$ for each experience
8. Update online critic parameters by minimizing: $J = \frac{1}{N}\sum(y_i - Q(x_t, u_t|\theta^\mu))^2$
9. Update online actor parameters by:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum \nabla_u Q(x, u|\theta^Q)|_{x=x_t, u=u_t} \nabla_{\theta^\mu}\mu(x|\theta^\mu)|_{x_t}$$

10. Update both actor and critic target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

Most steps above are intuitive to understand; however, steps 6 and 9 might be slightly confusing. In step 6, mini-batches of experiences are used for training to enhance data efficiency and to provide temporally de-correlated training data. For time-series problems, such as continuous control, a direct adaptive control method like RL will quickly adapt to the current operating condition and exhibit catastrophic interference on other operating conditions. By training on an uniformly sampled mini-batch of historical experiences, catastrophic interference can be largely avoided.

Step 9 shows the slow update of the actor and critic target networks. This follows the same intuition as DQN, where the target network is frozen for periods of time to prevent the moving target problem. Except in DDPG, the target networks are updated in small steps after each episode rather than being kept frozen, and then undergoing a complete update.

2.8 Model Predictive Control

Compared to all topics in process control, the concepts of model predictive control (MPC) is perhaps the closest resemblance to modern RL. MPC is a model-based control strategy (known as a planning method in RL literature) that optimizes the input trajectory of a system by using the functional equation (a function where the unknowns are also functions) generated from the system's state information together with a value function. The performance of MPCs heavily relies on the accuracy of system identification as the input trajectory is solved by extremizing an objective function using mathematical programming (MP) as a function of the process model [10]. The objective function is typically given as:

$$J = \sum_{i=1}^{N} x_i^T Q x_i + \sum_{i=1}^{N} u_i^T R u_i \tag{2.68}$$

where N, Q, and R are the prediction horizon and tuning matrices, respectively. Superscript T denotes the transpose operation. Q and R are diagonal matrices and are used to emphasize importance on different state and inputs, respectively. Here, x and u are given as:

$$x_{sp} - x_i \tag{2.69}$$

$$u_{ss} - u_i \tag{2.70}$$

where subscripts sp and ss denote the set-point and steady state, respectively. Often times, MPCs are applied onto integrating processes; thus, using Δu to handle such scenarios.

Implementation-wise, MPC uses a receding horizon approach where the controller predicts and optimizes for a set amount of steps into the future. However, only the first control action is implemented. During the next sampling time, the trajectory is re-optimized and the cycle repeats. The length of the input trajectory and the number of steps the controller predicts into the future are known as the control and prediction horizon, respectively. During design, it is paramount to ensure that both the prediction and control horizons are adequate in length to ensure optimal dynamic performance. Intuitively, the prediction and control horizon can be related to the everyday task of driving a car. It would be very dangerous if we only consider events one second into the future because it would be difficult to react to curves and other road side disturbances; therefore, the prediction and control horizons must be sufficiently long to ensure safe and optimal driving practices. Typically, the control horizon is chosen to be shorter than the prediction horizon due to computational cost and the unimportance of unnecessarily long input trajectories [59]. One flaw with the receding horizon approach is its extremely expensive online computational cost, especially in large non-linear systems.

Explicit MPC was developed to mitigate this computational burden by leveraging parametric programming to pre-compute solutions to the optimization problem offline [60]. During online evaluation, the controller simply looks up the optimal in-

put from a dictionary of pre-computed solutions, making online evaluation extremely fast. This idea is exactly equivalent to RL, where the agent is trained offline (i.e., solves the optimal policies offline), allowing extremely fast online evaluations.

Ultimately, MPCs provide many advantages compared to classical control strategies. For example, MPC considers long term planning and identifies the optimal input trajectory rather than the best immediate action. Furthermore, MPCs have predictive capabilities and can anticipate future events, allowing the controller to plan future control actions accordingly. A third advantage is that the MP methods used in MPC have been widely demonstrated to handle both input and state constraints relatively successfully. In modern times, MPCs are often implemented in the supervisory control layer.

The process control hierarchy is shown in Figure 2.11. Starting from the bottom, the *regulatory controllers* are typically used to ensure regulation of disturbances and set-point tracking of the process and directly actuate the process instrumentation. A common regulatory controller is the Proportional-Integral-Derivative controller (PID). The layers above are known as the *supervisory controllers*. MPC is a common supervisory controller and is classically implemented for regulation or set-point tracking problems exclusively. Economic objectives of the process were managed by the real time optimization (RTO) layer through steady state optimization [61]. More recently, control practitioners began to unify the ideas of RTO and MPC into a centralized algorithm called economic model predictive control (EMPC). Here, the economic objective of the RTO is placed into the objective function of the MPC, allowing for the input trajectory to optimize the economic objective instead [62], [63].

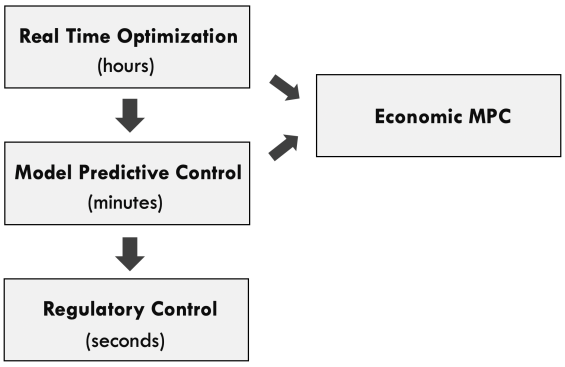


Figure 2.11: A typical industrial control architecture.

Comparatively, RL can be described as a general control algorithm and can be used to replace *any* layer in Figure 2.11. For example, a MPC or PID based RL would have its reward function to be identical as the negative of Equation 2.68. In the EMPC case, the reward function of RL would instead be the economic objective. In Chapter 4, the performance of RL based supervisory controls will be extensively compared to traditional methods on simple and complicated processes. Additionally, the pros and cons of each method will be summarized.

Chapter 3

Machine Learning for Prediction Applications

Cheap data storage and escalation of computational power allowed the world to enter a new age: the age of *big data*. With vast amounts of data, previously inviable and data hungry machine learning algorithms are now implementable. The technology sector was the first group to be able to exploit this arcane technology to create tremendous value in applications ranging from targeted advertisements to self driving cars. The value was so great that the current top four companies in America by market capitalization are all technology companies (Microsoft, Apple, Amazon, and Facebook) as of 2019. As the technology sector's successes grow, other industries begin to catch a glimpse of the potential value creation in their own respective industries and initiate their own digital revolution. The ripples of success from the technology industry ultimately resulted in waves of capital investments into machine learning (ML) and artificial intelligence (AI) from all industries.

ML solutions promise to be cheaper, more accurate, and have online learning abilities compared to traditional methods. Additionally, the solutions are promised to be easier to implement and will take less time to design; feed it data and it will learn, as they claim. With this mentality, machine learning engineers and data scientists from technology companies attempted to conquer other industries,

one industry being chemical process industry. Unfortunately, their crusade fell short and their successes were few due to their lack of engineering knowledge and inability to identify large value gains. Typically, projects in technology companies deal with very unambiguous information such as identifying location of objects or predicting the enjoyments of an individual based on previous articles they have read. However, the process industry typically generates time-series data and is often very ambiguous with data characteristics unique to the industry. Some characteristics include time delayed data, multi-modal data, unreliable data, highly noisy data, state transition dynamics data, and any combination of the prior. Due to the increased complexity, data pre-processing for ML projects in the process control industry is mission critical and much more vigorous for successful applications.

Table 3.1 shows some general machine learning applications for the process control industry. Currently, ML applications in the process industry can be broken down into prediction, monitoring, and control. The field of prediction deals with mapping from certain inputs to desired outputs. An example would be building a soft sensor to predict for a state, x_m, that is expensive to measure. By identifying states highly correlated to x_m, a multivariate soft sensor can be built to inexpensively predict the state in the future. In ML monitoring, the algorithms are tasked to monitor the process for anomalous activities. Here, an example would be applying a classification method to predict for failures in process equipment. Lastly, ML control is concerned with the topics of adaptive, multivariate optimal control. Reinforcement learning is the typical ML algorithm for control.

Table 3.1: General applications for machine learning in the process control industry.

Prediction	Monitoring	Control
Soft sensing	Anomaly detection	Supervisory control
Forecasting	Anomaly prediction	Regulatory control
Operator education	Alarm prioritization	Operator education
Process modelling	Alarm reduction	Multivariate control

Figure 3.1 shows a potential machine learning architecture that is generic enough for implementation in all industrial plants. First, the industrial process (e.g. refinery,

pipeline, reactor, etc.) sends raw sensor data into the cloud, where it is cleansed through data pre-processing methods. Then, the filtered data is sent into different machine learning algorithms depending on the objective of the application and will output the desired values. After a set time frame, all ML models will then be re-updated to learn the newest experiences.

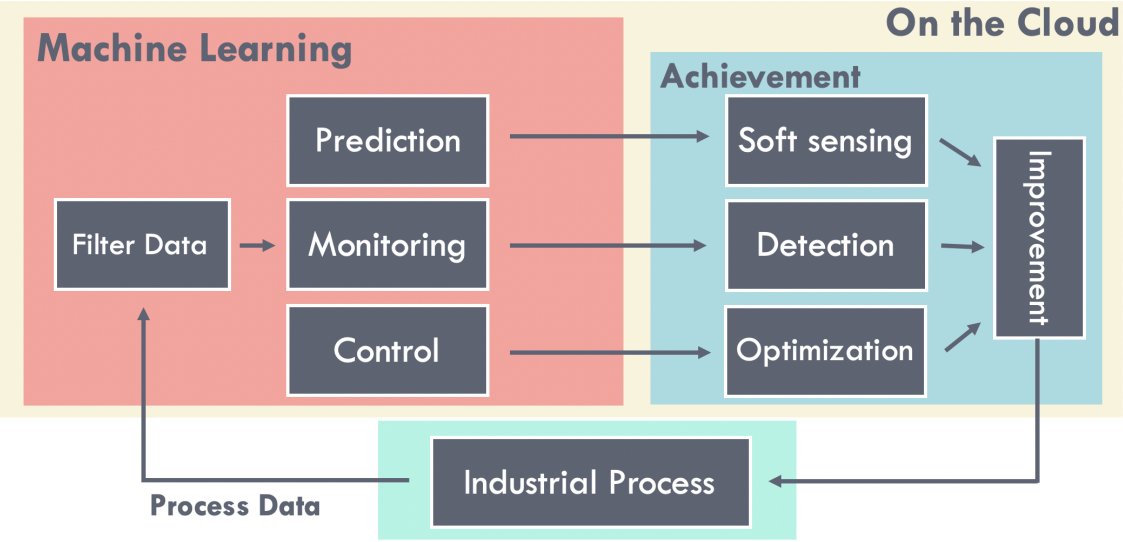


Figure 3.1: A potential machine learning architecture in an industrial environment

The objective of this chapter is to convey ideas for implementing machine learning solutions catered towards the **process control** industry. In this chapter, the first half consists of common data pre-processing techniques to handle common process control concerns and is visually described by the *Filter Data* box. The second half contains machine learning methods (in order of difficulty) to handle different process control prediction problems (*Prediction* box in the above figure). To conclude this chapter, the prediction algorithms will be closed off with an adaptive modelling technique inspired by reinforcement learning and adaptive resonance theory. For validation purposes, the machine learning methods were implemented onto an industrial pipeline for prediction, monitoring, and optimization[1, 2].

Contributions made in this chapter include:

1. Effective data pre-processing techniques for the process industry

[1]This project was supported in part by Mitacs through the Mitacs Accelerate program.
[2]This chapter only contains the theory and application highlights. The detailed industrial project report can be found in Appendix A.

2. Catering machine learning prediction techniques to the process industry

3. An outlier-free, data efficient, adaptive modelling method for multi-modal operations

3.1 Data Pre-processing

Data pre-processing typically includes many steps starting with filtering by subject matter expertise, and then transitioning to common statistical methods. For this section, only the filtering methods unique to process control will be discussed. Please refer to Appendix A for details regarding the other steps. Process control is typically concerned with multivariate time-series data plagued with noisy and/or unreliable sensor readings. Time delays are critical to successful prediction applications in process control. Furthermore, some processes may also have a variety of different operating regimes depending on downstream demand or ambient conditions. In order to have successful prediction algorithms for process control, all of the above must be considered.

3.1.1 Time Delay Data

Time delay is the time between the performance of a control action and the change in output. Time delays occur due to the physics of the natural world. For example, turning on a pump at the beginning of a pipeline does not result in higher flow rates immediately. The process takes time to adjust and **transition** to the new steady state; therefore, raw data must be first shifted to account for the time delay. Without doing so, models would be using current information to predict the past. Imagine building a model to predict for the outlet flow rate of a pipeline where the regressors are pump statuses 300 km upstream of the outlet. If a change in pump status occurred at $t = 0$, the pressure will take a few minutes to propagate down the pipeline. Thus, the model taking pump statuses at $t = 0$ must have its flow rate labels shifted from $t = \tau$, where τ is the expected time delay. An example of the

time delay shifting procedure for an industrial pipeline is shown in Table A.4 located in Appendix A, where data was shifted for different locations along the pipeline to enhance predictive capabilities.

Initial engineering expertise and/or data analysis must be conducted to identify the time delay for specific processes. For example, it is well known that pressure propagates down incompressible fluids at approximately the speed of sound (1480 km/h) [64]. Using this information, adjusting for the time delays along the pipeline was made trivial.

3.1.2 Multi-modal Data

In the process industry, it is common to have multiple modes of operation due to changing ambient conditions (e.g. summer, winter), different market demands, and a variety of other factors. Each operating condition also consists of unique equipment operation and process characteristics (flow rates, temperatures, etc.); therefore, a common model to predict for many different operating conditions lead to increased model errors. Here, unsupervised learning should be used to avoid this scenario for systems with many modes. More specifically, clustering methods should be applied to segregate data from different operating modes, and separate models should be built using data from each operating mode to enhance accuracy. For big data applications, k-means or density based scanning (DBSCAN) should be used due to their scalability and non-iterative nature [65]. Of the two methods, k-means is much faster while DBSCAN is more robust to outliers.

An example of the breakdown of a multi-modal system can be found in Figure A.11 in Appendix A. By segregating the system into multiple modes, more accurate weights can be identified for each mode compared to general weights for all modes. In fact, most modes would not even use the same equipment. Such a concept is similar to using a linear parameter-varying model to approximate a non-linear system.

3.1.3 Unreliable and Noisy Data

Thousands of measured data are recorded per minute on modern distributed control systems. However, many process variables such as viscosity, or parts per million (ppm) are difficult to measure with modern equipment on a live process. This results in inaccurate values being sent to the ML models, ultimately reducing accuracy. To overcome highly unreliable data, a general strategy is to identify how the operator(s) are using the data and to engineer the feature(s) to be used in the same way for the ML model. For example, the densitometers installed along the industrial pipeline shown in Appendix A all show different readings for the same crude. At times, the reading could be off by ±20%. However, the operators only use the density reading to determine the grade of crude inside the batching pipeline. The crude is light if the API is above a threshold, heavy otherwise. The physical number had no meaning for them. To improve the ML model accuracy, the density reading was feature engineered to be a binary variable reading *1* if the API was above the threshold, *0* otherwise. By doing so, the variable in the ML model was used in the same way as the operators and the accuracy increased.

Other data measurements may be accurate, but highly noisy. Noisy measurements may lead to significant predictive errors and should be minimized for proper predictions. One such method to reduce noise significantly is to apply an exponentially weighted moving average (EWMA) filter given by:

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t,\ v_0 = 0 \tag{3.1}$$

$$v_t \leftarrow \frac{v_t}{1-\beta^t}, \forall v \in V \tag{3.2}$$

where v_t is the exponentially weighted value at time t. β is the exponentially weighing factor. Larger β results in smoother results. θ_t is the original value at time t. V is a vector representing the exponentially weighted values before bias correction. EWMA is a very effective way to remove noise in chemical processes because these processes typically contain slow dynamics. By exponentially smoothing the data,

the fast peaks are removed while preserving the slow dynamics. An example of the EWMA algorithm applied to the measurement of drag reducing agent (DRA) ppm values is shown in Figure A.7 in Appendix A. DRA ppm measurements are known to be highly noisy; however, the noise can be almost completely removed by applying EWMA.

3.1.4 State Transition Dynamics

Another unique topic of process control is the dynamics of the system. System dynamics refer to the transitional period of going from one steady state to another after a control input is provided. Typically, dynamical models are used for advanced process controls where optimizing for the dynamics of the system is critical for optimal performance. In order to build machine learning models to describe the dynamics of systems, a time-series implementation must be pursued. Typical ML models map states and control actions at time t to the desired output at time t_{ss}, where t_{ss} is the time required for the system to transition to the new steady state. By doing so, the dynamics of the system are completely omitted. In order to build a dynamical ML model, the raw data needs to skip the time delay pre-processing step and be augmented by time. Imagine a simple single-input single-output (SISO) system:

$$y = w_1 x + b \tag{3.3}$$

In time-series implementation, the model would instead be:

$$y_{t+1} = w_1 x_t + w_2 x_{t-1} + w_3 x_{t-2} + ... + y_t + y_{t-1} + ... + b \tag{3.4}$$

where the input vector would be augmented as $\mathcal{X} = [x_t|x_{t-1}|x_{t-2}|...]$. Here, Equation 3.4 becomes the 1-step ahead predictor of the system and dynamics can be predicted for. This data augmentation method is identical to all ML models if a time-series implementation is desired. In Appendix A, Figure A.20 shows an example of a time-series prediction model. Because such models only predict one step in advance, error

is typically very low. The predicted value can also be fed in recursively to generate a infinite step ahead prediction that can be used for forecasting long term trends.

3.2 Machine Learning Methods

Many ML methods exist for prediction, each having its advantages and disadvantages. In this section, the most common ML methods will be shown along with their applications in process control. Unique hyper parameters for different ML methods will also be shown; however, common hyper parameters such as α, training epoch, and mini-batch size are common throughout and will be omitted. Furthermore, common model performance metrics will be introduced.

Performance Assessment

The model performance were assessed using the following three ways:

1. Root mean squared error (RMSE) [32]:

$$J = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{3.5}$$

2. Mean absolute error (MAE) [32]:

$$J = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{3.6}$$

3. Coefficient of determination (R^2) [32]:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y_i})^2} \tag{3.7}$$

Table 3.2 shows the advantages and disadvantages of each assessment metric.

Table 3.2: Pros and cons of different model performance assessment methods.

Method	Advantages	Disadvantages
RMSE	Useful for identifying large errors	Smaller errors are muted
MAE	Easy to interpret as all errors have the same weight	Inferior to RMSE when large errors are undesirable
R^2	Easy to understand, $-\infty \leq R^2 \leq 1$	Valid only for linear relationships

3.2.1 Linear Models

Linear models have two variants, linear regression and logistic regression. The former is used for prediction tasks associated with continuous variables while the latter is used for classification tasks. For example, linear regression is a great algorithm for soft sensor applications whereas logistic regression is more suitable for monitoring for anomalous activities. In this chapter, only the prediction variant will be shown. The model structure of linear regression is given as:

$$\hat{y} = W_1^T x + W_2^T u + b \tag{3.8}$$

where $x \in R^n$ is a vector of states, $u \in R^m$ is a vector of inputs and superscript T denotes the transpose operation. $\hat{y}$ is the predicted variable and can be anything; in soft sensors, $\hat{y}$ would be the *soft sensed* variable.

The most common model structure for ML in the process control industry are linear models despite all processes being non-linear. This is because the narrow region most processes operate around can typically be assumed to be linear [57]. Additionally, linear models are simple, interpretable, and require low amounts of data. However, the draw backs of linear models are their poor performance in the big data era where large amounts of data is available (see Figure 3.2). This trait is intensified given high dimensional data sets where identifying interaction effects are critical for accurate predictions.

Linear models were applied to the industrial pipeline as a benchmark algorithm. The performance of the linear models can be seen in Tables A.9, A.10 and Figures A.13, and A.14.

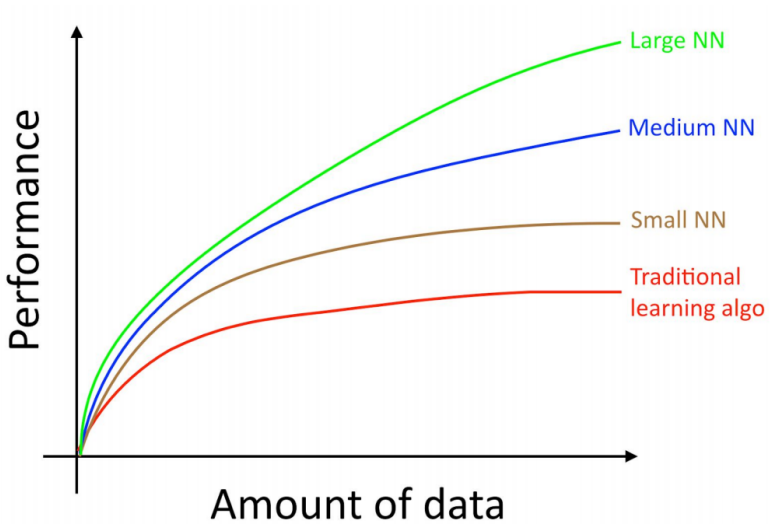


Figure 3.2: Performance as a function of data. Original image from [39].

3.2.2 Polynomial Models

Polynomial models are a general class of non-linear models that explores the main and interaction effects of its regressors. The general model structure of a two regressor polynomial model is given by:

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + w_5 x_1 x_2 + e \tag{3.9}$$

where w are the weights, x are the regressors, and e is the modelling error. In this model, linear, quadratic, and interaction effects are all explored simultaneously. However, the amount of parameterization required for a high dimensional prediction problem using this model structure might be difficult to interpret, thus, a truncated version of the model could be used instead for high dimensional problems. One special case of a truncated polynomial model is the exponential model given by:

$$\hat{y} = w_1 x_1^{w_2} + b \tag{3.10}$$

where the power of the regressor is also a weight to be identified. This is advantageous in situations where the non-linearity of the system is unknown. A quadratic and square root version of the exponential model were applied to the pipeline and the performances are shown in Table A.12 and Figure A.18. Compared to the linear

models, the errors were reduced by up to 10%.

3.2.3 Neural Network and Deep Learning Approaches

Neural network and deep learning approaches shine for predictive tasks where predictive power is the primary driver, while interpretability is not an issue, and acceptable[3] hardware exists. Deep learning is a special case of neural networks where many hidden layers exist. The general consensus of the AI and ML community is that any neural network with more than three hidden layer is considered deep learning; however, the idea is not concrete and is open to personal preference. The neural network model structure is highly non-linear and attempts to explore interaction effects of all regressors. For a more detailed explanation on neural networks, the notation of its the variables, and its theory, please refer back to Chapter 1. Only a brief summary of the theory will be provided here. Due to the model complexity and high parameterization of neural networks, its predictive powers are unparalleled compared to other methods and can fit any function. In [66], the authors showed that:

> There exists a two-layered neural network with ReLU activation functions and $2n+d$ weights that can represent any function on a sample of size n in d dimensions.

Three distinct types of neural networks exist: i) Multilayer perceptrons (MLPs); ii) Recurrent neural networks (RNNs); iii) Convolutional neural networks (CNNs).

A visual representation of a MLP is shown in Figure 3.3. MLPs (also known as feedforward neural networks) are the simplest and most common of the three. In MLPs, the outputs of each neuron, $a_j^{[r]}$, is computed as:

$$a_j^{[r]} = f(w_1 x_1 + w_2 x_2 + ... + w_m x_m + b) \tag{3.11}$$

[3]Neural network models are typically executed on servers and the outputs are sent to the actuators. For fast processes, deep learning models should be pushed to the edge device.

where the function, f, is non-linear and known as the *activation function*. The purpose of f is to introduce non-linearity to the model; a critical addition because no process in the real world is linear. In an intuitive sense, MLPs can be visualized as a brute force approach to identify the interaction effects of every regressor with each other. Due to the sheer number of parameterization, MLPs are very effective in predicting in-sample data points. However, the models suffer tremendously during events where the testing data is significant different. Large MLPs also tend to overfit; thus, it is critical to increase regularization as the MLP increases in size.

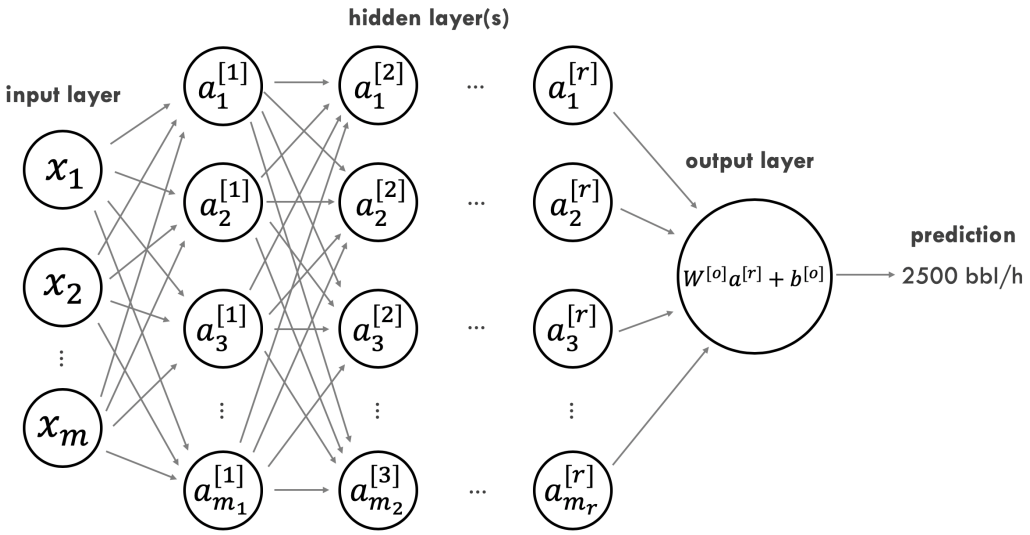


Figure 3.3: Structure of a general neural network.

MLPs were also applied to the industrial pipeline to model for the outlet flow rate. The input variables were measurements of variables along the pipeline such as temperatures and pressure, and the output was the outlet flow rate. Three different MLPs with varying sizes were applied. Their respective performances can be seen in Table A.14 and Figure A.18. It can be seen that the performance on the training and validation data were both excellent, though the error on the testing data increased significantly. This was caused by the testing data being significantly different from the training data. Data for model training was collected during the winter months, but was tested on summer months where the temperatures increased by up to 10° C. The increase in temperature resulted in reduced viscosity of the shipped crude and significantly hindered the predictive power of the MLPs. Ultimately, the MLPs'

performance on the test data was almost identical to the much simpler polynomial models and was not used.

An especially useful type of neural networks for the process industry are RNNs (Figure 3.4) due to their time-series architecture. Naturally, RNNs are set up to be infinite step ahead predictors and identifies temporal correlations within the data. Traditional applications of RNNs can be found in speech recognition, translation, and language modelling. In the process control industry where time-series data is abundant, RNN is the natural choice for typical soft sensing applications. On a high level, RNNs accept inputs x_t, and outputs y_t. At the same time, y_t is sent as an input, along with x_{t+1}, back into the RNN to compute for y_{t+1}. A similar computation is conducted until the end of the sequence of inputs. By recursively re-inputting outputs as input data, RNNs are able to predict for an output trajectory given an input trajectory.

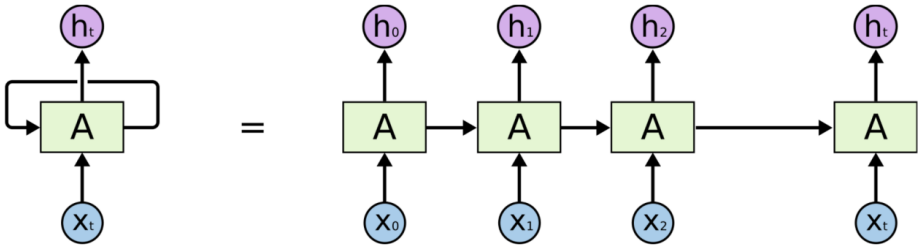


Figure 3.4: Architecture of a RNN. Original image from [32].

CNNs are the last type of neural networks and are typically used for computer vision applications. The architecture of a typical CNN can be found in Figure 3.5. Unlike its predecessors, CNNs make the explicit assumption that inputs to the network will be images. This enables certain properties to be encoded into the architecture, making the forward pass more efficient while reducing the number of parameters. More specifically, CNNs assume all inputs are arranged in three dimensions: height, width, and depth. The height and width are simply the resolution of an image while the depth is the amount of color channels. For example, a coloured

image contains 3 channels (red, green, blue) while a grayscale image contains only one. From this assumption, the weights of CNNs only need to be applied to specific locations, without the need of fully connected layers. Furthermore, the input data will then be downsampled using pooling layers to extract the most important features while discarding the rest.

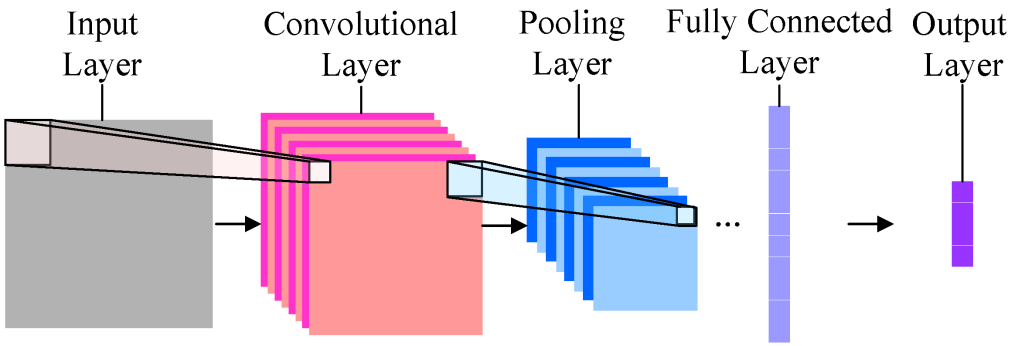


Figure 3.5: Architecture of a CNN. Original image from [32].

In the process industry, CNNs can be leveraged as a soft sensor to measure variables using cameras. An example would be detecting the level of crude inside a primary separation vessel using the sight glasses, since traditional methods are not as effective.

Both CNNs and RNNs were not applied to the industrial project directly, but were provided to the reader as **advanced** methods for future projects that command exceptionally high predictive power, or computer vision soft sensing capabilities. More information regarding RNNs and CNNs can be found in [32].

3.2.4 Linear Parameter-varying Models

Linear parameter-varying models (LPV) were last type of ML models that were applied to the industrial pipeline. The motivation behind LPV models is twofold: i) Achieves a non-linear representation of the data using a combination of linear models; ii) Models different operating regimes of a process using different models. The LPV model structure is identical to linear models with the exception that there are multiple linear models. The number of linear models is a function of the non-linearity of the system, the number of operating regimes, and the amount of available

data.

Figure 3.6 shows an example of fitting multiple linear models to approximate a non-linear system. Two separate approaches were used: 3-model approach and 6-model approach. Performance wise, the 6-model approach is far superior. However, the 6-model approach uses twice as many models resulting in a significantly higher maintenance and ownership cost. Additionally, the individual model performance for the 6-model approach may experience high variance if low amounts of data are present at certain points. If performance is of utmost importance and data is abundant, a large number of linear models could be used. Otherwise, a LPV model with a lower amount of models is adequate.

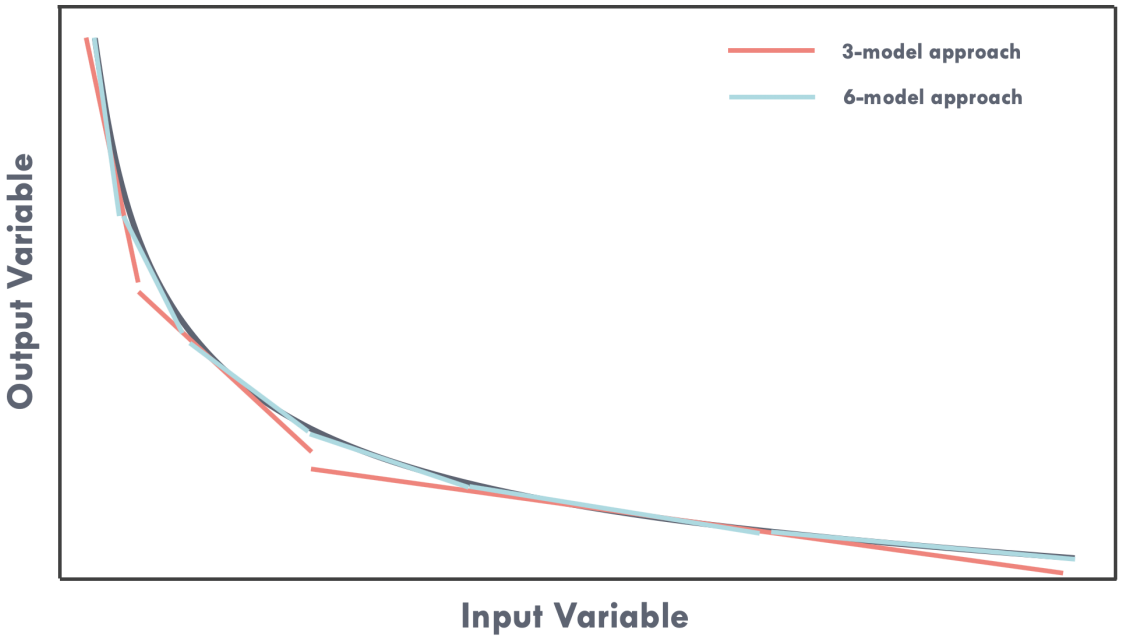


Figure 3.6: Fitting a non-linear function using multiple linear models.

Clustering techniques can be used to identify distinct models for large MIMO systems containing multiple operating regimes and/or where non-linearity is not easily visualized. Table A.15 and Figure A.19 shows the application of a 2-model LPV model onto the industrial pipeline for the two separate operating regimes. The original data set was first segregated into two clusters using DBSCAN. Then, separate linear models were built for each cluster. During model testing, the Euclidean distance of new samples to the centroids of each cluster is computed to determine

which linear model should be used for prediction. In terms of performance, the LPV model was able to achieve very similar performance metrics compared to the other non-linear models. Additionally, the LPV models are more representative of the process in terms of control because each model (for each operating regime) has unique weights and constraints. For control, this is especially beneficial. For example, if pump A was never used for operating regime B, the model would be highly inaccurate if the control system recommended its operation. Using LPV models, explicit constraints can be placed to prevent such a scenario, as shown in the LPV section in Appendix A.

The training and deployment procedure for using a LPV model in an arbitrary process is shown in Figure 3.7. Starting from the top, historical data for the process is first clustered into n data sets. Here, n can be pre-defined using subject matter knowledge, or can be found using DBSCAN. After segregation, each cluster should have enough data to effectively identify useful linear models. Finally, the data sets are used to identify linear models. Each model will have unique constraints to enhance the representation of the physical process. Deployment-wise, new measurements are obtained from field sensors and are sent to the LPV model. Then, the Euclidean distance (or desired distance metric) between the new measurement and the centroids of each model are calculated. The model exhibiting the lowest distance will be used for prediction. However, if the distance between the new measurement and the closest centroid is too large, the measurement will be labeled as anomalous instead.

3.2.5 USIS: Uniform Sampling Incremental Supervised learning

A major selling point of ML solutions is their promise of being adaptive. Because of the way ML solutions are updated (gradient descent), adaptation can easily be incorporated in a natural way. Figure 3.8 shows an intuitive representation of the gradient descent algorithm.

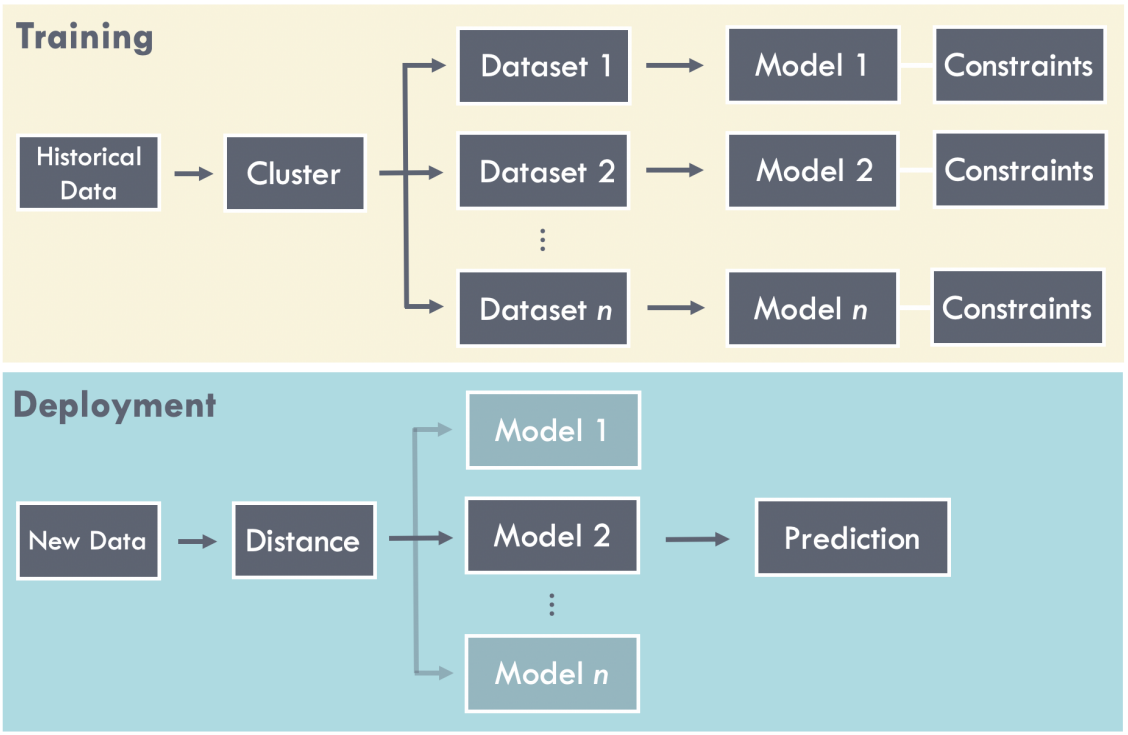


Figure 3.7: Architecture of the LPV model during training and implementation

During an update step, the new knowledge is intuitively the old knowledge adjusted by new learnings. The new learnings are typically multiplied by a fixed learning rate, α. From this, the new knowledge is always biased towards the most recent experience, giving ML solutions an adaptive characteristic. Adaptive ML can be implemented in two forms: online learning or incremental learning. Online learning refers to the ML models being updated after each prediction and can be understood mathematically as stochastic optimization. Common applications of online learning can be found in search optimization for web pages, where millions of data points are generated per minute. Online learning is not suitable for process control applications due to three reasons: catastrophic interference (i.e., tendency of neural networks to completely and abruptly forget previously learned information upon learning new information [67]), sequential noisy data decimating model accuracy, and insufficient incoming data to identify representative models for useful applications. In adaptive ML, incremental learning is the preferred choice for process control applications due to its reduced randomness. Incremental ML works by creating a data cache, and then updating the model using all data from the

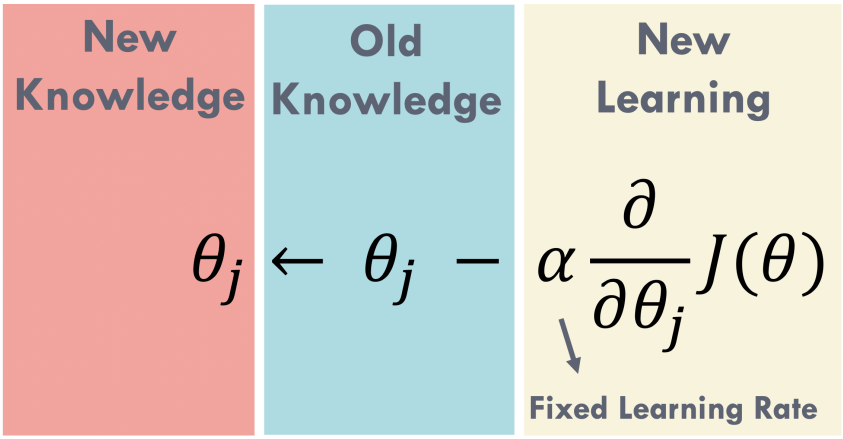


Figure 3.8: An intuitive representation of gradient descent.

cache simultaneously after a fixed interval. By doing so, the models will be updated much slower and the gradient of the loss function is an average of many examples, a method similar to semi-stochastic optimization.

Motivation

Incremental learning still falls short in terms of catastrophic interference (also known as catastrophic forgetting). This is especially a problem for the process industry, where operating conditions may be prolonged for many months before a switch is made. Because ML solutions are biased towards the most recent experiences, past experiences will be forgotten and the model will perform poorly if faced with old conditions after a long time.

The uniform sampling incremental supervised learning (USIS) algorithm was proposed to overcome this issue and to enhance data efficiency for adaptive ML techniques. USIS is a combination of adaptive resonance theory, uniform sampling, and experience replay (from reinforcement learning theory), where each update step is outlier free, data is efficiently used many times, and data is uniformly sampled across the distribution of the model to prevent catastrophic interference.

The simplified adaptive resonance theory (ART) architecture is shown in Figure 3.9. ART was originally proposed to prevent disrupting existing knowledge during learning of new knowledge. In ART, the comparison field first allocates the input vector to the best match model in the recognition field based on a similarity metric,

s. Then, the similarity between the input vector and the closest matching model is compared to the vigilance parameter, ν. If $s > \nu$, the model weights will be adjusted using the new input vector. Otherwise, the input vector is used to initiate a new model. The vigilance parameter has significant influence on the overall system. Intuitively, higher vigilance produces highly detailed memories (many specialized models), while lower vigilance creates generalized memories (fewer, generalized models) [68].

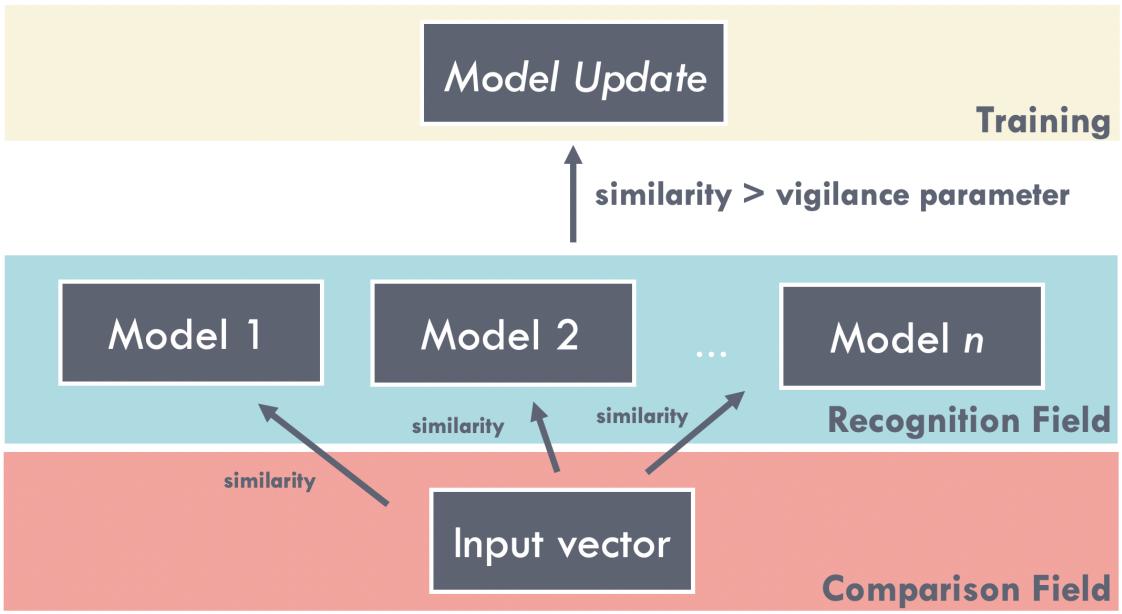


Figure 3.9: The simplified adaptive resonance theory architecture.

Uniform sampling is a random sampling method where proportions of each desired group are forced to be sampled equally. For example, imagine a pump data set used to predict for the pump RPM given an input current. The pump RPMs range between 0 - 1000 and the accuracy of the model is important across all values; however, the data set is significantly biased towards the 900 - 1000 RPM range, with only few data points in other regions. To ensure the model has acceptable performance across all values, the data set can be binned and equal amounts of data from each bin are sampled during the update step. In the context of process control where data typically remain constant over long periods of time, uniform sampling can guarantee data variety during an incremental learning update step.

Lastly, experience replay is a method that gained popularity when it was first

introduced to reinforcement learning to improve data efficiency and break temporal correlations [69]. In experience replay, data (or experiences) are accumulated over time and are stored in a buffer. During an update step, data is randomly sampled from the buffer to break temporal correlations that could potentially bias the models towards the most recent experience. Additionally, data within the buffer can be re-used many times rather than being discarded after one update. Only after many time steps are the oldest memories removed from the buffer. An biological interpretation behind the algorithm is that humans create memories of past experiences. Over the span of our lives, the same experiences are replayed many times; subconsciously when we eat, study, sleep, etc. In doing so, humans can learn new experiences without catastrophically forgetting about past experiences; however, after an elongated period of time, distance unimpactful memories are forgotten.

By combining the advantages of the previous three topics, a new adaptive, outlier-robust, and data efficient algorithm catered towards the process control industry was developed.

USIS Algorithm

Figure 3.10 shows the key steps for the USIS algorithm. Initially, $n \geq 1$ models are present to model the system. For input vectors in $n \geq 2$ systems, the distance d between the input vector and the centroid of the data for each model is computed. This step is skipped for systems where $n = 1$ because only one d is computed. The lowest d is then compared to the neglect parameter, η (opposite of vigilance parameter, ν). If $d \leq \eta$, the input vector will be added to the training archive (experience replay buffer) corresponding to the model. Otherwise, the data point is transferred to the new model archive where data currently not belonging to any model is stored. In the new model archive, new models will be generated and added to the existing system of models after enough similar data is accumulated. Likewise, the training archives are used to incrementally update all models after a certain amount of data or time elapsed. During an update step, the archive data

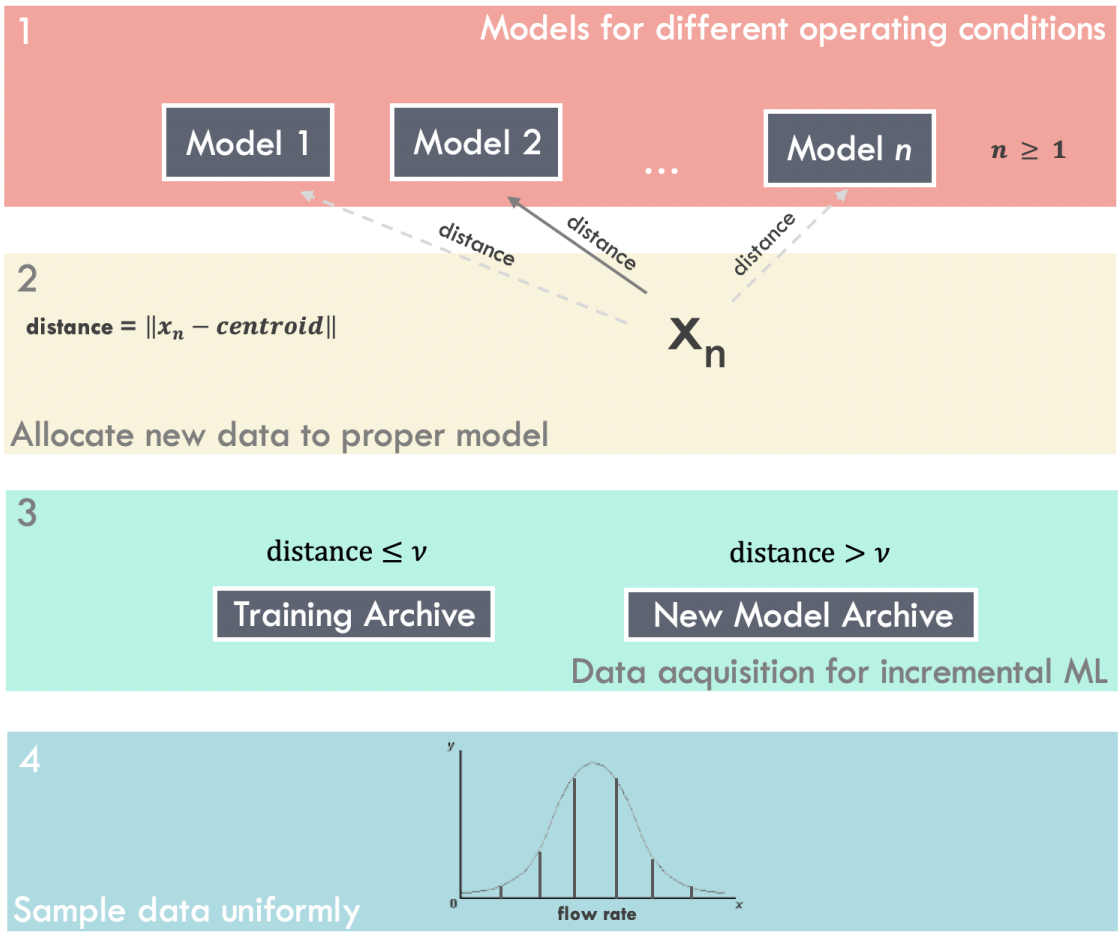


Figure 3.10: A brief visualization of the USIS algorithm.

will be blended with additional uniformly sampled data from the historical archive (experience replay buffer) to avoid catastrophic interference.

To enhance clarity of each update step, the data storage structure of USIS is shown in Figure 3.11. Before an update, the bins of each model must be defined. The bins are typically defined by dividing the distribution of the predicted variable. Bins can be narrower around regimes with abundant data, and wider for regimes lacking data. During an update step, all data from the training archive alongside some sampled data from the historical training archive are blended together and fed into the gradient descent step for model updates. Typically in process control applications, systems tend to linger around certain set points for many time steps. Given this characteristic, the data in the training archives, consisting of m training examples, typically belong to the same bin. To avoid catastrophic interference when learning on elongated periods of similar operating conditions, m examples of data

from each bin in the historical archive is also sampled and assimilated with the training data when performing update steps. After the updates, all training archives are emptied by transferring the data into the historical archives. When the historical archives get sufficiently full, the oldest data (or memories) are deleted.

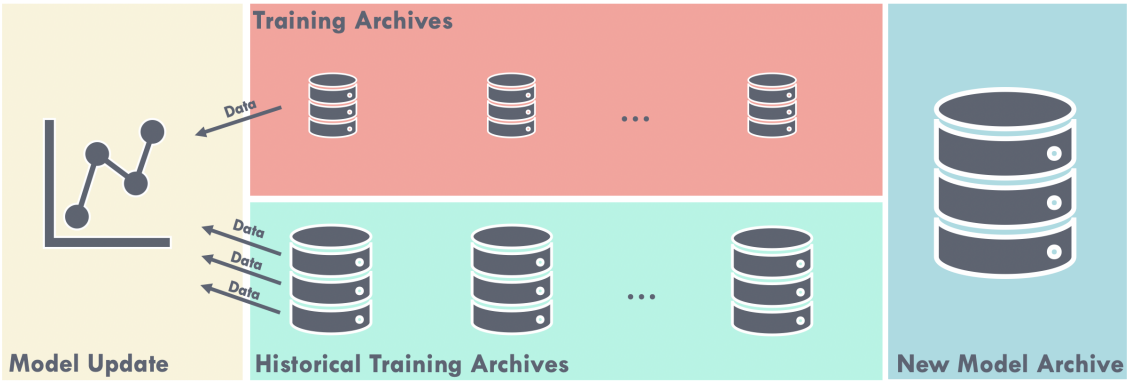


Figure 3.11: Data storage structure of USIS.

The full USIS algorithm is shown in Figure 3.12. There are five main tuning parameters of the USIS algorithm: i) when to update models; ii) bin size for each model; iii) neglect parameter; iv) distance metric; v) replay buffer size.

- **i) Model Updates:** Common strategies for model updates are periodically, by example size, or when models exhibit large errors. The first two are proactive methods, while the last is a reactive method. In the periodically updating method, the models are automatically updated after a certain time has elapsed (e.g., update every 24 hours). Updates by example size refers to model updates after a certain amount of data is accumulated in the training archive. Only the model with adequate amounts of data is updated. For these two methods, there should not be a significant time gap between updates because updates may lead to excessive model changes otherwise. Since this method is automated, significant model changes could potentially lead to sudden performance changes, jeopardizing production safety. Model updating after large errors have incurred is the last method. Here, the models are updated after an error threshold has been exceeded. Compared to the previous methods where smooth update steps are performed, this method introduces significant

changes to the model. Systems with highly noisy data and require inordinate amounts of manual data processing will benefit from this update style because completely automated updates may lead to model divergence.

- **ii) Bin size:** The bin size of each model should be tuned so the important sections of the predicted variable can be properly predicted for each model. Additionally, the bin can be narrower for sections where abundant data is available and wider for sections where data is sparse.

- **iii) Neglect parameter:** Neglect determines the specialization of each model. High neglect creates fewer generalized models, while low neglect produces many specialized models. For highly non-linear multi-modal systems, low neglect may be the preferred choice; however, the initial setup and cost of ownership for such a system could potentially be high if automated updates are not implemented/feasible. On the other hand, high neglect systems are cheaper to maintain due to the reduced number of models, but the accuracy may suffer. Picking a proper neglect parameter represents the trade-off between accuracy and complexity. A general rule of thumb for the neglect parameter would be the radius of the cluster. For example, η for a cluster identified using DBSCAN would be its ϵ value.

- **iv) Distance metric:** The Euclidean distance between the input vector and the historical data is the recommended choice; however, other distance metrics can be used for specialized applications.

- **v) Replay buffer size:** The replay buffer size can also be tuned to adjust when old memories are no longer relevant (intuitively, forgetting older memories). For processes where operating conditions are changing frequently, it would be beneficial to have smaller replay buffers to avoid learning on obsolete data. Replay buffers can also have an adaptive size depending on the current prediction error as proposed in [70]; however, such a method is still embryonic and further exploration should be conducted before implementation.

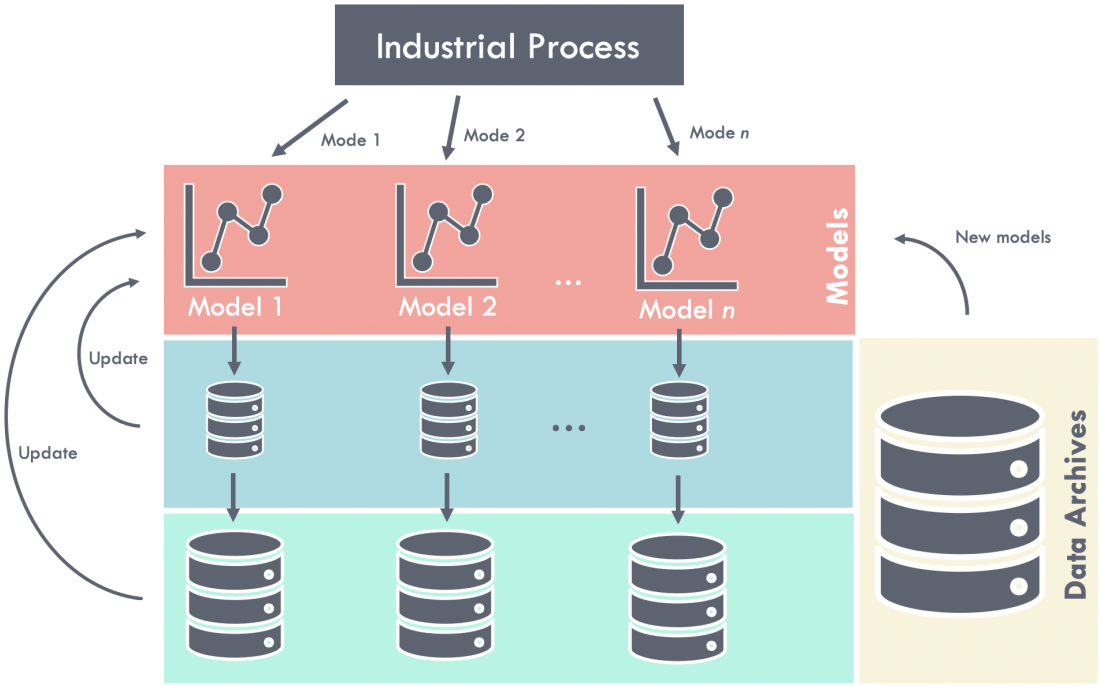


Figure 3.12: The complete USIS architecture.

Uniform sampling in USIS can also be swapped out for importance sampling, where high error examples are prioritized for the next sampling cycle (motivation from [71]. In doing so, high error examples (biologically modelled as *shocking* experiences) have higher probabilities of being recalled. Intuitively, humans recall *shocking* experiences more often. The method enhances accuracy in many reinforcement learning applications [69], [72], [73], although, such a method may prioritize noisy, near-outlier data points in process control applications and introduce model divergence.

3.3 Discussion: Cheaper, More Accurate, and Adaptive?

Machine learning has promised cheaper, more accurate, and adaptive solutions for industries around the world. However, applying ML solutions naively into the process control industry yields no significant value due to various unique factors in process control data. So far in this chapter, different data pre-processing methods

catered towards the process industry were introduced. General ML methods and their applicability were correlated to the process control industry. By innovatively combining parts of separate algorithms, a new algorithm catered towards the process industry, USIS, was also introduced. USIS is an adaptive, outlier-free, model structure that can be used to model a process with many operating conditions while avoiding catastrophic interference. But are these solutions truly cheaper, more accurate, and adaptive?

The comparison between ML prediction solutions and traditional methods is shown below for common areas where ML is implemented today:

- **Cheaper:** ML soft sensors will definitely be cheaper for difficult-to-measure process variables where lab test must be conducted. Conducting lab measurements may cost up to hundreds of thousands per year if frequent measurements are required. Initial cost for soft sensors may be high because a ML expert must collaborate with process operators to first develop the soft sensor(s); however, on-going costs for this application is minuscule. Soft sensors for prediction of variables that are difficult to measure are definitely cheaper than buying a highly advanced physical sensor. Lastly, prediction applications used for operator training or forecasting should be significantly cheaper compared to having a senior operator on shift explicitly to train a new operator or hiring a group of subject matter experts to forecast future production possibilities.

- **More Accurate:** Soft sensors used to predict lab measurements could yield high accuracy, but will never exceed lab measurements. This is because lab data is used to train the soft sensor; hence, serving as a performance ceiling. For soft sensors used to predict for difficult to measure process variables, the accuracy will depend on the quality of the data used to build the soft sensor. Additionally, the accuracy vary from application to application. For predicting for heights in a primary separation vessel using cameras and the sight glass, the accuracy can be extremely high. On the other hand, predicting the density of a crude given highly noisy readings and poor input data may be nearly

impossible.

- **Adaptive:** The adaptability factor of ML algorithms is purely dependent on available feedback. For applications where immediate feedback is available, such as predicting for a continuously measured output variable, adaptability is trivial. However, ML solutions cannot adapt for tasks where no feedback exists. One example of a non-adaptive ML application would be soft sensors for prediction of lab measured process variables. If no additional lab measurements are taken after the soft sensor is live, it will never adapt. Therefore, it is good practice to continue obtaining lab measurements to evaluate the soft sensor performance. For tasks requiring adaptation, ML solutions should be the preferred choice because human operators would have a hard time remembering all the historical data.

To summarize, ML prediction applications are generally cheaper compared to their traditional counterparts. This is simply because applications that requires ML are typically expensive and are solved poorly using traditional approaches to begin with. In terms of accuracy, ML solutions would not exceed the performance of a lab tested process variable; however, may surpass accuracy given representative data sets or unique applications. Lastly, ML solutions can only adapt if proper feedback is available for the algorithm.

3.4 Highlights of ML Application onto a Pipeline

The methods and algorithms in this chapter were applied onto an industrial pipeline[4, 5]. The highlights of the implementation will be shown in this section. For a detailed project report, please refer to Appendix A.

[4]Many values are significantly rounded and/or omitted to maintain confidentiality of the project

[5]The algorithms are live as of May 7^{th}, 2019 on one industrial pipeline. Work is being done to clone the project onto a second line.

3.4.1 Process Description

Pipelines are critical assets for the transport of fluids safely and efficiently across long distances. Fluids include petroleum products, clean water, natural gas, sewage, and even beer. In Canada, over 97% of petroleum products are shipped by pipeline alone. Due to the mission critical nature of pipelines on society's success, ensuring its reliability and efficiency has global-scale impact. Typical pipelines may contain hundreds of digital sensors littered across the pipeline. Such information overload may be overwhelming for humans; however, ML methods benefit greatly from the abundance of data. Here, an opportunity was discovered where ML methods can be applied to greatly increase the robustness and efficiency of the pipeline. For phase one of the project, ML prediction models were built to create a digital twin of the pipeline for operator training programs and future optimization purposes.

A schematic of the pipeline is shown in Figure 3.13. The highly complex pipeline spans over 100 kms and carries two products, a lighter product and a heavier product. The two products are batched (i.e., rotate between sending each product) and each product is sent for approximately eight hours before switching to the other product. The American Petroleum Institute (API) gravity for the lighter and heavier products are roughly 40 and 20, respectively. For the rest of this chapter, the lighter and heavier product will be referred to as *light crude* and *heavy crude*, respectively. The pipeline is typically operated between 1800 bbl/h to 3050 bbl/h.

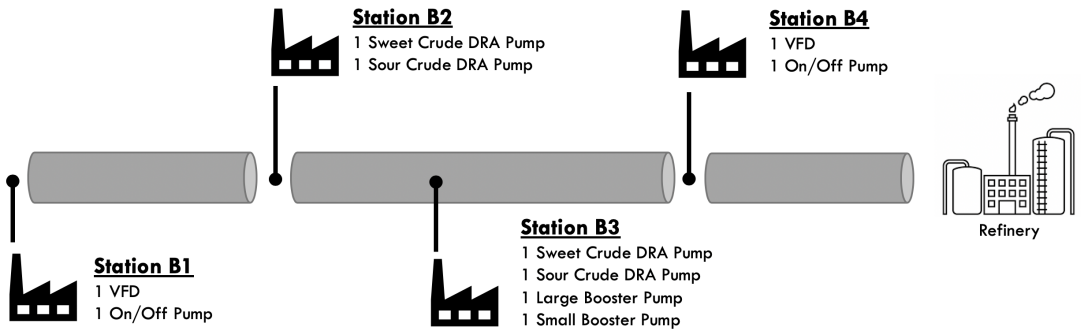


Figure 3.13: Schematic diagram of Line B.

Equipment wise, Line B boasts eight pumps spread across four pump stations. Two pumps are variable frequency drives (VFD), while the rest are on/off pumps.

Additionally, there are four drag reducing agent (DRA) injection pumps situated at the second and third pump stations. Each pump station contains a heavy crude and light crude DRA pump because the different crudes use different types of DRA. The DRA is injected based on the product present at the pump station.

3.4.2 Data Pre-processing

The industry sponsors initially provided a data set containing 899 tags and 525,601 data points for each tag. The data pre-processing step was deconstructed into three phases for this project: pre-processing by subject matter experts, automated data pre-processing, and manual data pre-processing. The first two phases are typical of any machine learning project. The third phase contains methods introduced in this chapter. Upon completion, an on-going iterative data pre-processing procedure continued until the end of the project to ensure industrial sponsors were satisfied with the model performance. The objective of the prediction models were to predict the **output flow rate** as a function of process variables.

The data was first pre-processed by experts within the sponsor's organization to remove useless tags such as alarm limits, fire detector status, and the sort. After doing so, the amount of tags was reduced to 124. The remaining data was sent into various automated data pre-processing algorithms to further remove redundant/insignificant tags. The algorithms include: missing data removal, data imbalance analysis, and collinear analysis. Missing data removal cleans up any NaN or empty values in the data set. Data imbalance analysis inspects all categorical variables and removes heavily imbalanced tags where one class significantly dominates all other classes. Lastly, collinear analysis identify heavily correlated columns and removes all but one to prevent redundancy in the data set. Total tags reduced to 65 after the automated feature selection.

Next, the manual pre-processing was conducted on the data set. First, engineering knowledge was applied to the pre-processing step to remove segments of data where the process exhibits abnormal operating behaviour. Then, the data at each

pump station was shifted to incorporate time delays. In pipelines transporting incompressible fluids, pressure propagates down at the speed of sound (1480 km/h) [64]; therefore, all pressure variables were shifted with accordance to their distance to the output of the pipeline (shown in Table 3.3. Likewise, DRA also had time delay. DRA must be transported down the entire pipeline to exhibit its full drag reducing effect. From giving flow rates and the length of each segment of the pipeline, it was calculated that coating the entire pipeline required approximately ten hours. Therefore, the first ten hours data corresponding to a set point change in DRA was removed to ensure the DRA's effect has been fully realized in the output flow rate data.

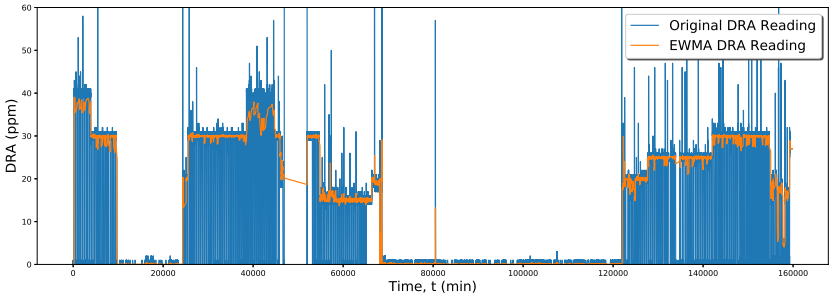
Table 3.3: Time required for pressure changes at each pump station to be realized at refinery.

	B1	B2	B3	B4
Time to refinery at speed of sound in liquids (1480 m/s) [64]	2.0 min	2.0 min	1.0 min	1.0 min

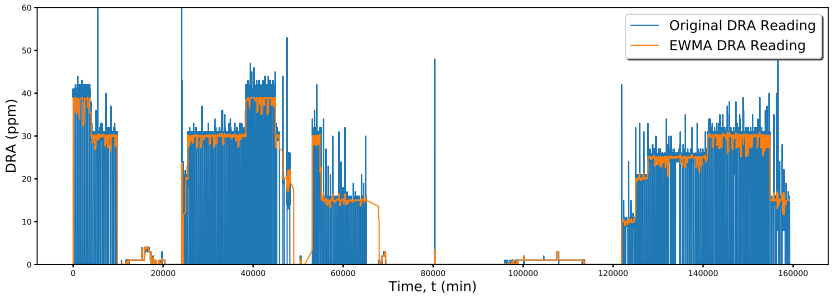
Additionally, the temperature and DRA ppm measurements were highly noisy and were exponentially smoothed using Equations 3.1 to increase reliability. The pre- and post-smoothed values of the DRA are shown in Figure 3.14 to illustrate the effect of exponentially smoothing.

To conclude the data pre-processing procedure, the process operators were consulted with to identify any *special* considerations that should be included into the ML models. Here, the operators stated that the density reading is unreliable and is only used to identify the crude type at each station. From this information, all density readings were feature engineered into binary readings where *1* represented light crude and *0* represented heavy crude to reflect the physical purpose of the reading.

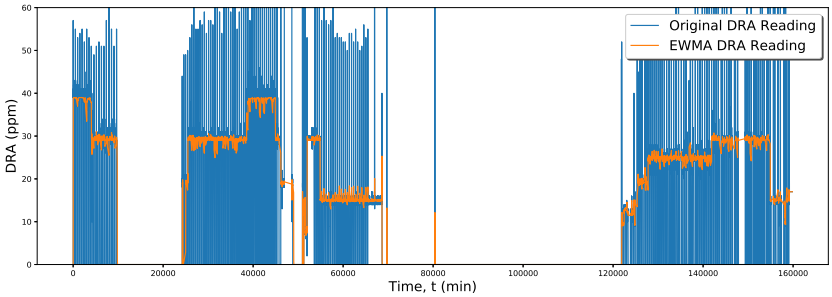
Exploratory data analysis was then performed on the processed data to gain additional insight. In Figure 3.15, a kernel density estimation was applied to the pipeline outlet flow rate to identify its distribution as shown. It can be seen that the flow rate follows a bi-modal distribution and most likely corresponds to two



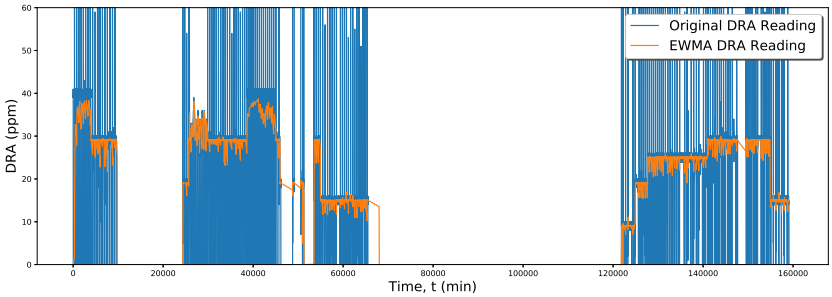
(a) Station B2 heavy DRA sensor reading.



(b) Station B2 light DRA sensor reading.



(c) Station B3 heavy DRA sensor reading.



(d) Station B2 light DRA sensor reading.

Figure 3.14: Pre- and post-processed DRA sensor readings.

different operating regimes. To enhance ML model performance, the two modes were segregated using DBSCAN. Here, DBSCAN was selected due to its scalability to big data and ability to identify outliers. The hyper parameters, ϵ and *min points*, for DBSCAN were 1.13 and 10,000, respectively.

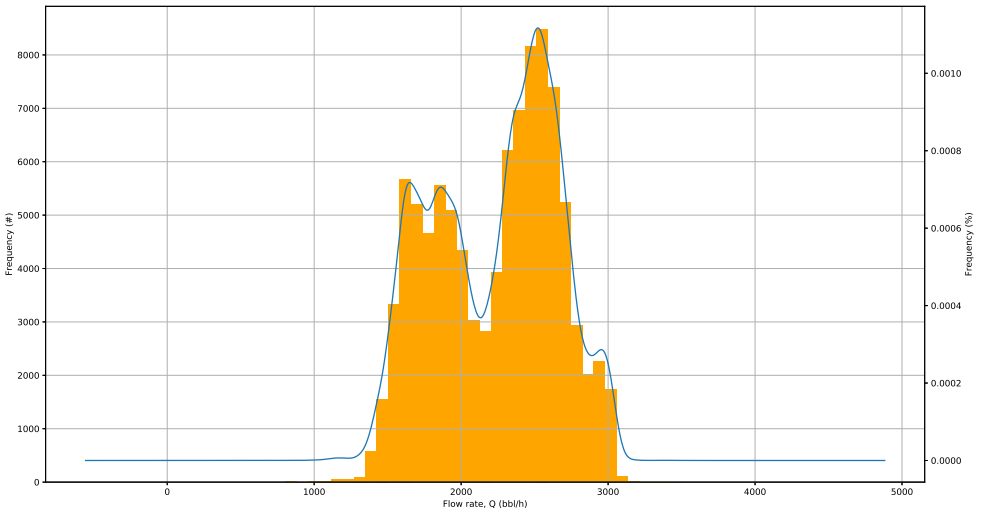


Figure 3.15: Flow rate distribution of the pre-processed data set.

The segregated distributions created from DBSCAN and the average characteristics of each cluster are shown in Figures 3.16 and 3.17, respectively. Evidently from Figure 3.17, the operation of equipment were vastly different in the two operating regimes. More specifically, cluster 2 used almost no DRA, and used Station B3 Pump 1 and Station B1 VFD exclusively. In cluster 1, all equipment were used with exception of Station B3 Pump 1. From this information, two ML models can be created to cater to each cluster.

3.4.3 Machine Learning Predictions

The following models were implemented to predict for the pipeline outlet flow rate: linear models, polynomial models, neural network models, and LPV models. Ultimately, the LPV model reigned supreme due to its interpretability, consistent results, simple model structure. The performance and hyper parameters of each

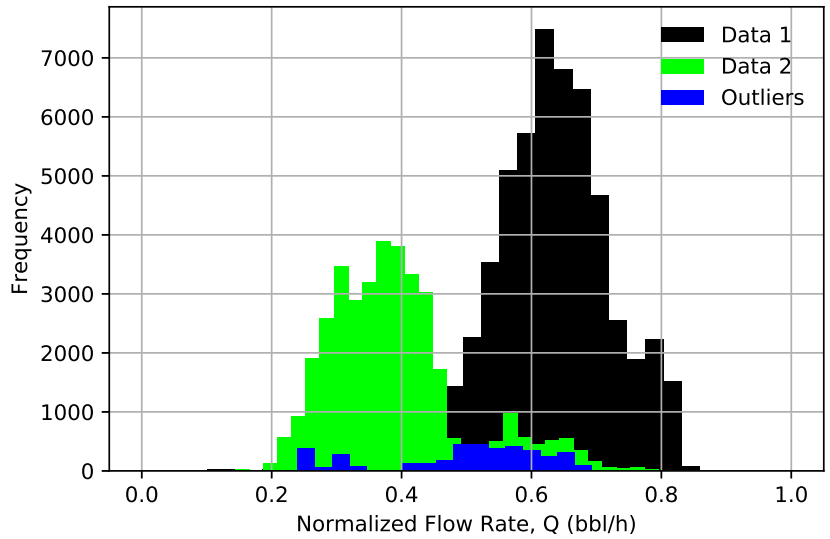


Figure 3.16: Clusters identified after applying the density-based scan.

	Feature	Cluster 1 (Black)	Cluster 2 (Green)
	Flow Rate (bbl/h)	2500	1900
DRA	DRA – Station B2 Heavy (ppm)	High	Low
DRA	DRA – Station B2 Light (ppm)	High	Low
DRA	DRA – Station B3 Heavy (ppm)	High	Low
DRA	DRA – Station B3 Light (ppm)	High	Low
On/off Pump	Station B1 Booster Pump (% on)	100%	0%
On/off Pump	Station B3 Booster Pump 1 (% on)	0%	100%
On/off Pump	Station B3 Booster Pump 2 (% on)	95%	0%
On/off Pump	Station B4 Booster Pump (% on)	63%	0%
VFD	Station B1 VFD (amps)	225	184
VFD	Station B4 VFD (amps)	30	0

Figure 3.17: Average operating variables for the two operating conditions.

model are shown below. During model identification, the data sets were divided into three sections: training, validation and testing. Table 3.4 shows the purpose of each data section.

Linear Models

Linear models pertain the simplest model structure and will serve as a benchmark for other models. The hyper parameters and performance metrics of the model are shown in Table 3.5 and 3.6, respectively. The model's performance on the validation

Table 3.4: Description of each data partition.

	% of Data	Description
Training	90%	Identify the ML model
Validation	5%	Tune ML model performance on unseen data
Testing	5%	Test ML model performance on proxy live data

and test data are shown in Figures 3.18a and 3.18b. From 3.6, the model MAE and RMSE increased 4% and 6% on the testing data, respectively. R^2 on the test data significantly reduced. However, the trends from Figure 3.18b does not exhibit any significant inaccuracies.

Table 3.5: Hyper parameters for linear regression.

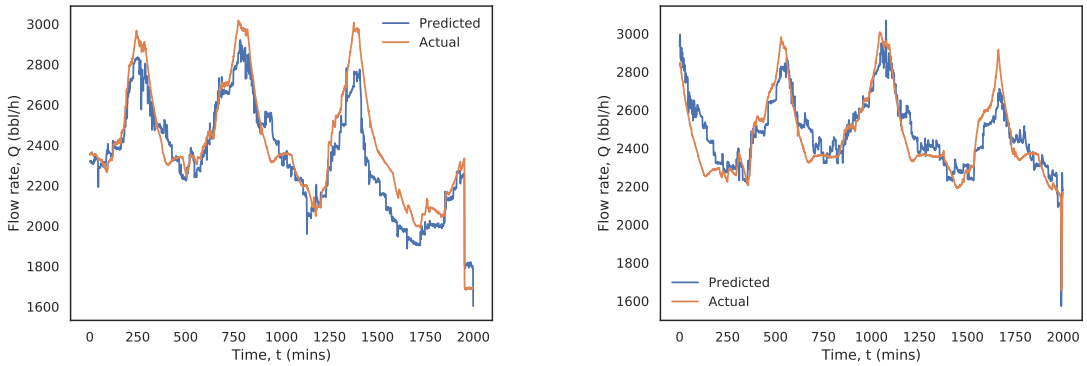
Hyper Parameter	Value
Epochs	800
Minibatch size	8192
Learning rate, α	0.001
Regularization, λ	0.001

Table 3.6: Performance assessment for the linear regression.

	Training data	Validation data	Test data
MAE	98	98	102
RMSE	127	127	135
R^2	0.91	0.91	0.70

Polynomial Models

Non-linear models were used to further enhance predictive capabilities. Two exponential models were applied: quadratic and square root. The hyper parameters and performance metrics of the exponential models are shown in Tables 3.7 and 3.7, respectively. The model performances on the validation and test data sets is shown in Figure 3.19. Compared to the benchmark model, the MAE and RMSE decreased by up to 13% and 15% on the test data set. Moreover, the model performance does not deteriorate when applied on the test data set.



(a) Predicted vs. actual flow rate for the validation data set.

(b) Predicted vs. actual flow rate for the test data set.

Figure 3.18: Linear regression validation and test plots.

Table 3.7: Hyper parameters for exponential regression.

Hyper Parameter	Quadratic	Square root
Epochs	1000	1000
Minibatch size	8192	8192
Learning rate, α	0.001	0.001
Regularization, λ	0.001	0.001

Neural Network Models

Neural networks and deep learning were the most advanced models to be applied to this prediction task. Here, three different neural networks with varying sizes were trained and their performances evaluated.

Tables 3.9 and 3.10 show the hyper parameters and performance metrics of the three neural networks. Figure 3.20 shows the neural networks' performance on the validation and test data sets. From Table 3.10, it is clear that the training and validation error decreased as the neural network increased in size; however, the errors increased significantly on the test data. This was caused by the test data being different than the training data. In this particular case, the test data was collected in the summer months where temperatures increased by up to 10° C compared to the training data. Because of the complexity and parameterization of neural network models, data even slightly different from the training data can have a significant impact on accuracy. Smaller, simpler, more regularized neural network

Table 3.8: Performance assessment for the quad. and sqrt. model.

	Training data		Validation data		Test data	
	Quad	Sqrt	Quad	Sqrt	Quad	Sqrt
MAE	92	89	92	89	89	91
RMSE	121	118	121	117	120	115

models could be used to avoid this behaviour. Indeed, it can be seen that the test data error increased as the size of the network increased. Another disadvantage of neural networks are its lack of interpretability. Investigating the effects of each regressor in this highly non-linear model is nearly impossible; hence, neural networks are black box models and might be undesirable for safety critical systems.

Table 3.9: Hyper parameters for the feed-forward neural network.

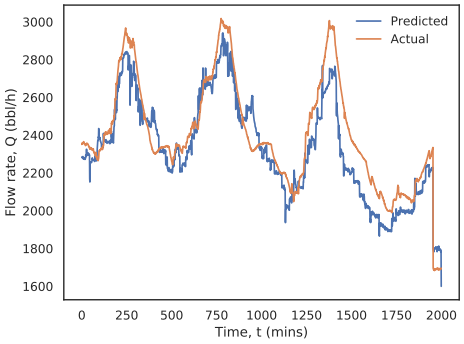
Hyper Parameter	Small NN	Med. NN	Large NN
Epochs	700	1000	1200
Minibatch size	8192	8192	8192
Learning rate, α	0.001	0.001	0.001
Regularization, λ	0.001	0.003	0.005
Number of layers	3	6	8
Neurons per layer	20	30	40
Activation function for hidden layers	ReLU	ReLU	ReLU
Activation function for hidden layers	Linear	Linear	Linear

Table 3.10: Performance assessment of the neural network models.

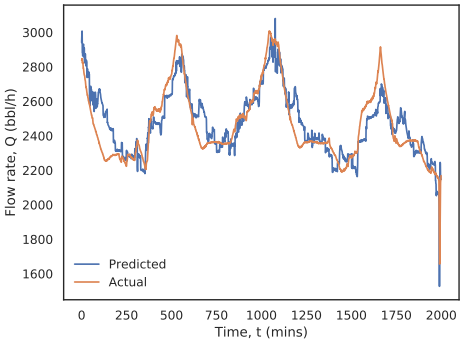
	Training Data			Validation Data			Test Data		
	Sm.	Med.	Lar.	Sm.	Med.	Lar.	Sm.	Med.	Lar.
MAE	48	42	38	50	45	37	87	87	91
RMSE	66	58	57	69	61	56	107	117	118

Linear Parameter-varying Models

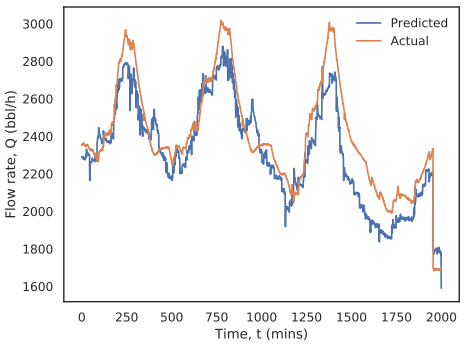
Linear parameter-varying models achieve the performance of non-linear models using a system of linear models. Furthermore, the models are fully interpretable through its weights. From the author's experience, this model structure is an ideal starting point for most industrial applications. For this project, two linear model



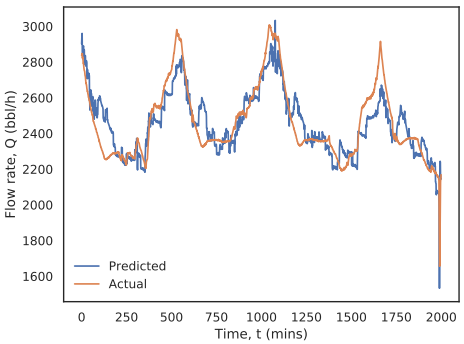
(a) Predicted vs. actual flow rate for validation data using the quad. model.



(b) Predicted vs. actual flow rate for the test data using the quadratic model.



(c) Predicted vs. actual flow rate for the validation data using the sqrt. model.

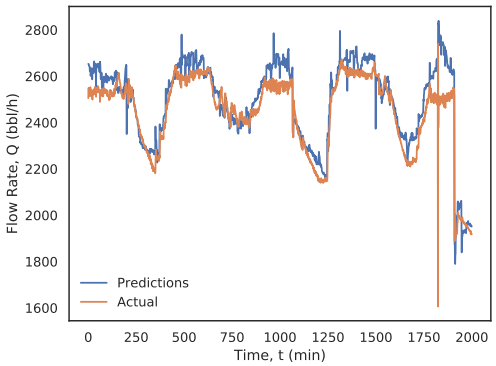


(d) Predicted vs. actual flow rate for the test data using the sqrt. model.

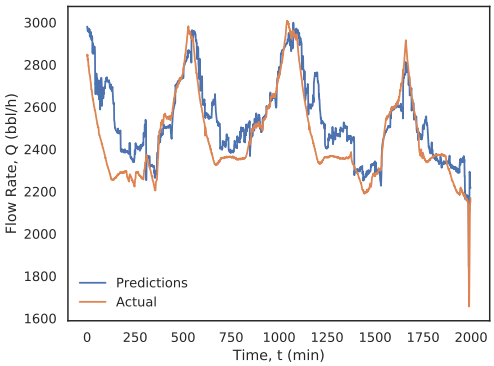
Figure 3.19: Polynomial regression validation and test plots.

were identified; one for each cluster in Figure 3.16.

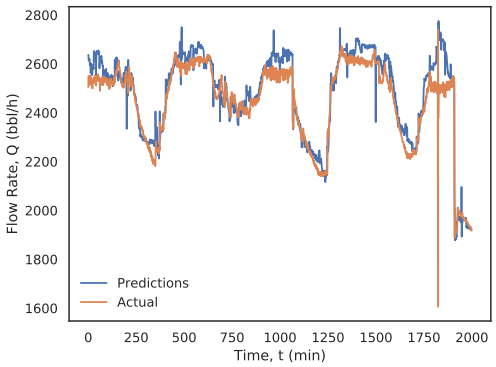
The hyper parameters for each linear model is identical to the values shown in Table 3.5. Performance metrics of the two linear models are shown in Table 3.11. The model performance on the validation and test data sets are shown in Figure 3.21. Average MAE and RMSE of the LPV model is nearly identical to the square root models (the best non-linear model). Moreover, the linear models are simpler and have unique regressors in each model. For example, the linear model for cluster 2 is only a function of the Station B1 VFD current and the DRA ppms because all other inputs did not change in the data set. Visually, the performance figures show low correlation; however, this is because the validation and test data sets were different compared to the previous models since the data set had to be decomposed. Additionally, the flow rate in these new data sets contain higher noise compared to previous data and the y-axis range is also reduced, enhancing the noise. Ultimately,



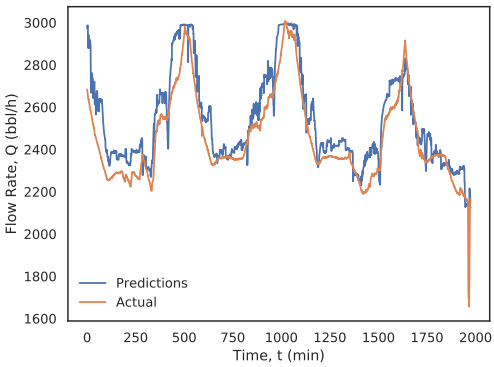
(a) Validation data for the small neural net.



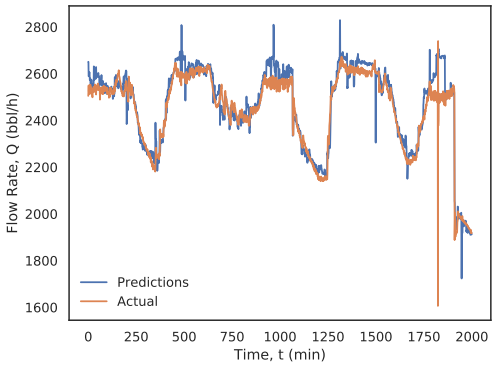
(b) Test data for the small neural net.



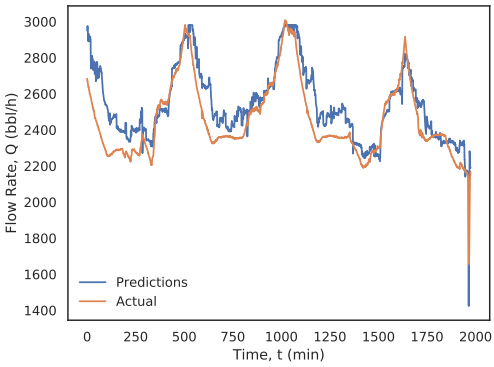
(c) Validation data for the med. neural net.



(d) Test data for the med. neural net.



(e) Validation data for the large neural net.



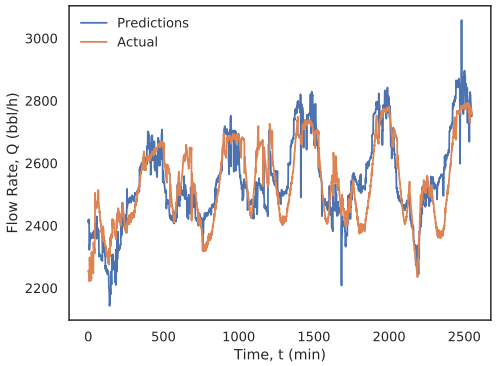
(f) Test data for the large neural net.

Figure 3.20: Predicted vs. actual flow rates for the feed-forward neural networks.

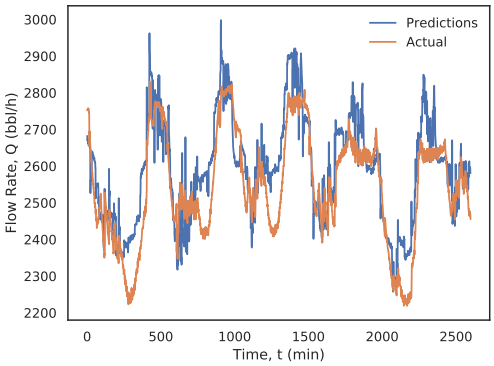
the performance metrics illustrate that the performance of the LPV model is nearly identical to the non-linear models and was the model of choice in this industrial application.

Table 3.11: Performance assessment for clusters 1 and 2 regression models.

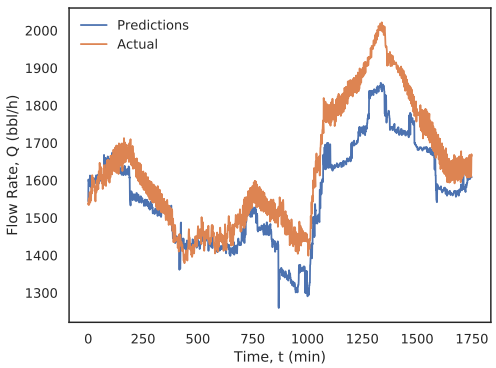
	Training data		Validation data		Test data	
	Cl. 1	Cl. 2	Cl. 1	Cl. 2	Cl. 1	Cl. 2
MAE	90	66	90	67	96	85
RMSE	115	91	116	92	120	110
R^2	0.87	0.90	0.86	0.89	0.78	0.57



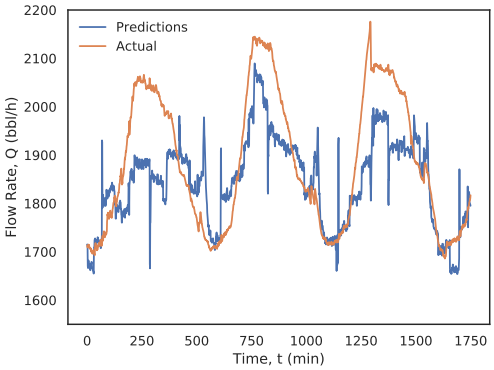
(a) Validation data using model 1.



(b) Test data using model 1.



(c) Validation data using model 2.



(d) Test data using model 2.

Figure 3.21: Predicted vs. actual flow rate for the linear parameter-varying models.

3.4.4 Implementation of USIS

After the model accuracy was deemed acceptable by the industrial sponsors, the next phase of the project aimed to automate machine learning updates, creating a

true self-learning system. The USIS algorithm was applied to automate each update step, while avoiding outliers and catastrophic interference.

The hyper parameters for USIS are summarized in Table 3.12. The model was set to update every 24 hours to avoid any significant changes in model weights. Each update step has a maximum of 1440 (24×60) data points per bin. There were ten bins for each model; each had a width of 200 bbl/h. The bins ranged from 1200 - 3200 bbl/h. The Euclidean distance was used to calculate the distance metric, d. Moreover, the neglect parameter was selected to be 1.13, identical to ϵ. Finally, each replay buffer had the capacity to keep three months data in memory to avoid catastrophic interference. Three months was recommended by the industrial sponsors to ensure relevant information was used for model updates.

Table 3.12: USIS hyper parameters for the pipeline project.

Hyper parameter	Value
# of models, n	2
Model update frequency	24 hrs
Bin size, (bbl/h)	200
Distance metric, d	Euclidean
Neglect parameter, η	1.13
Replay buffer	129,600 (3 months)

The overall USIS algorithm implemented onto the industrial pipeline is shown in Figure 3.22. Raw process data is first inputted into the data pre-processing step where data is smoothed (temperature, DRA), transformed (density data), and normalized. The processed data is then sent into the model selection phase, where one of the two linear models will be selected to perform the prediction. Then, the data is archived in the USIS architecture. After every 24 hours, the models are updated using historical data.

3.4.5 Concluding Remarks on the Pipeline Project

Pipelines are critical assets used to transport fluids across large distances. Common transported fluids include petroleum products, water, sewage, and natural gas.

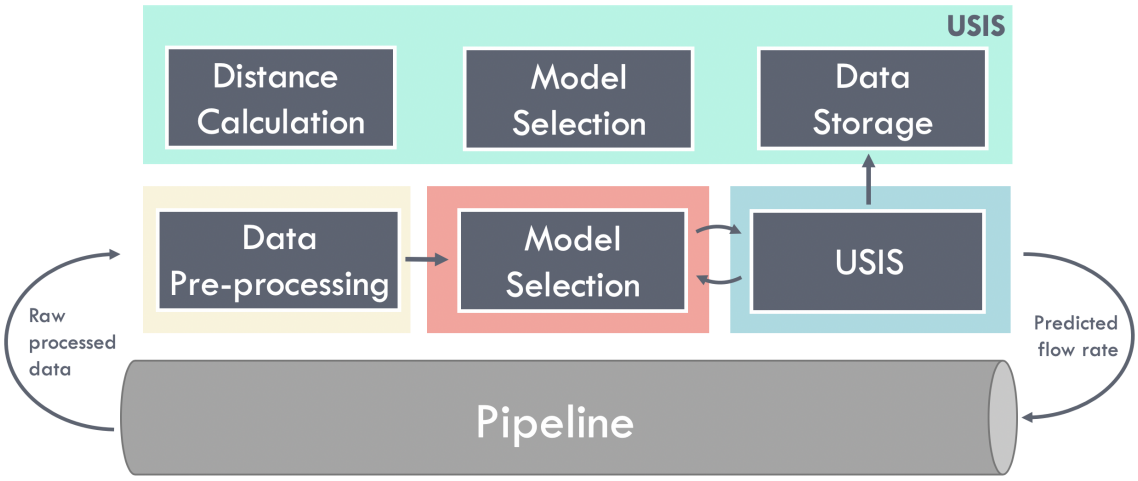


Figure 3.22: An overall look at the USIS algorithm on the industrial pipeline

Due to the mission critical nature of pipelines on society's success, ensuring its reliability and efficiency has global-scale impact.

The first phase of the pipeline project aimed to identify accurate, interpretable, and adaptive models to predict for the output flow rate of the pipeline. To enhance model performance, the data was initially pre-processed vigorously through smoothing, feature selection, feature engineering, and other methods. During model identification, the following models were tried: linear models, exponential models, neural networks with varying sizes, LPV models. Ultimately, LPV models were selected due to their interpretability and predictive capabilities. Adaptability is another key topic of this project. The USIS algorithm was equipped onto the LPV models to allow for outlier-free model updates without suffering from catastrophic interference.

Chapter 4

Machine Learning for Process Monitoring

ML prediction applications are effective complements to existing infrastructure in the process industry through soft sensors, state estimation, and forecasting. However, they are limited in applications regarding safety and risk management. In the process industry, safety is upheld as the greatest *value*; investing in a successful safety system is just good business.

> *Safety is a* ***value****, not a priority. Priorities change, but company values never do.*
>
> — Rex Tillerson, ex-CEO of ExxonMobil

Decades ago, process safety investments are frowned upon by management due to its high costs and *invisible* returns. In fact, construction workers used to cheer when project supervisors announced that *only* 20 deaths will incur for this project—an event completely unacceptable in today's standards [74]. Indeed, a perfect safety and risk management system results in *no change* in day-to-day activities because all the incidents are proactively mitigated. As such, it is incredibly easy to become complacent towards risk management. However, if safety takes a back seat, the occurrence of the next incident is not a matter of if, its a matter of *when*. Therefore, safety must be proactively (not reactively) managed to safeguard people,

the environment, company assets, and production capabilities in terms of physical equipment and the social license to operate. Ultimately, proactive safety and risk management just makes sense.

For enhanced process safety, ML can be leveraged to proactively monitor process systems and create an additional layer of engineering safety factor. In this chapter, ML algorithms will be applied to detect and predict equipment failures, process abnormalities, process variability and also perform alarm management. Through these applications, ML will be used to create multi-variate alarm systems that explore multi-variable interaction effects and gives fewer false alarms. Additionally, a new alarm management system that specifically tackles alarm flood scenarios will be introduced. The objectives of this system are twofold: 1) Reduce sheer number of alarms during a flooding scenario; 2) identify the most important alarms so operators can prioritize safety critical alarms.

This chapter is organized as follows: Section 1 introduces data pre-processing methods for anomaly detection/prediction applications where the data is heavily imbalanced. Section 2 introduces the anomaly detection and prediction algorithms and section 3 concludes this chapter with an introduction to a novel approach for alarm management.

The main contributions of this chapter are the data pre-processing methods used to prepare data sets for anomaly detection/prediction. Additionally, it was shown that using synthetic data was able to enhance accuracy. Lastly, a novel alarm management approach based on reinforcement learning was introduced to filter nuisance alarms and sort alarms based on their priority.

4.1 Data Pre-processing for Monitoring

Data containing anomalous and/or incident events are extremely rare—thankfully—in the process industry.

Anomaly or anomalous activity: An abnormal or unexpected event given

other variables (often multivariate). For example, a person walking in a t-shirt when it is -30° C outside.

In fact, it is not uncommon to have just one incident in a data set containing hundreds of thousands of records. Under such circumstances, building ML models to identify incidents is extremely difficult. Remember, ML models are nothing more than statistical models with training formulated in an incremental updating fashion. In the scenario where the training data set contains 999,999 non-anomalous activities with 1 anomalous activity, the model will simply learn to return non-anomalous for all inputs; such a model would still achieve 99.9999% accuracy on the training data! When a human is provided with this data set, the human would instead focus most of its attention on the one anomalous activity, studying how it is different from all the other points. A *tabula rasa* machine is simply not equipped with such cognitive abilities, and will treat every data point equally; however, humans can artificially provide cognition to the machine.

4.1.1 Data Prep for Anomaly Detection

Anomaly *detection* tasks are quite simple compared to anomaly *prediction* tasks that will be discussed later on in this section. In anomaly detection, the model simply has to classify if there is an anomaly at current time. For example, given some states of a reactor, is the output temperature anomalous? That is, is the measured output temperature expected given the states? Of course, such questions are difficult for humans to answer, especially in multi-variate environments; however, such questions can easily be answered by machines. Unfortunately, the events that are of interest to us (anomalies) are, often times, significantly more rare compared to normal process data. Hence, the leading reason for poor model performance in imbalanced data sets (i.e., distribution of different classes are vastly different) is the *imbalanced* nature. There exists two traditional ways to tackle the data imbalance issue: **undersampling** and **oversampling** [75]. Objectively, both methods aim to re-balance the data set so that the positive and negative data are in harmony.

A visual description of undersampling and oversampling is shown in Figure 4.1. In undersampling, the majority class(es) are significantly down-sampled to be the same size as the minority class. The obvious flaw with this technique is the significant data in-efficiency. In this method, the majority of the data that may contain critical features and information are simply discarded. To take advantage of the whole data set, oversampling is sometimes used. In oversampling, the minority class data is copied n times until is approximately equivalent to the majority class. Although oversampling allows the whole data set to be used, the models built typically overfit to the minority data simply because it was copied so many times. During deployment, anomalies that are even slightly different from the ones in the training minority data set are often times misclassified.

Another more mathematical way to tackle this data imbalance issue is to bias the cost function to **heavily** discourage misclassification of positive samples. However, such a methods requires tuning of the cost function, and is often times difficult.

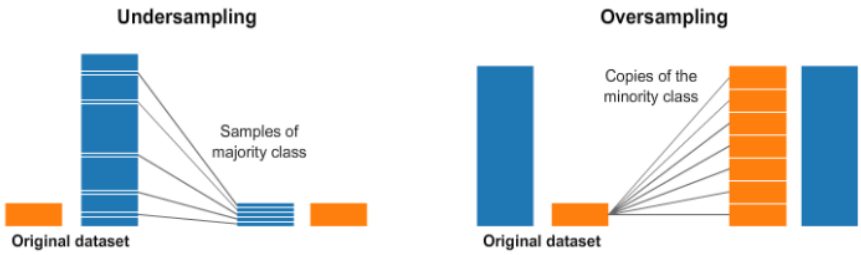


Figure 4.1: A visual representation of undersampling (left) and oversampling (right). Original image from Kaggle.

Luckily, normal process data are typically plentiful and lack increasing information (i.e., data is typically from steady state processes that hover around the same value for weeks) in the process industry. Therefore, discarding large bulks of data in certain processes do not affect the ultimate accuracy of the model. For anomaly detection, the majority class data was first undersampled to be comparable to the minority class. Undersampling should uniformly sample the data set to ensure all operating regimes are sufficiently captured. Then, a label column is generated; all anomalous events were given a value of 1 and normal process data had labels of 0. An example of undersampling is shown below. In this example, the process is

deemed anomalous when $x_1 + x_2 > 3$.

Table 4.1: Original data set before undersampling.

x_1	x_2	label
2	-1	0
3	0	0
2	5	1
2	-2	0
5	-1	1
-2	-1	0
1	-3	0
2	-4	0
-2	2	0
-2	4	0

Notice that in Table 4.1, the majority class dominates the minority class 5:1. After downsampling, the data set becomes:

Table 4.2: Original data set before undersampling.

x_1	x_2	label
2	5	1
2	-2	0
5	-1	1
-2	-1	0

Note that the majority and minority classes do not have to be perfectly balanced, especially in cases where perfectly balancing the classes require discarding unique information from the majority class.

4.1.2 Data Prep. for Anomaly Prediction

A visual breakdown of the anomaly prediction task is shown in Figure 4.2. From Figure 4.2, the *old event* refers to the anomaly detection problem. The *new event* denotes the latest time step to predict an anomalous event. Overall, anomaly prediction is a much more difficult problem compared to anomaly detection. In anomaly prediction, the model must *predict* if an anomaly is going to happen *before* it happens. Compared to anomaly detection, this task is much more difficult, and how far in advance an anomaly can be detected is heavily dependent on the speed of dynamics of the system. For example, in cat classification, anomaly detection simply has to identify if the picture is a cat or not. However, anomaly prediction has to predict if the animal in each picture will eventually grow up to become a cat.

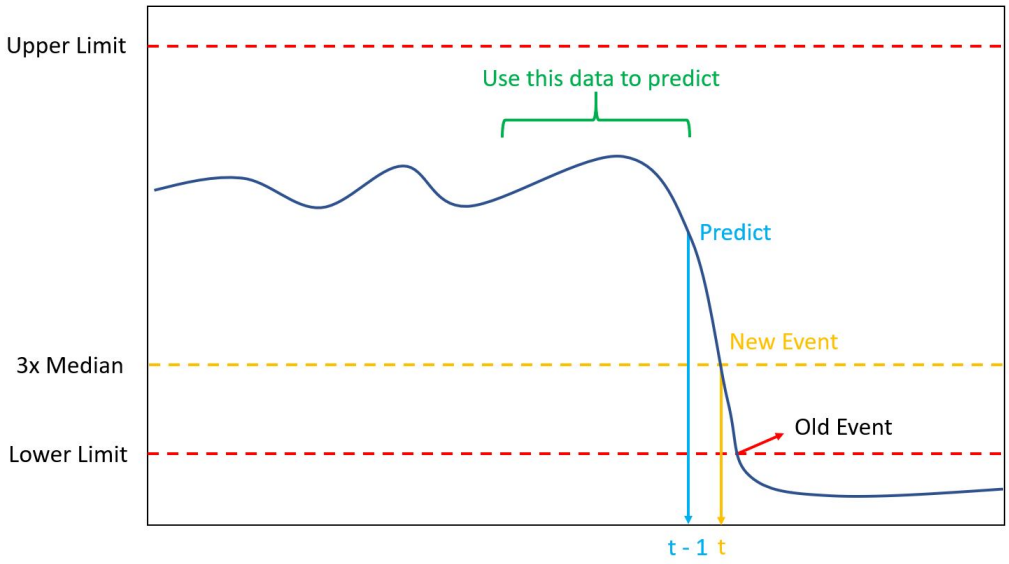


Figure 4.2: Visualization of the anomaly prediction.

In anomaly prediction, the data is first labelled as in anomaly detection. Afterwards, the data is augmented as follows:

$$x = [x_{t-l-L}, ..., x_{t-l-3}, x_{t-l-2}, x_{t-l-1}, x_{t-l}] \tag{4.1}$$

where $l > 0$ denotes the minimum number of sampling time the model should predict in advance. L represents the amount of previous information to be provided to the

model. Augmenting the data as such provides temporal information to the ML algorithms to perform more accurate predictions. Note that even though an l of 10 is chosen, this does not guarantee that the model will always predict anomalies 10 time steps in advance. Moreover, if the model predicts positive, it does not mean that an anomaly will occur exactly 10 time steps later. Merely, it just biases the algorithm to be more effective around that specific time range. How early an anomaly can be detected is purely dependent upon the dynamics of the system. For example, if the degradation of an equipment is slow and gradual, the anomaly might be detected days in advance; however, an instantaneous failure offers no time for any early detection.

Like in anomaly detection, the data is heavily skewed towards normal process data; therefore, the majority class must also be downsampled to be comparable to the minority data set. Downsampling occurs last to ensure no temporal relationships are broken. A quantitative example for the augmentation of an anomalous data is shown below. Here, suppose downsampling is not required (since an example is already shown above) and $l = 1; L = 2$.

Table 4.3: An arbitrary time-series data set.

time	x_2	label
0	5	0
1	3	0
2	7	0
3	6	0
4	4	0
5	2	0
6	-1	1
7	-3	0
8	-1	0
9	2	0

From Table 4.3, the time series augmented data point for the anomalous event would be:

$$x = [x_3, x_4, x_5]$$

$$x = [6, 4, 2]$$

4.1.3 Synthetic Data Generation

In the above methods, the assumption of plentiful data was made. In industry, this is not always true, especially for the anomalous data. Here, three different *synthetic* data generation methods will be introduced, with each method generating fake, yet similar, minority class data. This topic is an especially popular research topic for the computer vision field where good, labeled data are rare. In fact, many *Completely Automated Public Turing test to tell Computers and Humans Apart* (CAPTCHAs) use traffic signs to force potential users of the website to first label some data, before being allowed to proceed. Most likely, the labeled data are then sold to computer vision companies. The main idea of synthetic data generation is to construct fake data that is *exactly equivalent* to the real data that even a subject matter expert cannot tell them apart. Indeed, that was exactly the structure of one of the most advanced generative methods, the generative adversarial network (GAN) [76]. Synthetic data research in time-series data are unfortunately more primitive compared to GANs, but still provide valuable accuracy gains in the final model.

Time-series oversampling

The first method caters to time-series data. This method simply oversamples the data leading up to an event. Notice that this method is different from oversampling because it does not directly copy the data. Instead, the sampling rate is increased for periods leading up to an anomalous event (for anomaly prediction) or during the anomalous event (for anomaly detection). For example, the normal sampling time of a process might be one per minute. But to obtain more data, the resolution might be greatly enhanced to one per 10 seconds to increase anomalous data.

SMOTE

The second method for synthetic data generation is the Synthetic Minority Over-sampling Technique (SMOTE). As a high level overview, SMOTE generates synthetic minority data through combining features of real minority data points. Suppose we plotted a 2-class data set with 2 features. Most likely, data points corresponding to the two classes will be segregated (at least slightly). To generate synthetic data on either class:

1. Start with an arbitrary point within that class
2. Identify the distance between that point and it's closest neighbour (within the same class). Typically, Euclidean distance is used for the distance metric and is given by:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n + q_n)^2} \tag{4.2}$$

   where p and q denotes two arbitrary points belonging to the same class. Here, n denotes the total number of features for p and q.
3. Multiply the Euclidean distance by an arbitrary number, r, between 0 - 1.
4. Place a new data point $r \times d$ from the original point

A visual representation of SMOTE is shown in Figure 4.3. For more information regarding SMOTE, see [77].

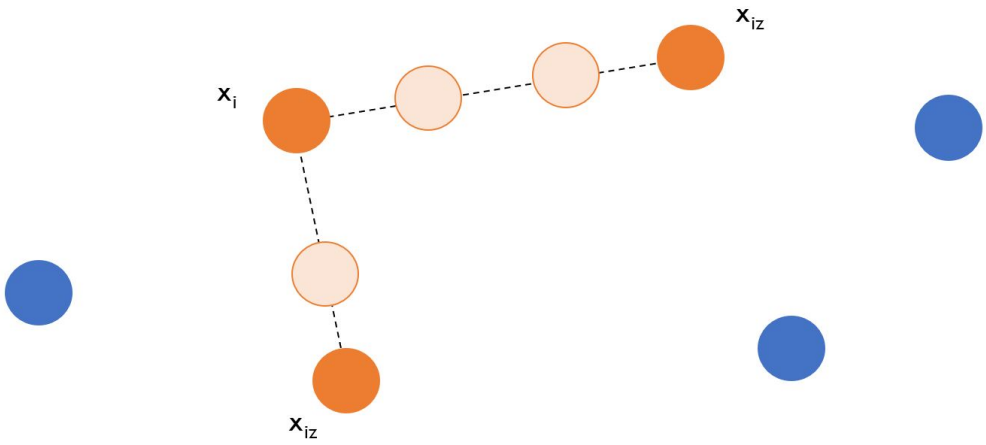


Figure 4.3: A visual representation of SMOTE.

ADASYN

The last method is an adaptive version of SMOTE called ADASYN. ADASYN offers *targeted* data generation—more synthetic data is generated for neighbourhoods where the minority class is heavily dominated by the majority class. For example, suppose the minority class is distributed into two distinct clusters, with one cluster having significantly more minority data compared to the other. When training a ML model on such a data set, the model will perform significantly better on the minority-dense cluster. ADASYN can be applied here to target more data generation on the less dense cluster, resulting in comparable model performance for both clusters. However, ADASYN does not work for sparsely distributed minority classes where only one data point is present. Additionally, some less dense clusters are a result of noise. When ADASYN is applied on such clusters, inaccurate minority data will be created which greatly decrease model accuracy during deployment. A more detailed explanation of ADASYN can be found in [78].

4.2 Anomaly Detection and Prediction

One of the simplest, yet very effective, classification machine learning algorithm is the logistic regression [79]. Logistic regression is a binary *classification* algorithm (although called regression) that outputs a value between 0 and 1, denoting the

probability of a certain event occurring. For example, the output value to a logistic regression model trained on pump failure data denotes the probability of the pump failing. The model structure of the logistic regression is given as:

$$\hat{y} = \frac{1}{1 + e^{-(W_1^T x_1 + W_2^T x_2 + ... + b)}} \tag{4.3}$$

where e denotes the exponential operator. For multi-class classification, softmax functions are typically used and are given by:

$$y = \frac{e^{z_i}}{\sum_{j=0}^{k} e^{z_j}} \tag{4.4}$$

where z_i is given as:

$$z_i = W_{i,1}^T x_1 + W_{i,2}^T x_2 + ... + b_i$$

The dimensions of the parameter matrices are $W_{j \times k}$ and $b_{j \times 1}$. Here, j amd k denote the number of classes and the number of features for each data point, respectively. Lastly, softmax are typically used because the function is continuously differentiable, thus allowing for gradient methods to be viable.

4.2.1 Deep Learning Classification and Prediction

A deep learning binary classification model simply modifies the regression neural network (introduced in the neural networks basic section in Chapter 1) by replacing the last layer's activation function to a sigmoid function. More specifically, the following is the mathematical procedure of the regression neural network model:

$$z_j^{[1]} = W^{[1]} x + b^{[1]}$$
$$a_j^{[1]} = ReLU(z_j^{[1]})$$
$$z_j^{[2]} = W^{[2]} a_j^{[1]} + b^{[2]}$$
$$a_j^{[2]} = ReLU(z_j^{[2]})$$
$$...$$
$$z_j^{[r]} = W^{[r]} a_j^{[r-1]} + b^{[r]}$$

$$a_j^{[r]} = ReLU(z_j^{[r]})$$
$$y = W^{[o]} a_j^{[r]} + b^{[o]}$$

In classification, the last step is simply replaced with:

$$z = W^{[o]} a_j^{[r]} + b^{[o]}$$

$$y = \frac{1}{1 + e^{-z}} \tag{4.5}$$

For multi-class deep learning classification, the output activation layer is replaced with the softmax function given in Equation 4.4.

4.2.2 Cost Function for Classification

The classification models are trained using the following convex cost function:

$$J = \frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(f(x_i)) + (1 - y_i) \cdot log(1 - f(x_i)) \tag{4.6}$$

where N denotes the total number of training data used for this update step (i.e., the size of the mini-batch). Here, y_i is the label of the i^{th} training data and $f(x_i)$ is the probability output of the classification model. Intuitively, the cost function penalizes incorrect misclassifications. For example, if $y_i = 1$ and $f(x_i) = 1$, then the cost function would be zero. Likewise, if $y_i = 1$ and $f(x_i) = 0$, the cost function would instead return 1.

4.3 Model Performance Assessment

Often times, accuracy (i.e., % of times the model predicted accurately) is a poor performance metric for heavily imbalanced data sets. For example, a model that only predicts false for a classification data set where 99% of the data is false will still result in a 99% accuracy even though the model has no predictive capabilities. In such data sets, **precision** and **recall** are more proper evaluations of model performance

[80].

Precision: Percent of time the model successfully predicted a true positive and is given as:

$$Precision = \frac{TP}{TP + FP} \tag{4.7}$$

where TP and FP denotes the true and false positives, respectively. A model with poor precision results in excessive false alarms and lead to operator complacency quickly.

Recall: Percent of total events detected, given as:

$$Recall = \frac{TP}{TP + FP} \tag{4.8}$$

A poor recall model misses many anomalous events.

Typically, there is a trade-off between precision and recall for traditional methods. This could be eliminated, to an extent, using deep learning models trained on a large repository of data [39]. An acceptable precision and recall depends on the particular application. For example, highly safety critical systems would require a near perfect recall because even missing one event could lead to catastrophic damage; therefore, a degree of false alarms is acceptable. On the other hand, safety non-critical applications may favor a high precision model where every alarm should be guaranteed to correspond to an actual event. In non-safety critical applications, false alarms may lead to operator complacency. There do exist models with both high precision and recall; however, such models require vastly more data to identify.

4.3.1 Interpreting ML Models

Another critical requirement of ML models is that it must provide **real value** to the operators. The algorithms presented here create value in two ways: 1) Provide explainability to the models which can increase intuition and gain addition buy-in from project shareholders and potential users; 2) provide actionable recommendations to operators. It is almost useless to tell the operators that the plant will blow

up in 10 minutes if no details on how to avoid such a fate is not provided.

The model weights can be analyzed to provide explainability for the logistic regression model.

The explainability of neural networks are significantly more difficult compared to logistic regression. Indeed, even providing a rough explanation is extremely difficult. Some simple approaches include permutation importance, partial dependence plots, and SHAP.

Permutation importance: Permutation importance identifies the importance of each input feature to the ML model and is applied *after* the model is identified. In permutation importance, the columns of the features are shuffled, one at a time. After each shuffle, the model is re-evaluated with one incorrect feature data. Here, if the model's performance significantly reduces after the shuffling of a feature, that shuffled feature is deemed to have high predictive power. On the other hand, if the model performance is unaffected, then the shuffled feature is assumed to have little to no predictive power. This step is repeated for all features in the feature space. More details regarding permutation importance can be found in [81], an example of feature shuffling is shown in Figure 4.4.

Height at age 20 (cm)	Height at age 10 (cm)	...	Socks owned at age 10
182	155	...	20
175	147	...	10
...	...	...	...
156	142	...	8
153	130	...	24

Figure 4.4: A visual example of permutation importance. Original image from [82].

Partial dependence plots: Partial dependence plots (PDPs) are also evaluated after a model is identified and are used to show how each feature affects the final model prediction. On a high level, PDPs are similar to evaluating the weights of the models, except PDPs also capture additional, more complex, relationships. Basically, the fitted model is used to predict for the output while keeping all variables constant except for one.

Figure 4.5 shows an example of a PDP plot for one variable. The y-axis shows the change in the prediction (in this case, winning *Player of the Game* in soccer) while the x-axis is the number of goals scored. Here, the number of goals scored is the variable being manipulated. The plot shows how the chance of winning *Player of the Game* changes as more goals are scored by a player. In this particular example, the PDP shows that scoring one goal helps tremendously in obtaining player of the game; however, any additional goals provide no impact. For more information on PDPs, see [83].

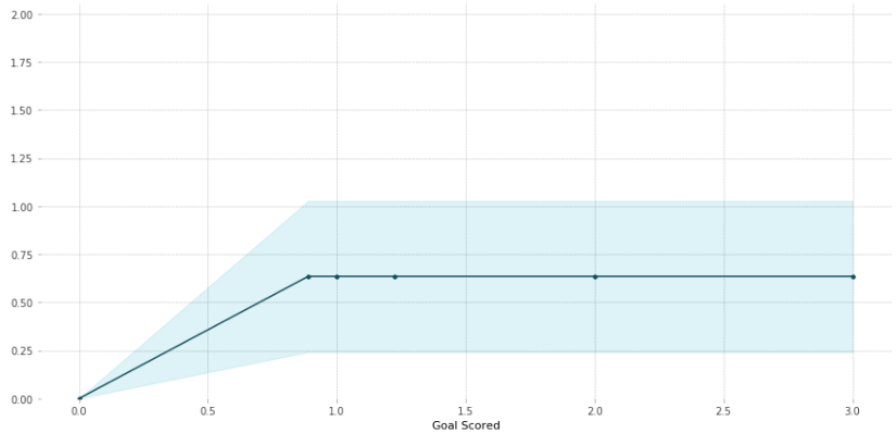


Figure 4.5: A visual example of permutation importance. Original image from [84].

SHAP: Shapley additive explanations (SHAP) are used to decompose model predictions so that the impact factor of each variable on the final prediction can be identified. This analysis is critical for safety-sensitive systems; by applying SHAP, positive anomaly predictions can be decomposed to identify the root case (i.e., which feature is causing this prediction to be positive). Overall, SHAP provide values that interprets the impact of having different values for certain features compared to if that feature took a baseline value. For example, PDPs show how different the prediction would be, given a change in one variable. Instead, SHAP shows how the prediction is affected if one variable was changed, compared to if that variable took some baseline value. In a multi-feature problem, a SHAP value is calculated from:

$$y_{shap} = \sum y - y_{baseline} \tag{4.9}$$

That is, the difference between what was predicted from the actual variable and what would have been predicted if a baseline value was used. The sum of all individual SHAP values correspond to how different the predicted value is from the baseline. Afterwards, a SHAP decomposition graph, as shown in Figure 4.6, can be generated to explain exactly how the prediction was constructed. Figure 4.6 was originally generated to predict the % chance of winning the *Player of the Game* award in a soccer match.

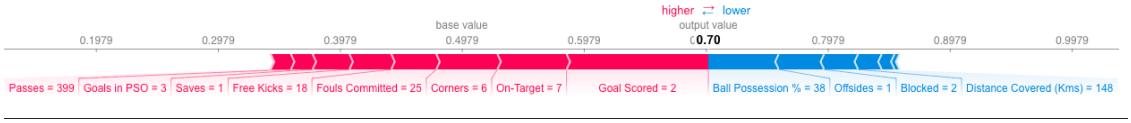


Figure 4.6: A visual example of permutation importance. Original image from [85].

From Figure 4.6, the red and blue arrows represent variables that resulted in a higher and lower chance of winning the *Player of the Game* award. For example, it can be seen that scoring two goals increased the chances of winning the award drastically; however, the ball possession being at 38% reduced the chances. A more detailed description of SHAP can be found in [86].

4.4 Industrial Application of ML Monitoring

The above algorithms were implemented onto an industrial pipeline for monitoring of an unreliable variable frequency drive (VFD) pump. Often times, the VFD would trip. When the VFD trips, the output pressure of the VFD drops to near zero. Initially, plant managers just wanted to detect when the VFDs trip through anomaly detection. For the second phase of the project, the managers wanted to manage the VFDs proactively; thus, anomaly prediction was used to predict *in advance* when the VFDs will trip. Furthermore, management wanted the largest contributors of the VFD trip.

4.4.1 Anomaly Detection

In total, the data set contained 54 anomalous events in a span of one year (525,600 data points) and are shown in Figure 4.7. Some anomalous events lasted up to a few days but most ended within a few hours. In total, approximately 15% of the data was anomalous (any values dipping below the dashed red line).

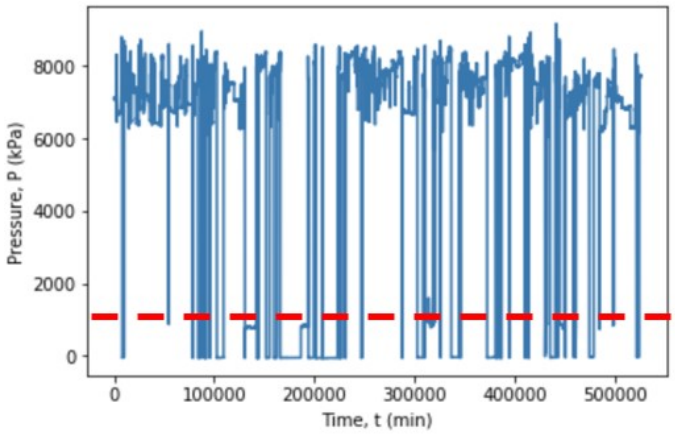


Figure 4.7: Anomalous events of a pump in an industrial pipeline.

The number of anomalous and non-anomalous data in the training and validation data sets are shown in Table 4.4. In this study, an anomalous sample is any point below the dashed red line shown in Figure 4.7. For training and validation purposes, the data underwent a 90/10 split for the anomalous data. In the end, 70,956 anomalous events were contained in the training data set and the remaining 7884 were in the validation data set. The data sets were then mixed with an equal number of non-anomalous data points. No synthetic data generation methods were used here because there exists enough anomalous events to build an effective model.

Table 4.4: Training and validation data split.

	Anomalous	Non-anomalous
Training	70,956	70,956
Validation	7884	7884

Data pre-processing followed similar methods as shown in Chapter 2 to initially reduce the number of redundant and/or unnecessary features. The initial 785 in-

put features were reduced to 64 following consulting with subject matter experts, redundancy analysis, and other analyses. Afterwards, logistic regression was applied onto the training data set. The learning rate, mini-batch size, and number of training epochs for the model are 0.001, 256, and 800, respectively. The threshold for a positive classification was set at 0.5. Ultimately, the algorithm was able to achieve 99.6% accuracy, 99.3% precision, and 100% recall on the training data. The performance on the validation data was 99.7% accuracy, 99.6% precision, and 100% recall. From the results, it can be seen that deep learning was not required.

The largest contributors to the prediction model were:

1. Motor Vibration
2. Current Imbalance
3. Ground Current
4. Discharge Pressure

Although the model above achieved high accuracy in detecting VFD trips, it is a reactive approach and does not prevent the incident from occurring. Next, anomaly prediction algorithms will be shown where the events are predicted prematurely.

4.4.2 Anomaly Prediction

Compared to anomaly detection, anomaly prediction is a much more difficult task. Additionally, the amount of data present will be reduced tremendously. That is, previously, there were approximately 78,840 anomalous data points; however, anomaly prediction can only use data right *before* an incident occurs because its goal is to identify the dynamics behaviour of incidents before they occur. Therefore, the total available data for anomaly prediction is only 54—the total number of events.

The anomaly prediction training and validation data also underwent a 90/10 split where 48 incidents were in the training data set and 6 incidents were in the

validation data set. The data was then balanced with non-anomalous examples. For the data augmentation, $l = 1$ and $L = 3$ (i.e., predict at least 1 step before the incident, and use the past 3 time step information for the prediction). The hyper parameters of the logistic regression were identical to those used in anomaly detection.

Table 4.5 shows the performance of the anomaly prediction algorithm using different parameters. In the headings of Table 4.5, *Syn. Data* represents the amount of synthetic generated data points using ADASYN. Here, 1× means 48 additional synthetic examples (1× the total incidents). N:A Ratio denotes the normal to anomalous data ratio. For example, a N:A ratio of 12:1 means that the normal data points outnumber the anomalous data points 12 to 1. From Table 4.5, it can be seen that the initial precision was extremely low. With the addition of more normal data points, the precision increased significantly without much reduction in recall. By tuning solely the N:A ratio, the best results obtained were 58% precision and 98% recall. To further increase accuracy, ADASYN was used to generate synthetic data. After applying ADASYN, the best results achieved increased to 81% precision and 92% recall; still a large gap to ideal performance.

In further improve accuracy, a three layer deep learning classifier was used. The learning rate, mini-batch size, and training epochs were set to be the same as the logistic regression. Ultimately, the deep learning classifier was able to achieve both 100% precision and recall.

Table 4.5: Precision and recall of the anomaly prediction algorithm using different parameters.

Precision	Recall	Activation	Syn. Data	N:A Ratio	Algorithm
15%	98%	0.70	0	1:1	Log. regression
30%	79%	0.95	0	1:1	Log. regression
54%	96%	0.70	0	5:1	Log. regression
56%	93%	0.82	0	5:1	Log. regression
41%	38%	0.95	0	5:1	Log. regression
40%	96%	0.70	0	8:1	Log. regression
42%	92%	0.83	0	8:1	Log. regression
58%	98%	0.70	0	10:1	Log. regression
57%	92%	0.83	0	10:1	Log. regression
53%	96%	0.70	0	12:1	Log. regression
57%	96%	0.83	0	12:1	Log. regression
54%	98%	0.70	1×	10:1	Log. regression
62%	92%	0.83	1×	10:1	Log. regression
75%	96%	0.70	5×	10:1	Log. regression
81%	92%	0.83	5×	10:1	Log. regression
100%	100%	0.5	5×	5:1	Deep learning

An example of the prediction algorithm in action is shown in Figure 4.8. It can be seen that the algorithm predicted the incident approximately 11 minutes in advance. The short prediction window is due to the suddenness of the events in this study.

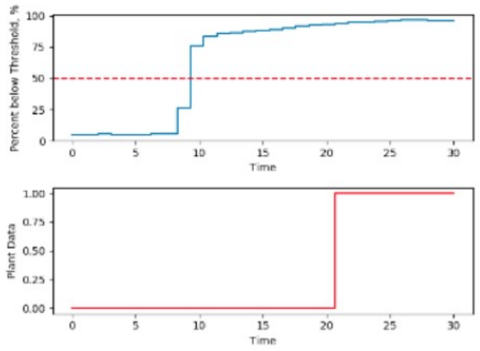


Figure 4.8: Anomaly prediction of an incident. Prediction algorithm output (top) as the incident gets closer (bottom).

4.5 Alarm Management

Industry today is plagued with many problems that require advanced algorithms to overcome. Two of the largest problems are production optimization and alarm management. With continually pressure from environmental groups and ever-increasing government regulations, large industrial companies are forced to reduce their environmental footprints while improving output quality. The status quo also believes in zero-incident policies, i.e., all workplace incidents are preventable and unacceptable; therefore, it is in the best interest of the companies to implement effective, yet cheap, safety systems or their social license to operate could be compromised. To tackle these issues, we applied artificial intelligence (AI) algorithms in conjunction with classical approaches to a wastewater treatment plant (WWTP)[1]. The objectives were threefold: i) Design self-learning and adaptive RL controllers to seek out optimal operating strategies. In this case, the controllers must identify the optimal policy to meet government regulations in the most energy efficient way; ii) direct adaptive control, allowing the RL controller to learn optimal operating strategies as the operating conditions change by adapting the policy directly (not model re-identification), i.e. adapting to changes in season, new government regulations, etc; iii) superior alarm management by developing state-of-the-art alarm reduction and prioritization algorithms through pattern recognition and communi-

[1]This project was funded in part through an engage program with NTwist.

cation establishment between RL and the alarm system. Objectives 1 and 2 will be discussed in Chapter 4, objective 3 will be discussed in the following subsections.

Traditionally, alarms were the first line of defense against abnormalities in chemical processes, and were very effective [87]. Due to their cost of implementation, many teams of engineers and subject matter experts would gather together to brain storm the most effective alarm strategies. However, today's plants are littered with thousands of alarms due to the price of alarms plummeting after the invention of digital alarms. And because of their sheer number, many alarms are redundant and convey no additional information. This project aims to firstly reduce the amount of alarm floods through suppression and pattern recognition. Secondly, an alarm prioritization algorithm will be introduced so operators can focus their attention on the highest impact alarms first.

4.5.1 System Description

Figure 4.9 shows a schematic diagram of the WWTP. A dynamic model of the WWTP was first built using the Benchmark Simulation No.1 documentation from the International Water Association [88]. The WWTP comprises of a five-compartment activated sludge reactor and a ten-stage gravity separator. The first two compartments of the reactor are anoxic tanks, the last three compartments are aerobic tanks. The two control loops are:

- Manipulating the oxygen transfer coefficient ($K_L a_5$) to control the dissolved oxygen level in tank 5
- Manipulating the internal recycle flow rate (Q_a) to control the nitrogen level in tank 2

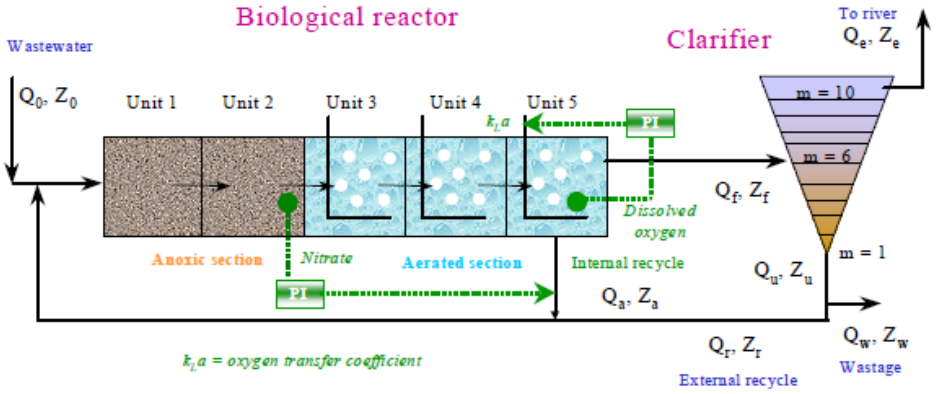


Figure 4.9: Schematic of the WWTP process. Original image from [88].

The WWTP contains 145 states, 2 control actions, and 14 disturbances. The states describe the characteristics of the overall WWTP system and contain process variables such as the flow rate and product compositions. There are three sets of disturbances available to the WWTP: i) Dry weather data. ii) Rain weather data. iii) Storm weather data. The results presented here use the dry weather data. For more detailed information regarding the WWTP, please refer to the Benchmark Simulation No.1 documentation [88].

There exists five environmental constraints on the process:

1. Total nitrogen (N_{tot})
2. Chemical oxygen demand (COD)
3. $NH_4^+ + NH_3$ nitrogen (S_{NH})
4. Total suspended solids (TSS)
5. Biochemical oxygen demand (BOD)

The threshold for the environmental constraints is given as follows:

$$Alarm = \begin{cases} On, & N_{tot} > 10 \\ On, & COD > 100 \\ On, & S_{nh} > 4 \\ On, & TSS > 30 \\ On, & BOD > 10 \end{cases} \tag{4.10}$$

If any of the above constraints are violated, an alarm will sound.

4.5.2 Basics of Alarms

In simple terms, alarm management is a special case of fault detection problem. Let $f_i \in \mathcal{F}$ be a set of possible faults and $A_i \in \mathcal{A}$ be a set of possible alarms. On a high level, $f_i \rightarrow A_i$. That is, all alarms are generated by some fault or some sequences of faults. In alarm management, the objective is reversed. That is, A_i is given, and the fault, f_i, that caused A_i must be identified instead.

In industrial processes, alarm floods are mostly caused by chattering and redundant alarms or by poor alarm threshold design [87]. Chattering alarms often occur on noisy process variables. Here, the measurement(s) repeatedly violate the alarm limits purely due to measurement noise and do not pose any threat to the operation. Redundant alarms refer to ones that repeat the same information as previous alarm. For example, placing two alarms on a pipeline in series will cause the second alarm to be redundant because any process upsets will already be reported by the first alarm. Furthermore, many alarms are just poorly designed and contain extremely low thresholds; thus, activating even when no process upsets are present. Given that thousands of alarms may be triggered at once, the prioritization of different alarms is equally difficult. In this study, alarm reduction methods based on pattern recognition and RL will first be introduced. Then, an alarm prioritization method on RL will be shown.

4.5.3 Alarm Reduction

The first algorithm in the proposed alarm management system is alarm reduction. Here, two methods will be proposed: 1) Alarm sequencing based on pattern recognition; 2) Alarm suppression via RL.

Alarm Sequencing

The first alarm reduction algorithm is called alarm sequencing. The main goal is to create a sequence dictionary comprising of alarms that often occur (temporally) together [89]. For example, if alarms 1 and 2 frequently occur one after another, an alarm sequence, *Sequence 1*, could be generated representing alarms 1 and 2. During an alarm flooding scenario, operators would see sequence 1 rather than alarms 1 and 2. In this simple scenario, the number of alarms appearing in the console would be halved. More significant reduction in alarms can be achieved if 10 or 100 alarms often occur together. The steps for the alarm reduction algorithm during initial set-up and online implementation is shown in Table 4.6.

Table 4.6: Alarm reduction algorithm

Alarm Reduction Algorithm

Alarms from Plant
Receive process alarms (typically as integers)
Alarm Coupling
Initialize alarm combinations list
Group all alarms in pairs
Save grouped alarms in list
Group all alarms in triplets
Save grouped alarms in list
...
Repeat until all alarms are grouped into one
Sequence Identifier
Count amount of times each **sequence** occurred
Sequence Dictionary Generation
Initialize sequence dictionary
If group alarm occurred $\geq n$
Add grouped alarm to sequence dictionary
Return **Sequence Dictionary**

Online Implementation
Receive alarms from process
Alarm Masking
Group previous two alarms
Check sequence dictionary for match
If match
Replace previous two alarms with first alarm in sequence
Group previous three alarms
Check sequence dictionary for match
If match
Replace previous three alarms with with first alarm in sequence
...
...
Repeat until end of sequence dictionary

During online implementation, the alarm sequences are masked with the *first* alarm in the sequence because it typically corresponds to the *root cause* of the entire alarm sequence [89]. Moreover, simply displaying alarm sequences do little to

help operators because it is essentially the same amount of alarms, except displayed in a more condensed format.

Alarm Reduction through Suppression

The second alarm reduction method suppresses low impact alarms altogether. The algorithm starts by using RL to assign credits to different alarms via the optimal value or action-value functions, $v^*(x)$ and $Q^*(x,u)$, respectively. During alarming events, alarms failing to exceed a threshold value will be suppressed and hidden from operators and is given mathematically by:

$$Alarm = \begin{cases} On, & \text{if } v(x) < z \\ Off, & otherwise \end{cases}$$

where z denotes some arbitrary threshold. Intuitively, this algorithm assesses each new alarm through observing the value of the current operating condition, $v(x)$. During critical events where alarms are necessary, $v(x)$ are intrinsically low because the plant would be operating far from ideal; however, during normal operations littered with nuisance alarms, $v(x)$ remains high resulting in suppression of all nuisance alarms. This algorithm also acts as a *multi-dimensional* alarm system. That is, typical alarms occur after a higher or low limit is exceeded and do not consider multi-dimensional effects. By using the value functions of the process, the interaction effects can be additionally captured.

During initial identification of the $v(x)$ or $Q(x,u)$ values, both Markov reward processes (MRP) or Markov decision processes (MDP) can be used. MRPs are used if only the alarm suppression algorithm is of interest and no actions are involved. MDPs should be used if the plant managers also want to understand the optimal actions during different alarm events. When using $Q(x,u)$, alarms are assigned the average $Q(x_i,u)$ for x_i (i.e., average of $Q(x,u)$ across all possible actions).

Because Q-values may have significantly different magnitudes depending on the

reward function design, the Q-values are first normalized using:

$$Q_{i,norm} = \frac{Q_i - Q_{ave}}{\sigma} \tag{4.11}$$

where σ denotes the standard deviation of the Q-values. In tabular cases, the mean and standard deviation of the Q-values can be found trivially because all Q-values are shown in the Q-table; however, deep RL approaches may face challenges. One recommended solution would be to discretize the system and calculate the corresponding Q-values for each set of states and actions. For example, suppose there exists a SISO system with $x \in [0, 5]$ and $u \in [5, 10]$. The states and actions can be discretized as: $x = [0, 1, 2, 3, 4, 5]; u = [5, 6, 7, 8, 9, 10]$ and the average Q-values can be calculated as:

$$Q_{ave} = \frac{1}{p \times q} \sum_{i,j=0}^{p,q} Q(x_i, u_j|\theta) \tag{4.12}$$

where x_i and u_j are the i^{th} and j^{th} state and action, respectively. Moreover, the standard deviation can be calculated as:

$$\sigma = \sqrt{\frac{\sum (Q_i - Q_{ave})^2}{p \times q}} \tag{4.13}$$

By normalizing the Q-values, the alarm threshold, z, across multiple units within a plant will share similar magnitudes and can be tuned easier. Tuning of z is dependent on the risk tolerance of the plant manager. For a high amount of alarm suppression, z can be tuned exceptionally low. The steps to implementing this alarm reduction algorithm is as follows:

1. Initialize the process and initialize the RL agent using any RL algorithm *tabula rasa*.

2. Allow RL to learn the value or action-values of the process through any traditional way (alarms are not required).

3. After training, find the mean and standard deviations of the Q-values through Equations 4.12 and 4.13.

4. Online implementation: monitor alarms. Any new alarm will be assigned the value or action-value at the time of their occurrence. For alarm sequences, the assigned value is an exponentially weighted moving average (given in Equation 3.1) of the alarm values inside the sequence. An EWMA was used because newer alarms matter more than old.

4.5.4 Alarm Prioritization

During alarm flooding events, alarm reduction algorithms may reduce the amount of alarms to less than 5% of total alarms. However, it does not solve the root cause of the problem. Also, 5% of thousands of alarms is still far from useful and too much for a few operators to handle; therefore, a second algorithm was designed to sort the active alarms by a score. Lower scores denote more dangerous alarms and will be placed on the top of the alarm log. Through this algorithm, the operators are equipped with knowledge regarding the priorities of different alarms and can work to resolve the safety critical ones first.

The alarm prioritization algorithm is shown in Table 4.7. On a high level, each alarm is assigned a value based on the current condition of the plant. Alarms with higher values denote normal plant behaviour and are seen as unimportant. Contrarily, alarms with low values correspond to poor plant performance; thus, alarms associated with low values may be critical.

Table 4.7: Alarm prioritization algorithm.

Alarm Prioritization Algorithm
Alarms from Plant
Receive process alarms (typically as integers)
Communication establishment with RL controller
Return value or average action-value in the given state
If new alarm is part of an existing sequence
Calculate the EWMA Q-value given previous values
Assign Q-value to alarm sequence
Else
Assign value or average action-value to the event
Alarm Sorting
While value of new alarm is higher than alarm below
Move sequence down alarm log
Return sorted alarm log

Tables 4.8 and 4.9 shows a comparison between a traditional alarm log compared to the proposed algorithm. Comparing the alarm logs, it can be seen that the SMART alarm system was able to vastly reduce the amount of alarms, while putting the most important alarms on the top of the list. Additionally, the top alarm is comprised of a sequence of alarms. The first alarm in the sequence (HH Vessel 2) is shown because it is assumed to be the root cause. Other alarms were not shown because their corresponding values are higher than the filtering threshold, z.

Table 4.8: State-of-the-art industrial alarm system. L, H, LL, and HH corresponds to low, high, low low and high high levels.

Events	Status	Date	Analysis	Value	Limit
1	Warning	March 22	L Vessel 5	1.05	1.10
2	Warning	March 22	H Tank 2	61.2	60.0
3	Warning	March 23	L Vessel 3	0.96	1.15
4	Alarm	March 23	HH Tank 1	1.51	1.10
5	Warning	March 23	L Vessel 2	32.7	33.5
6	Alarm	March 23	HH Tank 3	40.2	33.5

Table 4.9: SMART alarm system.

Date	Status	Equipment	RL Value
March 22	Alarm - Sequence	HH Tank 1	2.5
March 23	Warning	H Tank 2	9.5
March 23	Warning	L Vessel 2	10.5

4.5.5 Simulation Results

The algorithms proposed above were simulated on the WWTP. During training, the system was formulated as a MDP and the agent was trained for 10,000 episodes where each episode comprised of a 14 day period of dry weather. Intuitively, the agent was trained for approximately 383.6 years in simulation time (2 hours physical time). After training, the simulation was reset and the storm weather data was used. The storm weather data was used because it is guaranteed to create many alarms in the system. In this study, warnings are triggered when values exceed 75% of the constraints shown in Equation 4.10. Moreover, alarms are triggered if the constraints are violated. To replicate a redundant alarming scenario, S_{nh} alarms are placed on each stage of the separator; therefore, after one S_{nh} alarm triggers, 9 additional alarms will follow due to redundancy. Moreover, alarms are placed on the overall system to measure the other constraints as well. To activate the alarm management system, the alarm sequence dictionary must first be built through pattern recognition.

The alarm sequence dictionary was first built by running the simulation once to identify common alarm sequences. Alarm sequences were built based on the same alarm sequence happening more than 4 times. Additionally, any alarms above $z = 0.5$ was filtered from the alarm log. After z and the alarm sequence dictionary were specified, the alarm management system was ready for use. Two simulations were ran: one without the alarm management system and one with the alarm management system.

Without the system, there were 185 alarms generated within the process with

no prioritization. With the system, only 33 alarms were generated, resulting in a 82% reduction in total alarms. Furthermore, the alarms were organized based on their priority.

The algorithms presented above are exploratory in nature and demonstrate the potential of RL in alarm management and root cause analysis problems. Here, three algorithms are presented to reduce and prioritize alarms with the assistance of RL. The presented algorithms have shown potential in a simple simulated example; however, have not been applied to large scale systems with thousands of alarms. One identifiable flaw with the sequencing technique is its space complexity. In large industrial settings, there could exist millions of different sequences, rendering the method computational infeasible. To implement the alarm sequencing portion of this project, more research will be required.

Chapter 5

Machine Learning for Control Applications

Advanced process control and optimal control have traditionally used mathematical programming based trajectory optimization methods [10], [33], [62], [63]. The effectiveness of these methods in addressing multi-stage optimal control problems has been widely demonstrated; however, industrial scale application of such methods in stochastic multiple-input multiple-output (MIMO) systems are still limited due to design and computation complications [90]. For example, accurate model identification of complex MIMO non-linear systems is challenging. Even if a model were to exist, the computational cost for the non-linear program could be infeasible for online applications. Furthermore, the optimized trajectory to systems containing uncertainty uses stochastic programming with only a finite number of uncertainty scenarios and uncertainty information is assumed to be known. In practice, such information is typically unknown, non-stationary and are uncertain themselves [91]. Moreover, the prediction and control horizon of the trajectory optimization for large MIMO systems are generally truncated to ensure feasible computation time. Though, the identified optimal trajectory for short horizons is typically local optimal solutions [10]. Lastly, mathematical programming (MP) methods require accurate dynamical system models (although no models are perfect in real life); intuitively

bottlenecking the optimality of the solution, a scenario similar to supervised learning.

Comparatively, RL online computational time is significantly shorter even for long control horizons or large-scale MIMO systems because the optimal solutions are pre-computed and stored offline, a concept similar to explicit model predictive control (MPC) where parametric programming is used [60]. Furthermore, RL finds the optimal policy through meaningful interactions with the environment. After each interaction, values are assigned/updated for the visited state. The value functions are stored for future decision making. Through this identification process, the value functions implicitly contain the uncertainty information of all $x \in \mathcal{X}$. From these unique features of RL in control applications, it is a natural curiosity to explore its potential in the process control industry.

The contributions made in this chapter are as follows:

1. Introduction of a simple, cost-effective, and explainable RL algorithm for process control. The method is also continuous and non-linear.

2. Compared RL and deep RL to traditional optimal control methods such as MPC.

3. Applied RL to an industrial grade waste water treatment plant (WWTP) for optimal control. Results were compared to MPC, economic MPC, and distributed MPC frameworks.

4. Applied RL for fault prediction and fault tolerant control applications.

5.1 An direct adaptive optimal control method

On a high level, optimal control methods extremize the functional equation of a system through MP methods. In literature, it was found that optimal control methods are less tractable both computationally and analytically compared to set-point tracking or regulations methods (due to non-convex optimization among fac-

tors). Consequently, adaptive optimal control methods have received less academic attention, with most existing studies focused on indirect methods like model re-parameterization [92]. In [92], RL was shown to be an effective *direct* adaptive optimal control method as it adapts its control policy directly. Direct adaptive optimal control methods are especially useful for systems where accurate models are not identifiable and/or available. In such scenarios, indirect methods struggle because the accuracy of the model is paramount for successful control. On the other hand, direct methods can update the control policy directly through interactions with the system, eventually arriving at the optimal policy. In [93], the authors showcased RL's directly adaptive nature by applying an agent onto the control system of a data-center cooling application. In such systems, accurate models are difficult to identify due to complex non-linear relationships between thousands of variables. However, the agent here was able to adapt to the optimal policy directly after sufficient online interactions which ultimately resulted in 22% reduced power consumption compared to model-based approaches. Likewise, [94] applied RL onto power systems with ever-changing load fluctuations. The agent was still able to adapt its policy after sufficient online interactions. The authors in [95] demonstrated deep RL's direct adaptive nature through its application onto wireless networks, ultimately resulting in superior control compared to all previous methods.

5.2 Controlling a VFD using Q-learning

A detailed quantitative example is provided in this section to serve as a gentle introduction of RL's applicability in process control systems [1]. Here, the *off-policy* tabular Q-learning algorithm with upper confidence bound (learning acceleration heuristics) was used for output pressure tracking of an industrial variable frequency drive (VFD) pump.

[1]For further intuition, the supplementary code for all results generated in this section are located at: https://github.com/RuiNian7319/Research/tree/master/2.RL_Codes/Mechatronix

5.2.1 System Description

The industrial VFD system is built by Turbine Technologies and is called the FLUIDMechatronix. On the system, there exist thousands of different tags measuring various states. For this control example, the output pressure, P_{out}, and pump RPM will be used. From the FLUIDMechatronix manual, the safe operating ranges of the pressure and pump RPM are:

$$0\ kPa \leq P \leq 45\ kPa$$

$$0\ Hz \leq RPM \leq 60\ Hz$$

In terms of system representation, a FOMDP will be used because all system measurements are available and the system dynamics are fast. Initially, the system starts in:

$$P_0 = 41\ kPa \tag{5.1}$$

$$u_0 = 60\ RPM \tag{5.2}$$

Figure 5.1: The FLUIDMechatronix experiment from Turbine Technologies [96].

Implementing RL for the control of an industrial process typically involves four steps: i) Model identification; ii) agent design; iii) initial training; iv) online cali-

bration. The details of each step are shown in 5.2.

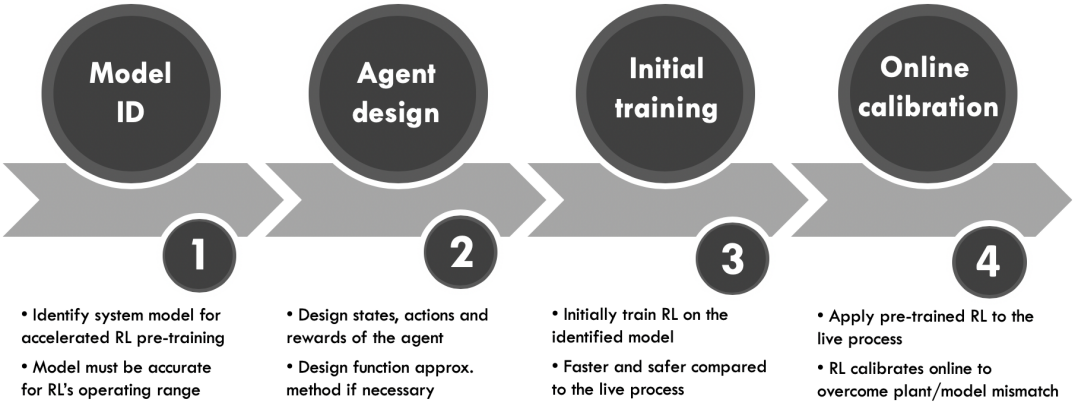


Figure 5.2: General procedure for implementing industrial reinforcement learning.

5.2.2 Step 1: Model ID

One major drawback of RL is its unreasonable data efficiency. In fact, it *may* take thousands of interactions before anything meaningful is first learned. As such, implementing RL to learn online is time-infeasible because decades may pass before the optimal policy is identified. To overcome this flaw, a *representative* simulation model can first be constructed to pre-train the agent offline. After adequate performance is observed in simulation, the agent can be implemented online. Initially, the agent will calibrate its policy to the live process to overcome any model-plant mismatches. The time required for calibration is heavily dependent on the accuracy of the simulation model. For perfectly representative models, such as video games, a calibration time is not required. Afterwards, optimal control can commence.

For model identification, pseudo-random input signals were used to provide excitation to the VFD for input-out data generation. The data collection process was terminated after 18,000 input-output signals were obtained. Then, a quadratic model was identified using least squares. The identified model is given as:

$$P_{out} = 0.012 \cdot RPM^2 + 0.024 \cdot RPM - 2.073 \tag{5.3}$$

The mean squared error (MSE) of the identified model was 0.056 and the fitted

model compared to the experimental data is shown in Figure 5.3.

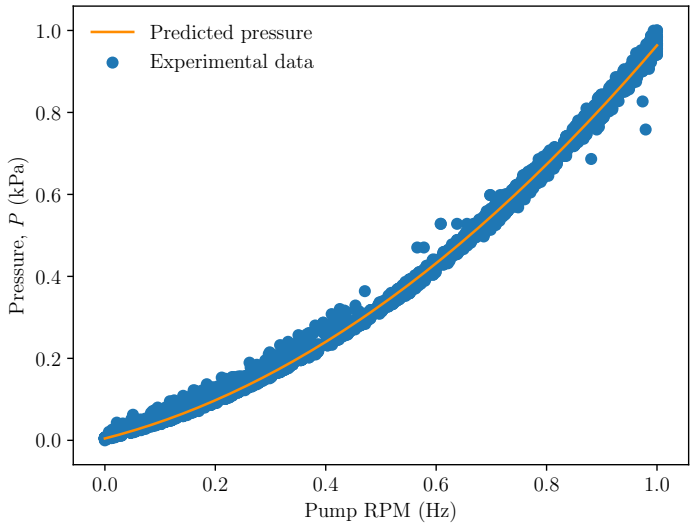


Figure 5.3: Performance of the identified system model on a test data set.

5.2.3 Step 2: Agent design

The overall RL paradigm for this example is shown in Figure 5.4. Intuitively, the agent tracks a set-point for the output pressure by manipulating the pump RPM. To allow for the tracking of a variable set-point, the state of the agent is the current tracking error:

$$\varepsilon = P_t - P_{t,sp} \tag{5.4}$$

and the action is the *change* in pump RPM Δu. Here, P_t is the pressure at time t and sp denotes the set-point. This *velocity* based implementation is a requirement for tracking multiple set-points. If the action was the pump RPM instead (and not Δu), then the agent would fail to track multiple set-points since it simply maps different tracking errors to one pump RPM that corresponds to *one* set-point. For example, suppose the current set-point is 10 kPa and $P_t = 0$ resulting in a -10 tracking error. Here, suppose the optimal RPM is $u = 20$. After implementing $u = 20$, the system reaches steady state and achieves a tracking error of 0. After some time, the machine operator may then decide to change the set-point to 20 kPa. Now again, the tracking error is -10; however, the RPM from the agent would still

be $u = 20$ because the state-action is a 1 to 1 mapping; thus, unable to track any changes in set-points.

The reward function of the agent is given by:

$$r(x, u) = max(-\varepsilon^2 - \Delta u, -200) \tag{5.5}$$

where Δu is the change in input to discourage the agent from making unnecessary actions. Additionally, the reward is clipped to -200 for convergence properties and to avoid numerical issues as explained in the reward clipping section. The upper reward limit is not clipped because the function is naturally capped at 0. Here, the agent evaluates every five seconds to guarantee that steady state has been reached before consecutive actions are made. Five-second was chosen because it was identified to be the longest transition time required. Moreover, the decision making would not be Markovian (i.e., observed states are not independent of the past because the states do not provide the transition information to the agent) if the agent evaluates during the transition period. In such scenarios, the agent will fail to learn anything meaningful.

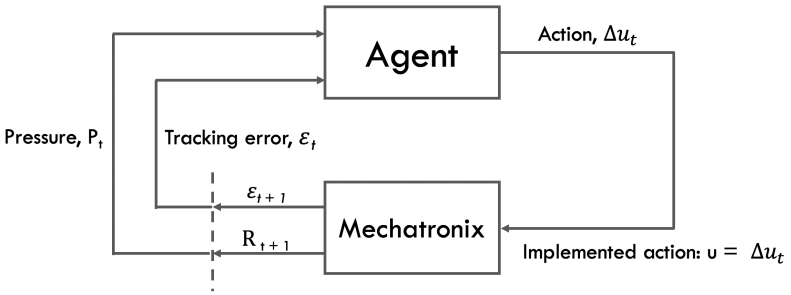


Figure 5.4: The RL set-up for the FLUIDMechatronix experiment.

All hyper parameters of the agent are shown in Table 5.1. The states and actions of the agent are discretized as:

$$x = [-20, -19, ..., 20]_{1\times 41} \tag{5.6}$$

$$u = [-10, -9, ..., 10]_{1\times 21} \tag{5.7}$$

totalling 861 different action-values to identify. Furthermore, the Q-matrix storing all the action-value functions is shown in Figure 5.5. The agent is initiated with all action-values as 0, a condition known as *tabula rasa*.

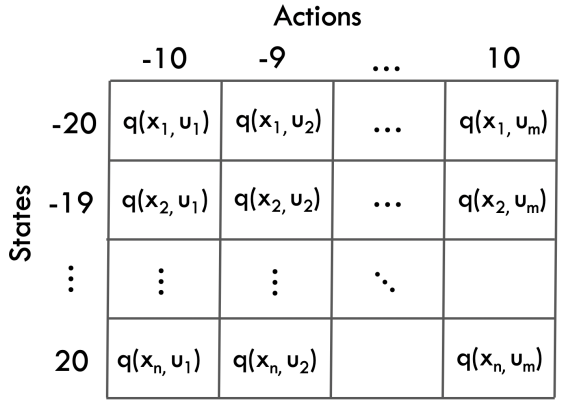


Figure 5.5: Q-matrix of the Mechatronix system.

The states and actions on the axis of the Q-matrix correspond to ε and Δu, respectively. The discount factor, γ, was 0.9. Altogether, 2,000,000 time steps were used to train the agent (corresponding to 115.7 days of continuous operating experience). After every 400^{th} time step, the agent's state and action was reset back to Equations 5.1 and 5.2 to prevent extreme controller saturation.

The agent uses a equiprobable random exploratory policy ($\epsilon = 1$) to conduct initial exploration. Throughout training, ϵ is slowly and linearly decayed until $\epsilon = 0.1$ by the $500{,}000^{th}$ update. Likewise, the learning rate is initiated at 0.7 and also linearly decays until 0.001.

Table 5.1: Summary of the agent's hyper parameters in the Mechatronix experiment.

Hyper Parameter	Value
States, x	$\varepsilon = [-20, -19, ..., 20]_{1\times 41}$
Actions, u	$\Delta u = [-10, -9, ..., 10]_{1\times 21}$
Reward, r	$\max(-(\varepsilon^2 + \Delta u)$, -200)
Learning rate, α	[0.001, 0.7]
Discount factor, γ	0.9
Exploratory factor, ϵ	[0.1, 1]
Evaluation time	5 seconds
System representation	FOMDP

5.2.4 Step 3: Initial training

The agent behaves as follows: the agent observes some initial tracking error ε_t and performs a random action Δu_t with accordance to its behaviour policy (initially equiprobable random). Next, the pump RPM corresponding to $u_t = u_{t-1} + \Delta u_t$ is sent to Mechatronix. After five seconds, the agent receives reward R_{t+1} and then observes new tracking error, ε_{t+1}. Using the tuple $(x_t, u_t, r_{t+1}, x_{t+1})$, the agent updates its current knowledge via Equation 2.38. This step is repeated many times until the optimal policy, π^*, is identified. A numerical walk-through of the calculations is shown below:

> Suppose another simpler agent was constructed for this system. For this agent, the system was discretized into five states and three actions:
>
> $$x = [-21, -10, 0, 10, 21]_{1\times 5}$$
>
> $$u = [-1, 0, 1]_{1\times 3}$$
>
> Consequently, the Q-matrix was initialized as:

$$Q(x,u) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

where the rows and columns correspond to the different states and actions, respectively. The system's set-point was initially at 30 kPa. The agent was initiated at steady state with 15 kPa and 37 RPM, resulting in $\varepsilon = -15$. At $t = 0$, the agent receives the error and rounds it to the nearest discretized value, $x = -10$. Given this state, the agent uses the Q-table and picks the action that corresponds to the highest Q value (note if a equiprobable random policy was initially followed, a random action would be selected instead):

$$Q(-10,u) = [0,0,0]$$

where the three values correspond to the predicted action-values for selecting actions $\Delta u = -1, 0, -1$, respectively. Since the agent is inexperienced and has not been provided with prior information about the system, it thinks that all three actions are indifferent; therefore, the agent will pick an arbitrarily action to learn more about the system. Moreover, during scenarios where $Q^{max} = Q_1 = Q_2 = ... = Q_n$, ties *must* be broken arbitrarily to avoid biasing one action over all others.

Assuming that $u = -1$ was picked, the system will transition to 13.8 kPa. After five seconds, the new observed tracking error would be -16.2. Clearly, this was a sub-optimal action; if the pressure was already lower than the set-point, it would be intuitive to increase pump RPM instead. However, a *tabula rasa* agent is not aware of such a phenomenon,

humans only know this through prior experience. Here, the agent would also receive reward:

$$max(-16.2^2 - 1, -200) = -200$$

and be in new state $x_1 = -21$. From this interaction, the agent would then update the Q-matrix using Equation 2.38:

$$Q(-10, -1) \leftarrow Q(-10, -1) + 0.7[-200 + \gamma Q(-21, 0) - Q(-10, -1)]$$

$$Q(-10, -1) \leftarrow 0 + 0.7[-200 + 0.9 \cdot 0 - 0]$$

$$Q(-10, -1) \leftarrow -140$$

and the updated Q-matrix would be given as:

$$Q(x, u) = \begin{pmatrix} 0 & 0 & 0 \\ -140 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

In this case, all three u's for $Q(x_{t+1}, u_{t+1})$ are also reward maximizing; therefore, the ties here must also be broken randomly to avoid unnecessary bias. Suppose the system was reset, initiating at $x_0 = -10$. This time, the Q-matrix provides:

$$Q(-10, u) = [-140, 0, 0]$$

telling the agent that $\Delta u = -1$ is a sub-optimal compared to $\Delta u = 0$ or 1. For a reward maximizing agent, either $\Delta u = 0$ or 1 would be picked instead.

After many interactions with the system, the Q-matrix now becomes:

$$Q(x,u) = \begin{pmatrix} -152 & -133 & -120 \\ -149 & -121 & -99 \\ -31 & -22 & -33 \\ -102 & -142 & -162 \\ -152 & -162 & -177 \end{pmatrix}$$

Now, the agent has vastly more knowledge about the system and can begin acting optimally. After resetting the agent back to $x_0 = -10$, the decision making of the agent is now deterministic. The corresponding action-values given $x_0 = -10$ are:

$$Q(-10,u) = [-149, -121, -99]$$

Here, the agent would pick $\Delta u = 1$ corresponding to $Q(-10,3) = -99$ (greedy action) and the system would transition to $P_1 = 16\ kPa$. Although the error is still closest to -10 (set-point is 30 kPa), the reward obtained is much better compared to actions -1 or 0. The new update step is given as:

$$Q(-10,1) \leftarrow Q(-10,1) + 0.001[-197 + \gamma Q(-10,1) - Q(-10,1)]$$

$$Q(-10,1) \leftarrow -99 + 0.001[-197 + 0.9 \cdot -99 + 99]$$

$$Q(-10,1) \leftarrow -99 + 0.001[-187.1]$$

$$Q(-10,1) \leftarrow -99.19$$

Here, α is much lower compared to previously due to the continuous decay throughout the training process. The currently TD error is -187.1,

still quite a high value. Eventually, all TD errors will approach near-zero and the agent's policy will become optimal.

The reward obtained across the 2 million training steps is shown in Figure 5.6. Ultimately, the reward was unable to become zero because the lower bound of ϵ was set to 0.1, forcing exploratory moves even when the agent had the capability of acting optimally. During training, the set-point was drawn from a Gaussian distribution $N(30, 5)$.

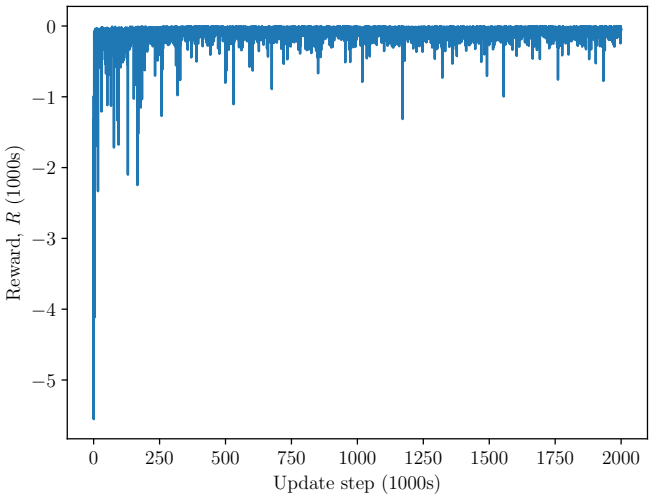
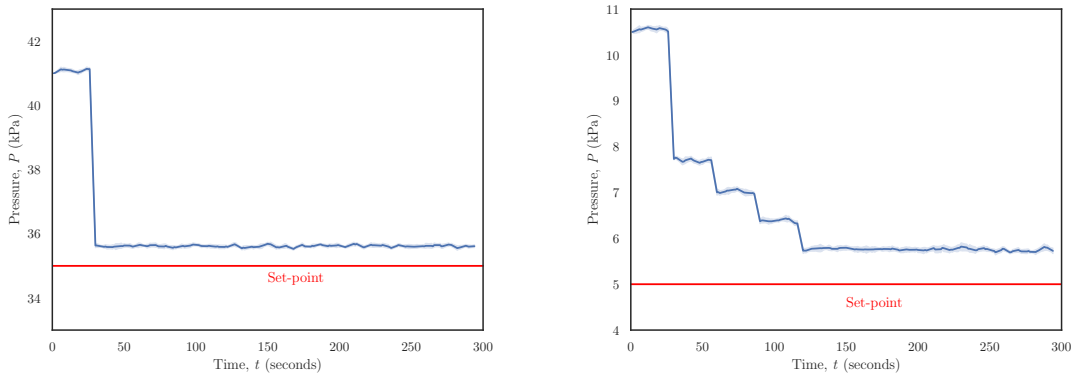


Figure 5.6: Loss curve of the agent during training.

5.2.5 Step 4: Online calibration

The agent was then applied onto the real process to track pressure set-points of 35 and 5. The output pressure trajectory of the Mechatronix is shown in 5.7a and 5.7b. Performance-wise, the MSE was 14.2 and 15.5 for set-points 35 and 5, respectively. To ensure a fair comparison, both cases started with initial pressures approximately 5.5 kPa above the desired set-point. In this simple set-up, the agent behaves much like a PID where the RL maps tracking errors to changes in input and is linear in nature. Unfortunately, such a set-up only works well locally for non-linear systems. Moreover, the performance decreases significantly as the agent ponders away from the linear region, as shown in Figure 5.7b. In this experiment,

the agent's performance is significantly better when tracking $P = 35$ because the training set-points were heavily biased towards $P = 35$. From Figure 5.3, it can be seen quite obviously that the controller gain changes significantly at lower pressures, resulting in the optimal policy for the higher pressure range being completely sub-optimal at lower pressures. Furthermore, the large off-set seen in these trajectories is caused by the discretization error; there exists no action $\Delta u \in \mathcal{U}$ that can obtain exactly $P = 35$ *or* $5\ kPa$. To overcome this, one option is to discretize the action space more finely, but this will unavoidably increase the training time and space complexity required by the agent (perhaps by a massive margin). A simpler way will be introduced in the latter half of this example.



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Figure 5.7: Pressure trajectory of the Mechatronix experiment. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

5.2.6 Extension to Non-linear Systems

A simple, cost effective way for the agent's capabilities to extend to non-linear systems is to model the system using a linear parameter-varying model as shown in Figure 5.8. This way, each sub-piece of the model is linear, allowing even linear control laws to be optimal. To create a Markovian setting, the agent will receive

this information through a *second state* given by:

$$x^{(2)} = \begin{cases} 1, & \text{if } P \leq 10 \\ 2, & \text{if } 10 < P \leq 20 \\ 3, & \text{if } 20 < P \leq 30 \\ 4, & \text{if } 30 < P \leq 50 \\ 5, & P > 50 \end{cases}$$

The new state space for the agent is given by:

$$x = [(-20, 1), (-20, 2), (-20, 3), ..., (-19, 1), ..., (20, 5)]_{1\times 205} \tag{5.8}$$

where the first value denotes the error and the second value correspond to the region the agent is currently in. Here, the Q-matrix will be initialized as $0_{(41\cdot 5)\times 3}$ to accommodate for the second state. Intuitively, the agent now observes both the magnitude and the context of the incurred tracking error; intrinsically, allowing the agent to change its policy depending on the region it is currently in.

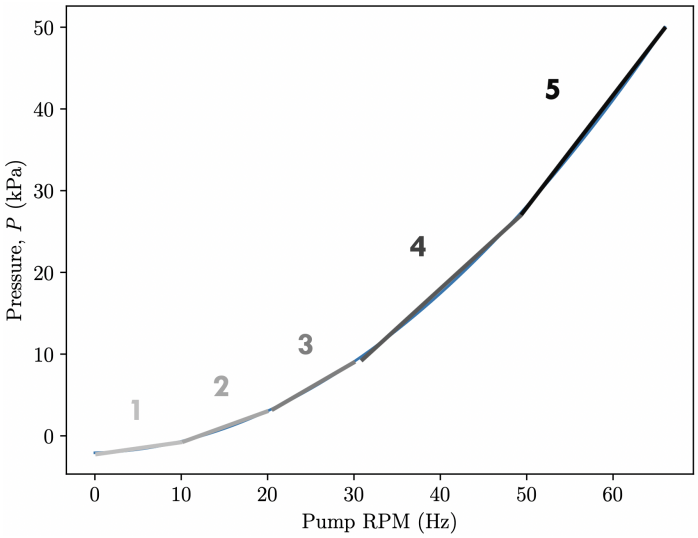
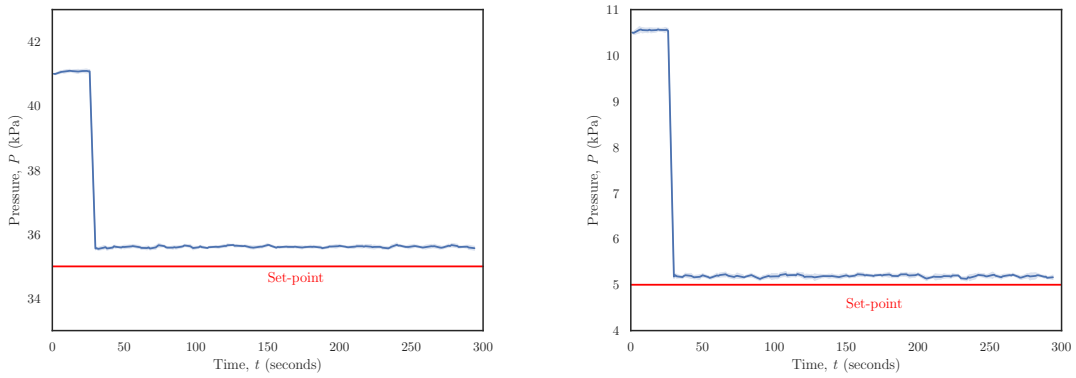


Figure 5.8: Approximating the non-linear Mechatronix system.

The new agent was implemented onto Mechatronix after 2,000,000 time steps of training. The new agent's output pressure trajectories for tracking $P = 35$ and 5

are shown in Figures 5.9a and 5.9b. Performance-wise, the agent achieved MSEs of 14.2 and 12.5 for the higher and lower set-point; a massive improvement for the lower set-point. Nevertheless, the offset still exists.



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Figure 5.9: Pressure trajectory using the non-linear agent. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

5.2.7 Extension to continuous states and actions

Because the non-linear system was approximated using a LPV model, the control law for the system should be always linear. Because of this, linear interpolation can be used to find the optimal control action using [97]:

$$u = u_{low} + (x - x_{low})\frac{u_{high} - u_{low}}{x_{high} - x_{low}} \tag{5.9}$$

where x is the actual tracking error; typically, the exact value of x is not included in the discretized state space $\mathcal{X}$. Instead, x is typically between x_{high} and x_{low}, where x_{high} and x_{low} correspond to the state that is higher and lower than x, respectively. For example, if the discretized state space is given as $x = [0, -5, -10]$ and the current state is -3, x_{high} and x_{low} would be 0 and -5, respectively. Similarily, u_{high} and u_{low} are the greedy actions for x_{high} and x_{low}, respectively. For example, given the action space $u = [-5, 0, 5]$ and $Q-$matrix:

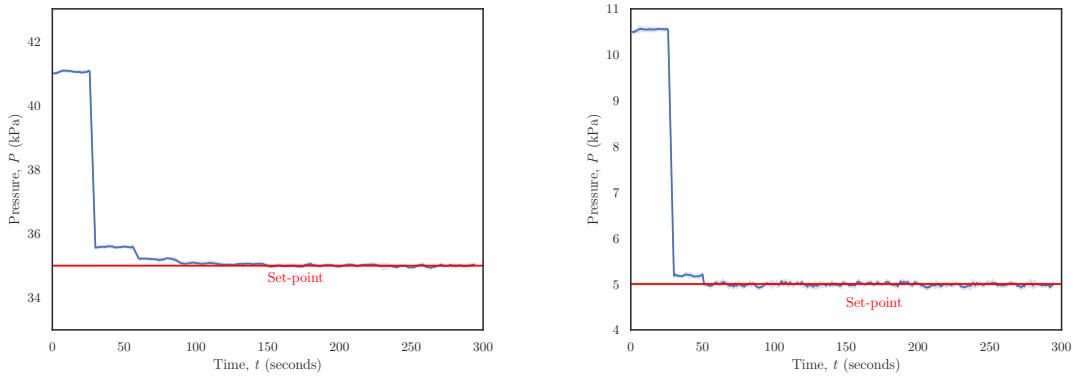
$$Q(x,u) = \begin{pmatrix} -5 & 2 & 1 \\ 4 & 1 & -2 \\ -2 & 0 & 3 \end{pmatrix},$$

u_{high} and u_{low} are 0 and -5 (actions corresponding to the index of the highest Q-value), respectively. Moreover, the optimal action for $x = -3$ would be:

$$u = -5 + (-3+5)\frac{0+5}{0+5}$$

$$u = -3$$

With the addition of interpolation action selection, the 2-state RL agent (without re-training the agent) achieved pressure trajectories shown in Figures 5.10a and 5.10b with MSEs of 13.6 and 11.7, respectively. Additionally, it can be seen that the off-set is completely eliminated.



(a) Tracking $P_{sp} = 35\ kPa$ (b) Tracking $P_{sp} = 5\ kPa$

Figure 5.10: Pressure trajectory of the non-linear agent using interpolation action selection. Solid line represents the average of 10 runs to ensure reproducability. Shaded area correspond to one standard deviation.

5.2.8 A Study on Interpolated RL

As shown in previous examples, a critical flaw regarding tabular RL is its discretized states and actions. In control problems requiring precise actions, the algorithm assumes sufficient discretization resolution is provided. Unfortunately, in-

creasing resolution of the system greatly increases the time required for RL to learn the optimal policy. Contrarily, a system with coarse discretization results in sub-optimal control and large off-sets. One simple, yet effective way to extend RL to continuous space *without dramatically increasing training time* is to interpolate actions (as shown above). Here, an agent with and without interpolated actions will be applied a low resolution SISO system to explore the technique's effectiveness.

The SISO system is given by:

$$\dot{x} = -4x + 2u$$

$$y = x$$

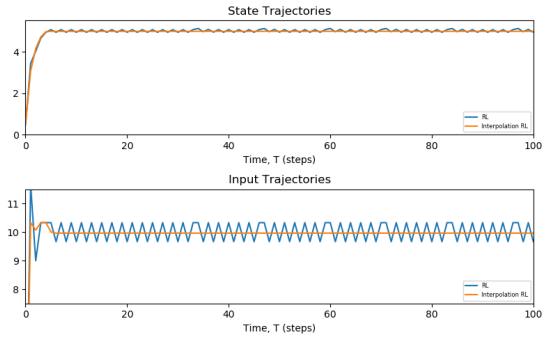
The RL hyper parameters are given in Table 5.2.

Table 5.2: A low resolution RL agent's hyper parameters.

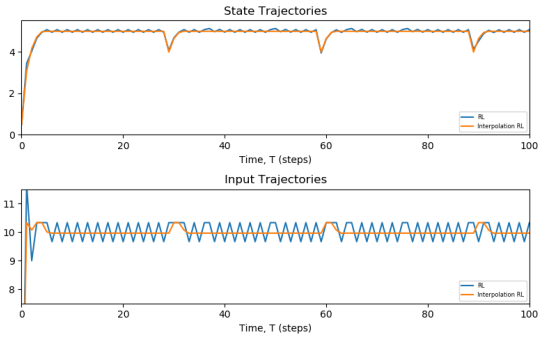
Hyper Parameter	Value
States, x	$\varepsilon = [0, 1, ..., 8]_{1\times 9}$
Actions, u	$\Delta u = [5, 6.4, ..., 16]_{1\times 9}$
Reward, r	$\max(-(\varepsilon^2 + \Delta u)$, -200)
Learning rate, α	[0.001, 0.7]
Discount factor, γ	0.9
Exploratory factor, ϵ	[0.1, 1]
Evaluation time	1 seconds
System representation	FOMDP

Figures 5.11a, 5.11b, 5.12a, and 5.12b compare the agent's performance with and without interpolated actions after experiencing no, low, medium, and large disturbances. The tracking errors of the systems are provided in Table 5.3. From the trajectory figures, it can be seen that the oscillations in the input are completely removed; a significant benefit during implementation due to reduced wear-and-tear on the actuator. The tracking performance also significantly increases and results

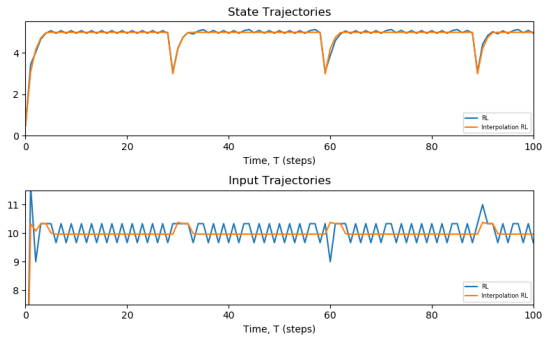
in no oscillations. In terms of performance, the % improvement decreased during larger disturbances due to the disturbance dominating the majority of the error. From Table 5.12b, interpolated RL never performed worse than normal RL, suggesting that there is little downside to implement this method other than the slightly increased computational cost. Note that the algorithm cannot be used for extrapolation; therefore, interpolated actions cannot be conducted for states that surpass the largest discretized state.



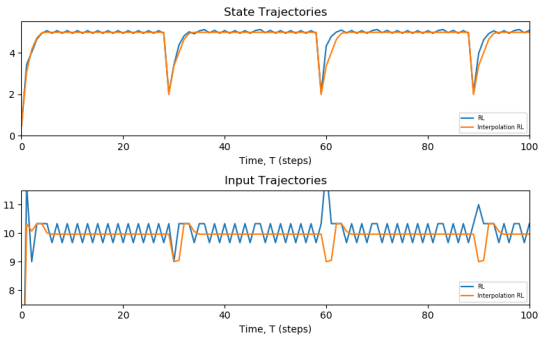
(a) State and input trajectory of the system with no disturbance.



(b) State and input trajectory of the system with small disturbances.



(a) State and input trajectory of the system with medium disturbances.



(b) State and input trajectory of the system with large disturbances.

Table 5.3: Tracking error for the agent with and without interpolated actions.

	Normal RL	**Interpolated RL**	**% Change**
No disturbances	11.3	4.8	58
Small disturbances	14.5	8.6	41
Medium disturbances	25.9	19.7	24
Large disturbances	46.0	56.0	22

5.2.9 Final Remarks

The RL methods introduced in this section along with their respective characteristics are summarized in Table 5.4. Throughout this section, a simple RL agent was implemented onto an industrial VFD system. It was shown that the vanilla algorithm was unable to handle either non-linear systems or achieve off-set free control; therefore, simple, implementable techniques that extended the agent's ability to non-linear systems and for off-set free control were introduced. Each state trajectory in this study was replicated on the live systems 10 times to ensure reproducability; the standard deviation for every algorithm was very narrow, representing highly reproducible results (indirectly, less risk).

Table 5.4: A comparison between RL, MPC in literature, and industrial MPC software.

	Normal Q-learning	**2-state Q-learning**	**2-state interpolated Q-learning**
MSE (High/Low SP)	14.2 & 15.5	14.2 & 12.5	13.6 & 11.7
Offset	Yes	Yes	No
Non-linear	No	Yes	Yes

In the implementation above, the agent only provided the input for the *immediate* future. A concept very similar to MPC where only the next input is used; however, MPC is considered receding horizon control, where an input trajectory for future steps is also calculated. Using this trajectory, MPC is viable for short horizon open-loop control. Comparatively, RL can also conduct receding horizon control. Such RL methods typically employ a model of the system and are called planning methods. In receding horizon RL, the agent still only outputs the immediate control action; however, it then uses the model to identify the next state and its corresponding optimal control action. The cycle continues until the set control horizon is met. Additionally, like MPC, the trajectory is heavily inaccurate for long control horizons.

The example shown here is simple, has a pre-set sampling time and does not consider transition dynamics or unobservable states. For systems containing dynamic transition times and to consider systems dynamics, semi-MDPs must be used. The

semi-MDP variant of Q-learning algorithm is [25]:

$$Q(x,u) \leftarrow Q(x,u) + \alpha \left[\frac{1-e^{-\beta\tau}}{\beta} r(x, x_{t+1}, u) + e^{-\beta\tau} \max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x,u) \right] \tag{5.10}$$

where $r(x_t, x_{t+1}, u)$ is the reward rate and is provided in Equation 5.39. For systems with unmeasurable states, concepts of POMDPs provided in Chapter 1 should be used.

5.3 Optimality Evaluation of Reinforcement Learning

UCB tabular Q-learning suffers from discretized states and actions and the curse of dimensionality. To overcome such issues, deep RL will be explored. More specifically, this section uses the deep deterministic policy gradient (DDPG) algorithm introduced in Chapter 1. DDPG offers several advantages and disadvantages compared to tabular Q-learning. The advantages are that it can handle continuous states and perform continuous actions. Furthermore, the scalability of the algorithm is greatly enhanced because deep function approximators are used to map states to actions, rather than a Q-table. However, these advantages come with some disadvantages. First and foremost, DDPG is a black box approach and employs four deep neural networks to perform function approximation; therefore, explicitly identifying the control policy for DDPG is nearly impossible. Secondly, the function approximations cause small perturbations in control. That is, the control outputs can sometimes contain small jitters (given an ideal output of 1, DDPG may output values between 0.995 - 1.005). Intuitively, this is caused by the generality of the algorithm and can be related to humans. Humans possess perhaps one of the most general intelligence available, yet we are not even capable of drawing a straight line. DDPG follows a similar idea; the algorithm is so general that it sometimes struggle with highly precise actions. For advanced details regarding DDPG, see [47].

The objective of this section is to **explore the optimality** of RL and how closely it can approach the optimal solution (assuming MPC provides the optimal solution when given a perfect process model). Here, four different control strategies will be applied onto simple SISO, SIMO, MISO, and MIMO systems. The four strategies are shown in Table 5.5.

Table 5.5: Different control strategies to be compared.

Control Algorithm	**Reward Function**
MPC	MPC cost function
Tabular Q-learning	MPC cost function
DDPG	MPC cost function
DDPG	Custom RL reward function

From Table 5.5, the MPC cost function is given as:

$$J = \sum_{i=1}^{N} x_i^T Q_{mpc} x_i + \sum_{i=1}^{N} u_i^T R_{mpc} u_i \tag{5.11}$$

where:

$$x_i = x_t - x_{ss}$$

$$u_i = u_t - u_{ss}$$

and the custom RL reward function is:

$$reward = \begin{cases} 15 - (x_t - x_{sp}) \times 15, & \text{if } x_i \leq 1 \\ x_i^2 + u_i^2, & otherwise \end{cases}$$

5.3.1 Single-Input Single-Output System

First, a SISO system will be used to benchmark RL against MPC. The system is given by:

$$\frac{dx}{dt} = -4x + u \tag{5.12}$$

In the s-domain, the system equation is:

$$Y(s) = \frac{1}{s+4} \tag{5.13}$$

The system is stable with one pole at -4. Initially, the system is at steady state with:

$$x_0 = 0.5$$

$$u_0 = 1.0$$

The steady-state state and input, x_{ss} and u_{ss}, are:

$$x_{ss} = 5.0$$

$$u_{ss} = 10.0$$

Here, MPC used a prediction and control horizon of 20, with Q_{mpc} and R_{mpc} as 0.1 and 0.1, respectively. The RL hyper parameters are given in Tables 5.6 and 5.7

Table 5.6: Tabular RL hyper parameters for the SISO system.

Hyper Parameter	**Value**
States, x	$[0, 0.18, ..., 8]_{1\times45}$
Actions, u	$[5, 5.22, ..., 15]_{1\times45}$
Reward, r	Equation 5.11
Learning rate, α	[0.001, 0.7]
Discount factor, γ	0.95
Exploratory factor, ϵ	[0.1, 1]
Evaluation time	1 second
System representation	FOMDP

Table 5.7: DDPG hyper parameters for the SISO system.

Hyper Parameter	**Value**
Actor network size	3 layers: 50, 40, 40 neurons
Actor learning rate	0.0001
Critic network size	3 layers: 40, 30, 30 neurons
Critic learning rate	0.001
Reward, r	Equation 5.11 or Equation 5.3
Discount factor, γ	0.95
Evaluation time	1 second
System representation	FOMDP

Figure 5.13 shows the input and state trajectories of the four control algorithms. The total cost of each trajectory (all calculated using Equation 5.11) is shown in Table 5.8. Ultimately, RL actually *surpassed* the performance of MPC in this case, even when MPC is equipped with a perfect process model. Comparing the trajectories, it can be seen that RL started with aggressive inputs, but reduced the magnitude thereafter. On the other hand, MPC started with a smaller initial inputs, and gradually increased it along the trajectory. Deep RL had the poorest cost performance; it achieved the state set-point, but incurred large losses with its aggressive inputs. In theory, MPCs using a perfect process model will output the optimal solution. In this case, one source of error could be the discount factor in RL that is not present in MPC. Typically, a discount factor of 0.95 denotes a 20 prediction horizon [39]; however, these are not exactly equivalent. Nevertheless, this small scale study demonstrates that RL can actually surpass MPC even when the cost functions are *nearly* identical.

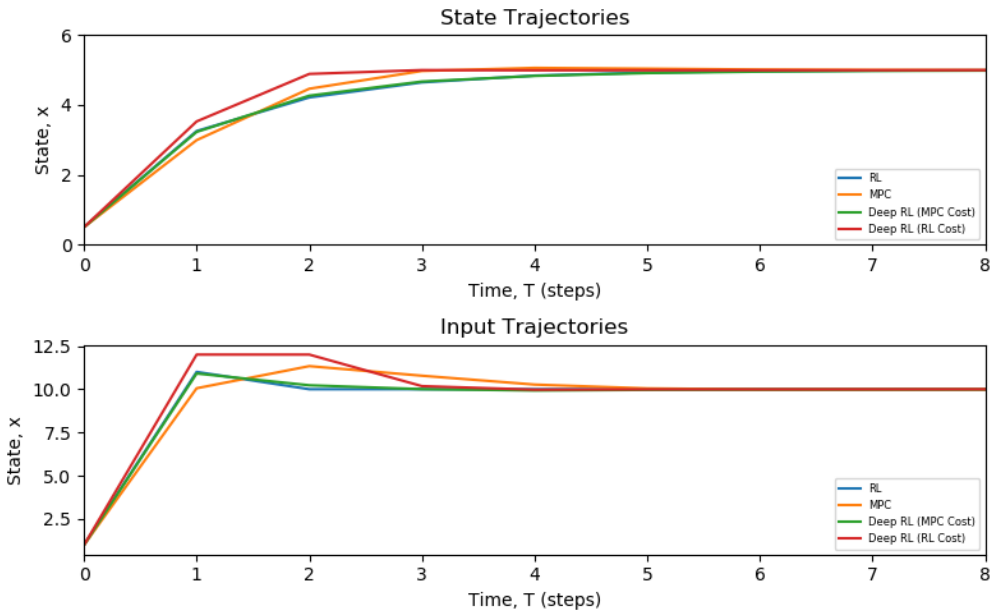


Figure 5.13: Input and state trajectories of the four control strategies on the SISO system.

Table 5.8: Controller cost for the input and state trajectories on the SISO system.

MPC	**Tabular RL**	**Deep RL with MPC cost**	**Deep RL with RL cost**
5.56	4.55	4.27	6.19

5.3.2 Multiple-Input Multiple-Output System

The above study was repeated for a simple MIMO system described by:

$$\frac{dx_1}{dt} = -3x_1 - 2x_2 + 4u_1 \tag{5.14}$$

$$\frac{dx_2}{dt} = -3x_2 + 2u_2 \tag{5.15}$$

In the s-domain, the system equations are:

$$Y_1(s) = \frac{4}{(s+3)^2} \tag{5.16}$$

$$Y_2(s) = \frac{2}{(s+3)^2} \tag{5.17}$$

The system is critically damped and stable (i.e., two identical poles at -3). Initially, the system is at steady state with:

$$x_0 = \begin{bmatrix} 1.3 \\ 4.0 \end{bmatrix} \; u_0 = \begin{bmatrix} 3.0 \\ 6.0 \end{bmatrix} \tag{5.18}$$

The steady-state states and inputs, x_{ss} and u_{ss} are:

$$x_{ss} = \begin{bmatrix} 3.6 \\ 4.7 \end{bmatrix}$$

$$u_{ss} = \begin{bmatrix} 5.0 \\ 7.0 \end{bmatrix}$$

The MPC for this system also has a prediction and control horizon of 20. The Q and R matrices are given by:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The RL hyper parameters for the MIMO system is given in Tables 5.9 and 5.10.

Table 5.9: Tabular RL hyper parameters for the MIMO system.

Hyper Parameter	**Value**
States, x	$x_1 = [0, 0.5, ..., 6]_{1\times 13}$
	$x_2 = [2, 2.5, ..., 6]_{1\times 9}$
Actions, u	$u_1 = [1, 2, ..., 7]_{1\times 7}$
	$u_2 = [4, 5, ..., 9]_{1\times 6}$
Reward, r	Equation 5.11
Learning rate, α	[0.001, 0.5]
Discount factor, γ	0.95
Exploratory factor, ϵ	[0.1, 1]
Evaluation time	1 second
System representation	FOMDP

Table 5.10: DDPG hyper parameters for the MIMO system (identical to the SISO system).

Hyper Parameter	**Value**
Actor network size	3 layers: 50, 40, 40 neurons
Actor learning rate	0.0001
Critic network size	3 layers: 40, 30, 30 neurons
Critic learning rate	0.001
Reward, r	Equation 5.11 or Equation 5.3
Discount factor, γ	0.95
Evaluation time	1 second
System representation	FOMDP

Figure 5.14 shows the input and state trajectories of the MIMO system using the four control methods. The total cost of each trajectory (again calculated using Equation 5.11) is shown in Table 5.11. Here, MPC performed the best while the two deep RL methods performed the worst. Furthermore, tabular RL's performance was nearly identical to MPC. Comparing the trajectories, the deep RL tends to favour

x_2 and made large actions in u_2 while MPC and tabular RL instead focused on x_1. It is difficult to identify exactly why deep RL favored x_2. Perhaps it was because x_2 was only a function of itself and u_2, but the exact reason is unknown.

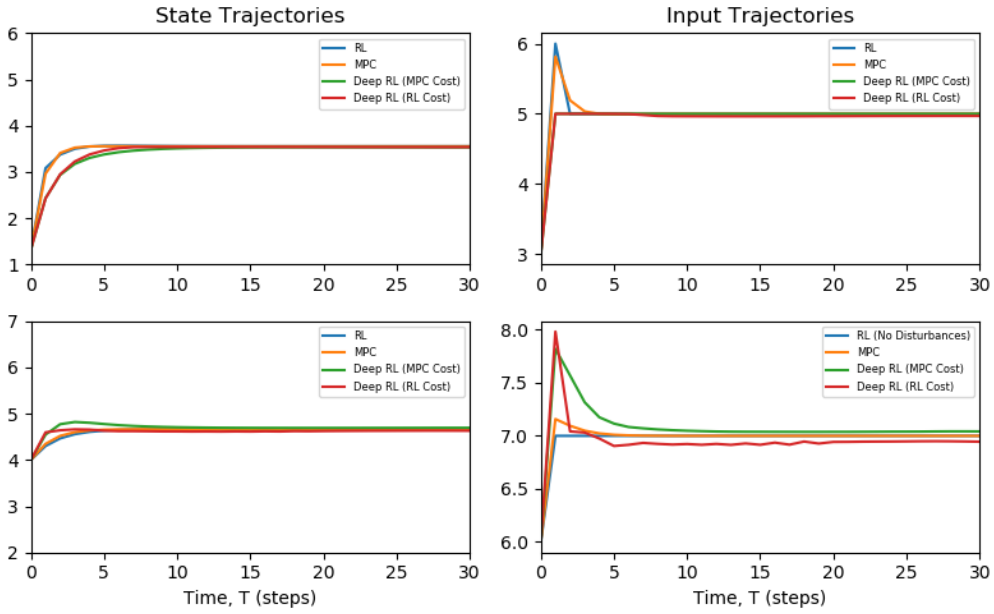


Figure 5.14: Input and state trajectories of the four control strategies on the MIMO system. The top figures denote x_1 and u_1. The bottom figures denote x_2 and u_2.

Table 5.11: Summary of the controller behaviour on the four different system.

MPC	**Tabular RL**	**Deep RL with MPC cost**	**Deep RL with RL cost**
0.86	0.95	2.33	2.59

5.3.3 Discounted Stage Cost for MPC

To enhance the comparability of the two methods, MPC was changed to have an infinite horizon; however, the cost function would be discounted at each successive state. The new MPC cost function is given as:

$$J = \sum_{i=1}^{N} \gamma^i [x^T Q x + u^T R u] \tag{5.19}$$

Using this new cost function, the previous comparisons along with new SIMO and MISO systems were used to compare the four control strategies. The new state

and input trajectories of the SISO, SIMO, MISO, and MIMO systems using the four control strategies are shown in Figures 5.15, 5.16, 5.17, and 5.18, respectively. The system descriptions and the costs are provided in Table 5.12. Note that the deep RL using the custom RL cost was omitted here because it under-performed in all previous scenarios. It can be seen that after discounting was introduced into the MPC cost function, all trajectories look nearly identical with only slight differences. MPC was able to achieve the lowest cost (theoretically sound); however, RL and deep RL both achieved very comparable results through self-play alone! From these plots, it can be concluded that RL does indeed approach the optimality of MPC, although cannot achieve exactly the optimal results.

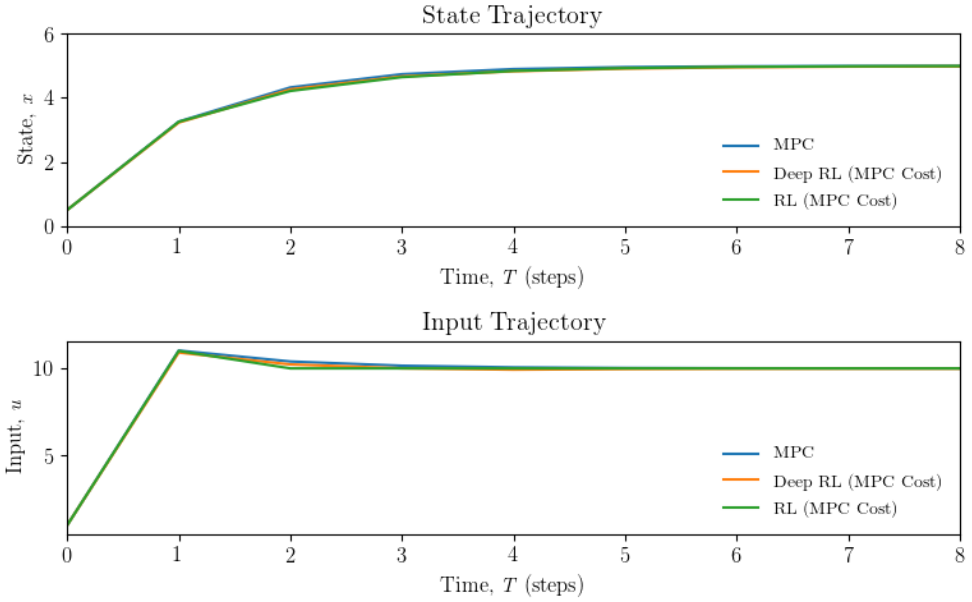


Figure 5.15: Input and state trajectories of the four control strategies on the SISO system. MPC cost is calculated using Equation 5.19.

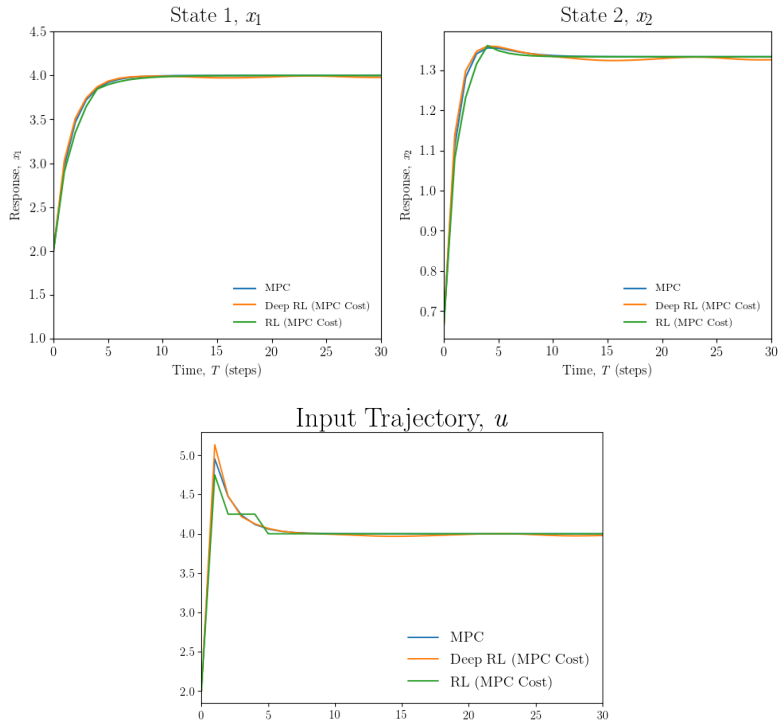


Figure 5.16: Input and state trajectories of the four control strategies on the SIMO system. MPC cost is calculated using Equation 5.19.

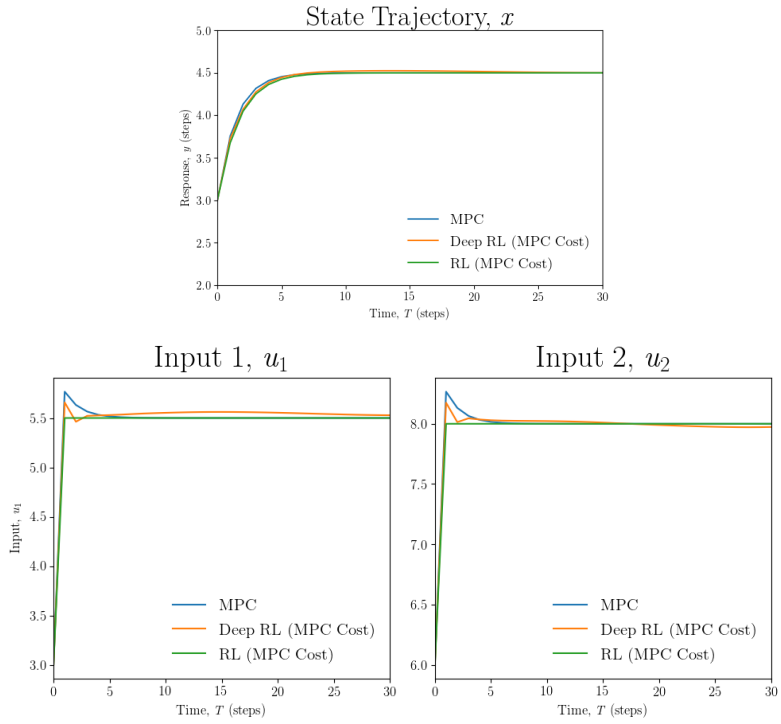


Figure 5.17: Input and state trajectories of the four control strategies on the MISO system. MPC cost is calculated using Equation 5.19.

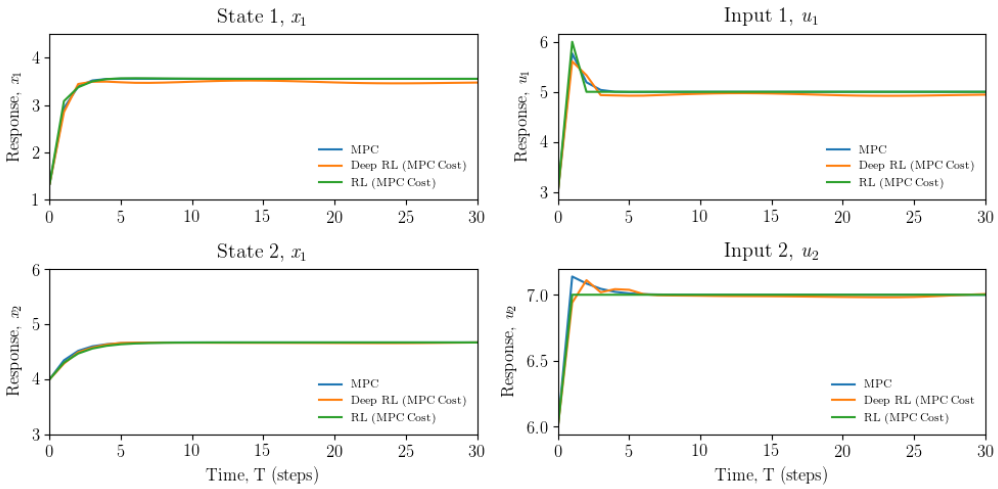


Figure 5.18: Input and state trajectories of the four control strategies on the MIMO system. MPC cost is calculated using Equation 5.19.

Table 5.12: Controller cost for the input and state trajectories on the MIMO system.

	SISO	**SIMO**	**MISO**	**MIMO**
System Equation	$\frac{dx}{dt} = -4x + u$	$\frac{dx_1}{dt} = -2x_1 + u$ $\frac{dx_2}{dt} = -3x_2 + u$	$\frac{dx}{dt} = -3x + u_1 + u_2$	$\frac{dx_1}{dt} = -3x_1 - 2x_2 + 4u_1$ $\frac{dx_2}{dt} = -3x_2 + 2u_2$
Initial States	$x_0 = 0.5$ $u_0 = 1.0$	$x_0 = [2.0, 0.7]$ $u_0 = 2.0$	$x_0 = 3.0$ $u_0 = [3.0, 6.0]$	$x_{ss} = [1.3, 4.0]$ $u_{ss} = [3.0, 6.0]$
Steady States	$x_{ss} = 5.0$ $u_{ss} = 10.0$	$x_{ss} = [4.0, 4.3]$ $u_{ss} = 4.0$	$x_{ss} = 4.5$ $u_{ss} = [5.5, 8.0$	$x_{ss} = [3.6, 4.7]$ $u_{ss} = [5.0, 7.0]$
MPC Cost	4.18	2.1	0.83	0.87
Tabular RL Cost	4.33	2.2	0.97	0.95
Deep RL Cost	4.32	2.1	0.89	1.14

5.3.4 Comparison of RL and MPC on a CSTR

All previous systems were applied onto arbitrary linear systems. Here, a non-linear CSTR system will be used to evaluate the optimality of each control strategy. The CSTR is given by the following differential equations [98]:

$$\frac{dC}{dt} = \frac{F_0(c_0 - c)}{\pi r^2 h} - k_0 exp(-\frac{E}{RT})c \tag{5.20}$$

$$\frac{dT}{dt} = \frac{F_0(T_0 - T)}{\pi r^2 h} + \frac{\Delta H}{\rho C_p} k_0 exp(-\frac{E}{RT})c + \frac{2U}{\gamma \rho C_p}(T_c - T) \tag{5.21}$$

$$\frac{dh}{dt} = \frac{F_0 - F}{\pi r^2} \tag{5.22}$$

The constants in the above equations are:

$$\begin{array}{ccc} F_0 = 0.1m^3/min & F = 0.1m^3/min & T_0 = 350K \\ c_0 = 1kmol/m^3 & \gamma = 0.219m & k_0 = 7.2x10^{10} \\ E/R = 8750K & U = 54.94kJ/min * m^2 * K & \rho = 1000kg/m^3 \\ C_p = 0.239kJ/kg * K & \Delta H = -5 \times 10^4 kJ/kmol & \end{array}$$

Here, the agent's states, x_1 and x_2, are the concentration of reactant A and the reactor temperature, respectively. The control action, u_1, is the coolant temperature. Throughout this study, the reactant height inside the reactor remained constant. The optimal set-points of the system were given as: $x_1 = 0.88, x_2 = 324.5$. The set-up of the tabular RL is as follows: $x_1 = [0.5, 0.525, ..., 1.2]_{1\times 29}$, $x_2 = [300, 302, ..., 350]_{1\times 26}$, and $u_1 = [285, 286, ..., 315]_{1\times 31}$. For the deep RL agent, the neural network set-ups were identical as for the SISO and MIMO systems.

Initially, all controllers (tabular RL, deep RL, and MPC) were evaluated once every five seconds. The state and input trajectories in this case are shown in Figure 5.19. Interestingly, all the state and input trajectories are almost identical. Intuitively, this means that RL was able to exactly recover the optimal solution in this case. Deep RL was slightly off the optimal solution towards the end of the trajectory, but it was minuscule ($T = 0.2$).

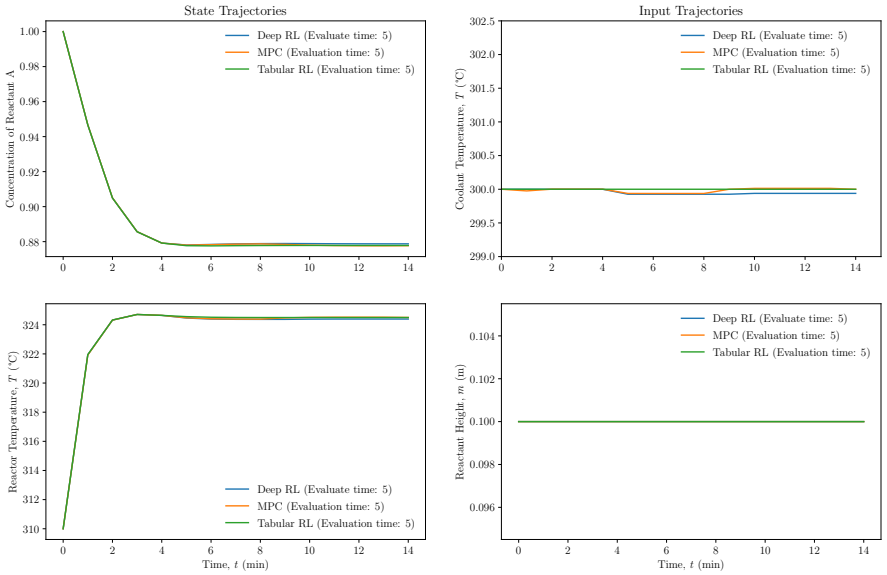


Figure 5.19: Input and state trajectories of the CSTR using controllers with sampling time = 5.

Figure 5.20 shows the new state and input trajectories when the sampling time reduces to 1. The tracking errors here are 2.9, 6.6, 188.9 and 6.5 for the MPC, tabular RL, deep RL with MPC cost, and deep RL with RL cost, respectively. In this system, the dynamics are too slow and do not finish in 1 second. Hence, RL showed slight sub-optimality due to the system not being Markovian. Additionally, deep RL showed an offset from the optimal set point. This extremely small off-set was difficult to identify for an agent using deep learning function approximation; thus, leading to an off-set. However, the off-set was able to be eliminated through training a separate agent using the reward proposed in Equation 5.3. In this reward function, there was more emphasis placed on small off-sets.

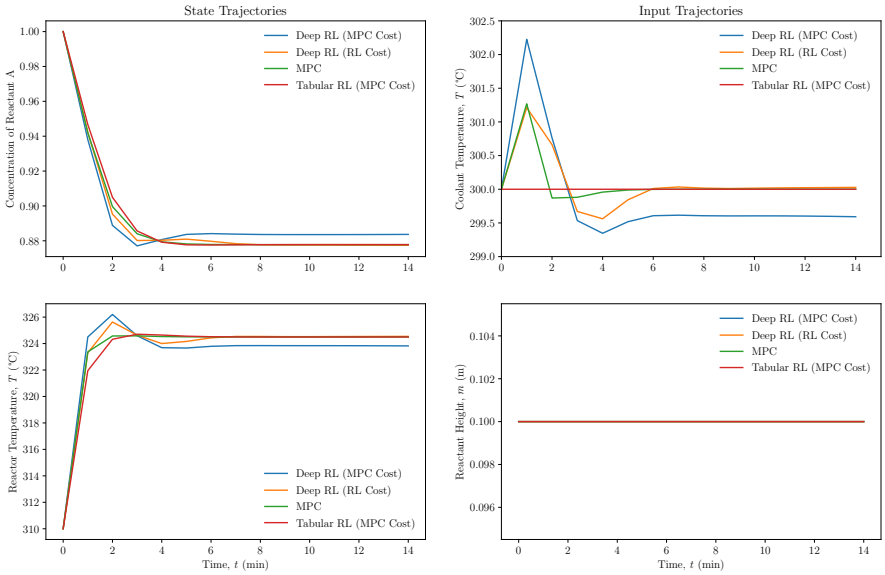


Figure 5.20: Input and state trajectories of the CSTR using controllers with sampling time = 1.

The control strategies were also explored when a disturbance was introduced into the system. It was assumed that the reactor ran away for a second, resulting in a large decrease in reactor temperature. Here, the RL agents were trained for 500,000 time steps, where a random disturbance was introduced once every 100 steps. After training, the RL was simulated against the MPC in the disturbance case. The tracking errors (as per Equation 5.19) for the MPC, deep RL with MPC cost, and deep RL with RL cost are 58.1, 57.4 and 43.9, respectively. Comparatively, the agent using the RL cost function was actually able to accumulate significantly lower tracking error compared to MPC equipped with a *perfect process model*.

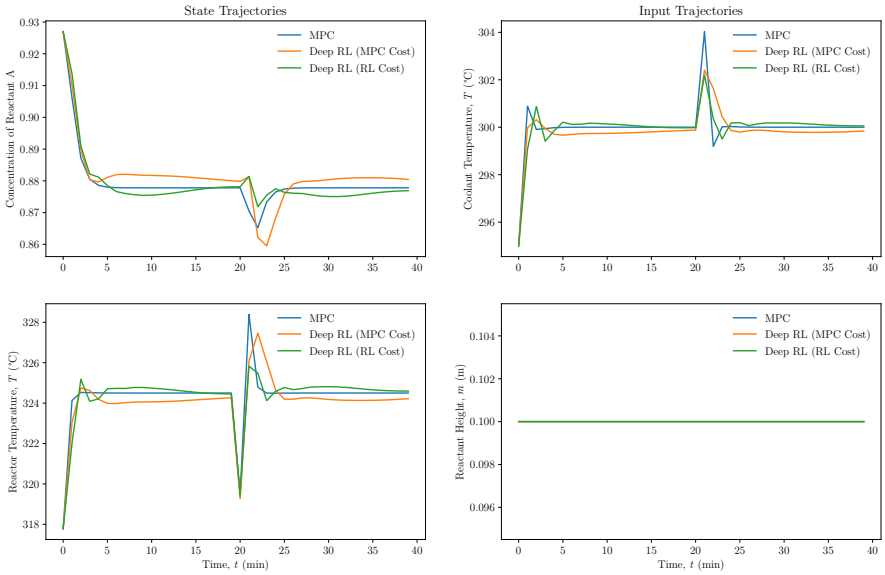


Figure 5.21: Input and state trajectories of the CSTR under a disturbance.

5.4 Control of Wastewater Treatment Plant

As shown in the previous section, RL can approach the optimal solution on small, simple problems. This section applies an UCB tabular Q-learning algorithm for the optimal control of an industrial scale waste water treatment plant. The details of the plant were previously introduced in the alarm management section in Chapter 3. Here, the objectives are twofold: i) Design a self-learning and adaptive RL controller to seek out optimal operating strategies. In the WWTP, the controllers must identify the optimal policy to meet government regulations in the most energy efficient way; ii) direct adaptive control, allowing the RL controller to learn optimal operating strategies as the operating conditions change by adapting the policy directly. In the end, the results achieved by RL will be compared to different variants of MPC.

As a quick recap, the WWTP contains 145 states and 2 control actions and the schematic is provided again, in Figure 5.22. In the WWTP, the dissolved oxygen level in unit 5 is controlled via manipulation of the oxygen transfer coefficient, u_{kLa_5} and the nitrogen level in unit 2 is controlled through manipulation of the internal

recycle flow rate, Q_a. For more information regarding the WWTP, see [88].

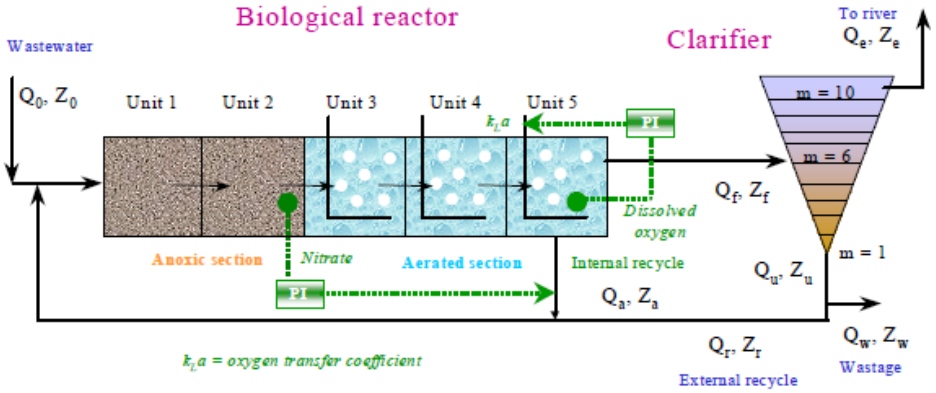


Figure 5.22: Schematic of the WWTP process. Original image from [88].

5.4.1 Performance Assessment

The WWTP performance is evaluated based on two metrics, the effluent quality (EQ) (kg pollutant per day) and the overall cost index (OCI). Each performance metric contains numerous states. For the exact definition of each state, please refer to [88]. The effluent quality is given by:

$$EQ = \frac{1}{T \cdot 1000} \int_{t_0}^{t_f} (\beta_{ss} \cdot SS_e(t) + B_{COD} \cdot COD_e(t) + B_{Nkj} \cdot S_{Nkj,e}(t) + B_{NO} \cdot S_{NO,e}(t) + B_{BOD5} \cdot BOD_e(t)) Q_e(t) dt \quad (5.23)$$

where T denotes the evaluation period in days, the 1000 in the denominator is for unit conversion to kg, and:

$$S_{Nkj,e} = S_{NH,e} + S_{ND,e} + X_{ND,e} + i_{XB}(X_{B,H,e} + X_{X,A,e}) + i_{XP}(X_{P,e} + X_{i,e})$$

$$SS_e = 0.75 \cdot (X_{S,e} + X_{I,e} + X_{B,H,e} + X_{B,A,e} + X_{P,e})$$

$$BOD_{S,e} = 0.25 \cdot (S_{S,e} + X_{S,e} + (1 - f_p) \cdot (X_{B,H,e} + X_{B,A,e}))$$

$$COD_e = S_{S,e} + S_{I,e} + X_{S,e} + X_{I,e} + X_{B,H,e} + X_{B,A,e} + X_{P,e}$$

The OCI is a combination of four factors: 1) sludge production (SP) that requires

disposal; 2) aeration energy (AE); 3) pumping energy (PE); 4) mixing energy (ME).

The sludge production is measured as the average solids production per day (kg/day) and is given as:

$$SP = \frac{0.75}{T \cdot 1000} \int_{t_0}^{t_f} (X_{s,w}(t) + X_{I,w}(t) + X_{B_A,w}(t) + X_{B_H,w}(t) + X_{P,w}(t)Q_w(t)dt + \frac{1}{T \cdot 1000}(SS(t_f) - SS(t_0)) \quad (5.24)$$

where solids include the sedimentary particles remaining in the WWTP and the particles discharged through the waste flow, Q_w. Additionally, subscripts w denote the waste flow.

The aeration energy (kWh/day) is a function of the oxygen transfer rate in units 1-5, and is given by:

$$AE = \frac{S_o^{sat}}{T \cdot 1800} \int_{t_0}^{t_f} \sum_{i=1}^{5} V_i \cdot K_L a_i(t)dt \quad (5.25)$$

where V_i and $K_L a_i$ represents the volume and oxygen transfer rate of the i^{th} unit, respectively. Furthermore, S_o^{sat} is the saturation concentration of oxygen. Here, the value was assumed to be 8 g/m^3.

Similarily, the pumping energy (kWh/day) accounts for the energy consumed by the internal recycle and outer recycle pumps. PE is mathematically quantified as:

$$PE = \frac{1}{T} \int_{t_0}^{t_f} (0.004Q_a(t) + 0.05Q_w(t) + 0.008Q_T(t))dt \quad (5.26)$$

The mixing energy (kWh / day) is calculated as the total energy consumed for mixing activities in the anoxic units (units 1 and 2). Total ME consumption is given as:

$$ME = \frac{24}{T} \int_{t_0}^{t_f} \left(\sum 0.005 \cdot V_i \right) dt \quad (5.27)$$

Finally, the OCI can be computed as the summation of the above equations and

is given as:

$$OCI = 5 \cdot SP + AE + PE + ME \tag{5.28}$$

5.4.2 Agent design

In this study, an *off-policy* Q-learning agent with UCB acceleration was used for the controllers. The hyper parameters of the agent used to control the WWTP are provided in Table 5.13. The agent's states and actions are $x \in \mathbb{R}^2$ and $u \in \mathbb{R}^2$.

Table 5.13: Hyper parameters for the agent controlling the WWTP.

Hyper Parameter	**Value**
States, x	$x_{O_2} = [0.35, 0.5, ..., 2.35]_{1\times65}$
	$x_{N_2} = [1, 2.5, ..., 3]_{1\times65}$
Actions, u	$\Delta u_{O_2} = [-0.50, -0.44, ..., 0.50]_{1\times16}$
	$\Delta u_{N_2} = [-0.50, -0.44, ..., 0.50]_{1\times16}$
Reward, r	Equation 5.33
Learning rate, α	[0.001, 0.5]
Discount factor, γ	0.97
Exploratory factor, ϵ	[0.1, 0.5]
Heuristics Acceleration	Upper Confidence Bound
Degree of exploration, c	1.2
Evaluation time	15 minutes
System representation	FOMDP

The reward function of the agent is given by the summation of the following four Equations. Equation 5.29 provides feedback regarding the effluent quality and is given as:

$$r_1 = \begin{cases} 5130 - EQ, & \text{if } EQ < 5130 \\ -(EQ - 5130)^2, & otherwise \end{cases} \tag{5.29}$$

Equation 5.30 considers the aeration energy cost.

$$r_2 = \begin{cases} (3480 - AE) * 3, & \text{if } AE < 3480 \\ -(AE - 3480)^2, & otherwise \end{cases} \tag{5.30}$$

Similarily, Equation 5.31 considers the pumping energy cost.

$$r_3 = \begin{cases} 288 - PE, & \text{if } PE < 288 \\ -(PE - 288)^2, & otherwise \end{cases} \tag{5.31}$$

And lastly, Equation 5.32 is for constraint handling. Note here that constraint handling is not guaranteed with RL.

$$r_4 = \begin{cases} -5000, & \text{if } N_{tot} > 18 \\ -5000, & \text{if } COD > 100 \\ -5000, & \text{if } S_{nh(e)} > 4 \\ -5000, & \text{if } TSS > 30 \\ -5000, & \text{if } BOD > 10 \end{cases} \tag{5.32}$$

Combining Equations 5.29, 5.30, 5.31, and 5.32, the scalar reward function for the agent is given by:

$$r = r_1 + r_2 + r_3 + r_4 \tag{5.33}$$

For exploration, the agent used UCB and followed an ϵ-greedy exploratory policy. As a quick reminder, UCB provides the agent with unique learning rates for each state-action pair and promotes exploration in states that have a high probability of being optimal (instead of random exploration). The UCB algorithm was given in

Equation 2.36 and ϵ was given by:

$$\epsilon = \begin{cases} \frac{1}{1+\sqrt{N_t}}\epsilon_0, & \text{if } N_t \geq 15 \\ 0.5, & otherwise \end{cases}$$

where N_t is the number of times action u was picked in state x and ϵ_0 is the initial *epsilon* value. Intuitively, as $N_t \to \infty, \epsilon_0 \to 0$.

Lastly, the learning rate of the agent was decayed using:

$$\alpha = \begin{cases} \frac{1}{\sqrt{N_t-14}}\alpha_0, & \text{if } N_t \geq 15 \\ 0.5, & otherwise \end{cases}$$

Intuitively, $\alpha \in [0.001, 0.5]$ is very high for the first 15 visits for each state-action pair. Afterwards, the learning rate is decayed until a minimum value of 0.001.

Finally, the pseudo-code for the UCB Q-learning algorithm used in this particular study is given by:

5.4.3 Control System Design

Typically, all implementation of advanced controls into today's process systems reside in the supervisory layer. In such a structure, the traditional regulatory controllers guarantee stability of the process while supervisory controls simply provide ideal operating set-points.

The control system design is shown in Figure 5.23. In this study, the RL controllers were implemented in the supervisory controls layer and only provide the set-points to the PIDs. Such a structure simulates a real world implementation example. Here, the RL controller provides set-points to the Q_a and K_La_5 PIDs.

Two discrete PI controllers were used to control this system. The discrete PI formulation is given by:

$$\Delta u) = K_c[e(t_k) - e(t_{k-1})] + \frac{K_cT_s}{T_i}e(t_k) \tag{5.34}$$

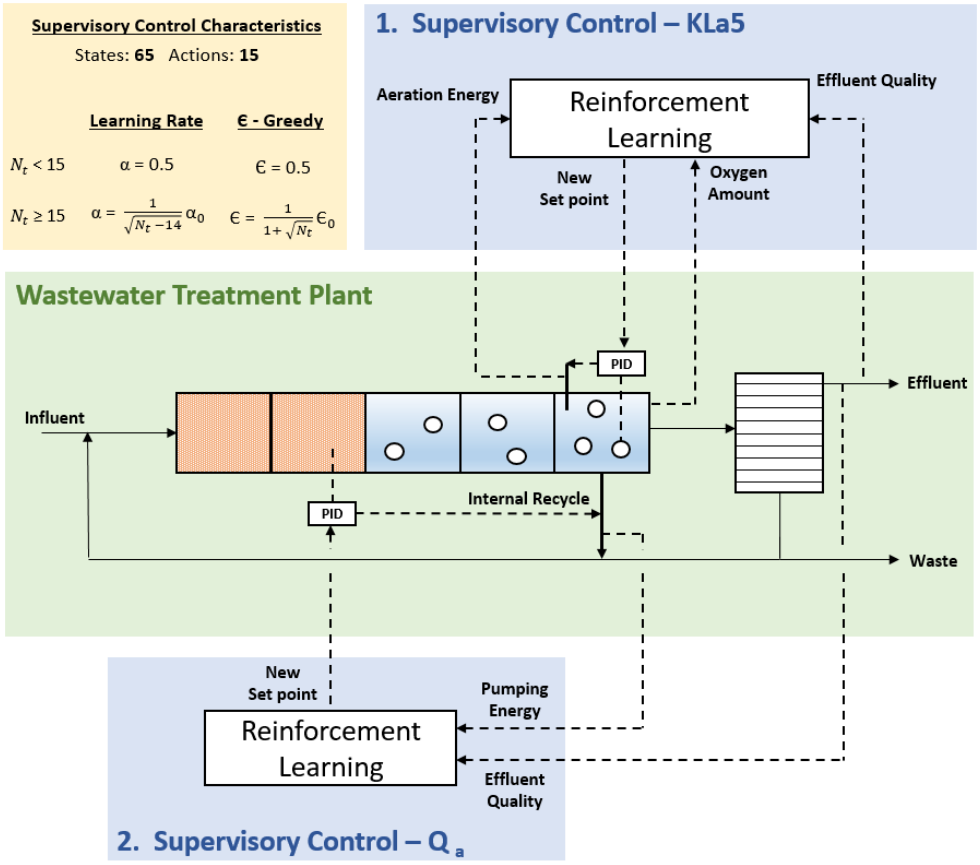


Figure 5.23: Control structure of the wastewater treatment plant.

Upper Confidence Bound (UCB) Q-Learning: Learn Function Q: $\mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$

Require:
States $\mathcal{X} = \{x_1, ..., x_n\}$
Actions $\mathcal{U} = \{u_1, ..., u_n\}$
Reward Function $R : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$
Probabilistic transition function $T : \mathcal{X}_t \times \mathcal{U} \rightarrow \mathcal{X}_{t+1}$
Learning rate function $\alpha \in [0, 1], \quad min = 0.0001$
ϵ-Greedy function $\epsilon \in [0, 1], \quad min = 0.01$
Discount factor $\gamma \in [0, 1]$
Q-matrix $Q : [...]_{x \times u}$
Memory Matrices N_t *and* $T : [...]_{x \times u}$
Procedure *UCB Q-Learning* ($\mathcal{X}$, $\mathcal{U}$, $\mathbb{R}$, T, α, ϵ, γ)
    Initialize $Q(x, u) : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$ *arbitrarily*
    Initialize initial state, x
    While Q is not converged **do**
        Generate random number, $n \sim U[0, 1]$
        If $n > \epsilon$: Choose u from x given max Q, **break ties randomly***
        Else: Choose random u
        Perform action u, observe r, s'
        $Q(x, u) \leftarrow Q(x, u) + \alpha(r + \gamma max'_u Q(x', u') - Q(x, u))$
        $x \leftarrow x'$
        $N_t(x, u) \leftarrow N_t(x, u) + 1$
        $T(x, \neq u) \leftarrow t(x, \neq u) + 1$
    until convergence or x is a terminal state

*Note: Ties must be broken randomly to avoid bias.

$u(u)$	Output as a function of time sample
K_c	Controller gain
$e(t_k)$	Current error
$e(t_{k-1})$	Error at last time step
T_s	Sampling time
T_i	Integral time

The parameters of the two PI controllers are given in Table 5.14 and were originally provided from [88].

The pseudo-code for the supervisory RL controllers are as follows:

Table 5.14: Hyper parameters for the PI controllers

	Dissolved Oxygen Controller	Nitrogen Controller
K_c	25	10,000
τ_i	0.002	0.00167

Supervisory Control - Dissolved Oxygen Controller Information Flow

Initiate Plant Operations

Regulatory Control Layer

Observe dissolved oxygen level, $x_{O_2,t}$, in unit 5

Calculate error e: x_{O_2} - $x_{O_2,setpoint}$

Calculate Δu_{O_2} from Equation 5.34

Supervisory Control Layer *every 15 mins*

Observe initial $x_{O_2,t}$

Generate random number, $n \sim U[0,1]$

If $n > \epsilon$***:***

Choose u from x given max Q, **break ties randomly***

Else:

Choose random u

Perform action: $x_{O_2,sp} \leftarrow x_{O_2,sp} + u$

Observe r_{t+1}, $x_{O_2,t+1}$, update Q, N_t, T

$$Q(x_{O_2,t}, u_t) \leftarrow Q(x_{O_2,t}, u_t) + \alpha(r_t + \gamma \max_{u_{t+1}} Q(x_{O_2,t+1}, u_{t+1}) - Q(x_{O_2,t}, u_t))$$

$x_{O_2} \leftarrow x_{O_2,t+1}$

repeat forever or until terminal state

Supervisory Control - Nitrogen Controller Information Flow

Initiate Plant Operations

Regulatory Control Layer

Observe nitrogen level, $x_{N_2,t}$, in unit 5

Calculate error e: x_{N_2} - $x_{N_2,setpoint}$

Calculate Δu_{N_2} from Equation 5.34

Supervisory Control Layer *every 15 mins*

Observe initial $x_{N_2,t}$

Generate random number, $n \sim U[0,1]$

If $n > \epsilon$***:***

Choose u from x given max Q, **break ties randomly***

Else:

Choose random u

Perform action: $x_{N_2,sp} \leftarrow x_{N_2,sp} + u$

Observe r_{t+1}, $x_{N_2,t+1}$, update Q, N_t, T

$$Q(x_{N_2,t}, u_t) \leftarrow Q(x_{N_2,t}, u_t) + \alpha(r_t + \gamma \max_{u_{t+1}} Q(x_{N_2,t+1}, u_{t+1}) - Q(x_{N_2,t}, u_t))$$

$x_{N_2} \leftarrow x_{N_2,t+1}$

repeat forever or until terminal state

5.4.4 Comparison with MPC

To validate the optimality of the solution provided by RL, it was compared to four variants of MPC: 1) centralized MPC (CMPC); 2) centralized EMPC (CEMPC); 3) distributed EMPC using a centralized model (DEMPCE); 4) distributed EMPC using subsystem models (DEMPCS). The objective functions and subsystem decomposition of the MPCs are from [99]. All controllers had sampling times of 15 minutes. The RL agent was trained for approximately 383.6 years in simulation (2 hour physical time) under dry weather conditions on a typical office computer (Intel

i7-6700 with no graphics processing unit).

The EQ and OCI of RL and the four MPC variants are shown in Table 5.15. Even in such a complex environment, RL was able to generate highly comparable performance to MPC. In terms of EQ and energy usage, RL used more energy but produced less waste while MPC used less energy, but produced more waste. In terms of OCI, RL was only 0.129% higher compared to the best performance achieved by MPC. Moreover, MPC used a perfect process model whereas RL learned the optimal policy solely through self-play. Additionally, the computation time of RL in this system was dramatically less compared to MPC. In fact, most of the computation time used by RL was to solve the system equations, and not to generate the control inputs.

Table 5.15: A comparison of performance between RL and MPC variants. The MPCs all used a prediction and control horizon of 40. DEMPCS and DEMPCE are the distributed economic MPCs using the subsystem and centralized models, respectively [99].

	RL	**CMPC**	**CEMPC**	**DEMPCS**	**DEMPCE**
EQ (kg pollution/day)	6034	6111	5834	6332	5828
Aeration Energy (kWh / day)	3454	3416	-	-	-
Pumping Energy (kWh / day)	315	338	-	-	-
OCI	16207	16186	16244	16197	16698
Computational Time (s)	1.91	8.79×10^4	3.77×10^5	8.47×10^4	3.50×10^5

The three most interesting confirmations of this study were:

1. RL can approach MPC performance even on highly complex, MIMO systems.

2. RL is significantly less computationally demanding compared to MPC during online evaluation

3. RL's online computation time is not a function of the complexity of the system, only a function of its own state and action size (i.e., agents with more states and actions require longer online evaluation time).

5.5 Comparison of Optimal Control Frameworks

The control framework of a typical process was first introduced in Chapter 1, and is shown again here in Figure 5.24. As a brief refresher: RTO evaluates seldomly (hourly basis) and is used to find the optimal steady states with accordance to a desired performance metric [61]. These optimal steady states are then passed onto the MPC layer, where the optimal input trajectories are identified. A typical MPC objective function is:

$$J = \sum_{i=1}^{H} x_i^T Q_{mpc} x_i + u_i^T R_{mpc} u_i \tag{5.35}$$

due to its convexity [10]. In recent advanced control literature, researchers intertwined the concepts of RTO and MPC into one unifying algorithm that is now known as economic model predictive control (EMPC) [62], [63]. Here, the objective function of EMPC explicitly contains the economic objectives of the process.

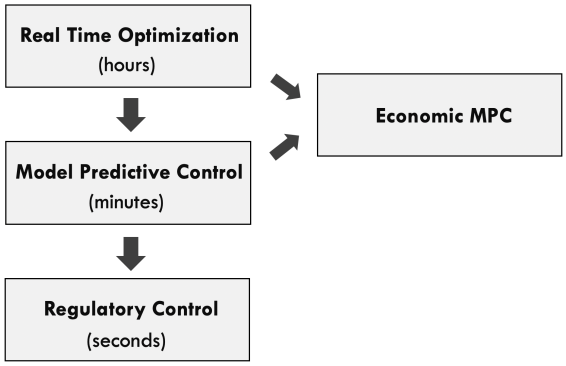


Figure 5.24: A typical industrial control architecture.

In control literature, there typically exists very distinct algorithms for each layer presented in 5.24. For example, PID controllers are typically used for the regulatory layers whereas the supervisory layers typically employ predictive controls. Due to RL's general nature, it is highly flexible and can used in *any* control layer. For example, a MPC RL agent's reward function is simply the negative of Equation 5.35, with the actions being the control actions or recommended set points. For a regulatory layer agent, the reward function remains the same, but the actions would involve direct actuation of the system's hardware. Lastly, an EMPC based RL agent would have an economic objective as the reward function.

The *biggest difference* between RL and other advanced control methods is the computation time. Typical MPCs have a computational complexity of $O(H(n^2 + m^2))$, where H, n, and m are the control horizon and the dimensions of the states and actions, respectively [100]. By exploiting prior information, MPCs' computational complexity can further decrease to $O(H(n+m)^3)$ [101]–[103]. Even then, the online computation time scales exponentially with states and actions, ultimately becoming infeasible for large systems and/or for systems with exceptionally long prediction/control horizons. Comparatively, RL's optimal policy is first pre-computed *offline* through a training process. Consequently, this makes online evaluation exceptionally quick. In control theory, the concepts of RL are very similar to parametric programming from explicit MPCs [60]. Some may see the training requirement of RL as extremely unattractive; however, offline computation time is typically very flexible (i.e., offline computations can be done anywhere, anytime, on any machine) and does not matter so long as it is not unreasonably long (e.g., 1 month).

Another major difference between RL and MP-based methods is RL's *model-free* nature. In RL, a (representative) model is only required for initial training of the agent; online implementations are *model-free*. Conversely, the system model is almost exclusively used in MPC. Inaccurate models are detrimental to control performance. In literature, a technique known as off-set free control overcomes this issue through online parameter re-adjustment [104]; however, the re-identification process does not work well for extremely noisy processes. Moreover, most plants experience process drift, where the processes slowly changes as a function of wear and tear. RL can inherently adapt to such an issue through a gradual and smooth process. MPCs, however, adapt using off-set free control or other model re-identification methods. For such methods, the updates directly and completely change the model parameters at each sampling time. For processes heavily corrupted by noise, the parameters at each update is most certainly incorrect.

Lastly, tabular RL has very few hyper parameters. As long as reasonable learning rates are used, RL would work for most cases. Deep RL methods contain much more

hyper parameters and are much more mathematically complex. Without a doubt, the tuning of deep RL require many parameter sweeps; however, such details will be omitted here because deep RL may not be economically viable to implement in its current state. For MPC, adequate tuning of the Q_{mpc} and R_{mpc} matrices are often times paramount for optimal process control.

A comparison between RL and MPC on various important categories is shown in Table 5.16. In addition to comparing RL and MPC in literature, RL was also compared to industrial-grade MPCs currently implemented onto many processes world wide. One popular MPC product in industry is AspenTech's signature DMCplus and DMC3 products [105], [106]. When implementing such products in real life, the system model will never be perfect; therefore, only a near-optimal solution is possible. Additionally, the computation time for DMC is exceptionally high and is unviable for many non-linear systems. Comparatively, RL is not concerned with the structure of the system; however, identifying an optimal policy for noisy systems will be more difficult. Adaptation-wise, RL performs random actions to calibrate to the real system (an idea that *sounds* dangerous for online processes). Interestingly, AspenTech's technology also performs such random actions for model calibration. During commissioning, the controller is typically initialized in the *smart step* mode, where the system performs random step tests *online* to calibrate the system model to the real process. Afterwards, operators often switch the system to *calibrate* mode. In this mode, the system continues to perform step tests in a much more infrequent and lower in magnitude fashion. Such an adaptation paradigm is identical to RL, where exploration is initially plentiful, but is eventually annealed to near zero. For an more mathematical comparison between RL and MPC, see [107].

The features of RL, academic MPC, and industrial MPC is shown in Table 5.16.

Table 5.16: A comparison between RL, MPC in literature, and industrial MPC software.

	Reinforcement Learning	**Model Predictive Control**	**AspenTech DMC**
Performance	Close to optimal	Optimal with perfect model	Close to optimal
Online comp. cost	Low	High	High
Offline comp. cost	Policy & model identification	Model identification	Model identification
Reliance on model	Only for training	At all times	At all times
Online calibration	Exploratory moves	Various methods	Exploratory moves
Sensitivity to tuning	Low	High	High

5.6 Fault-Tolerant Control System

It is seen that RL can only approach the performance of MPC in academic studies where MPC utilizes a perfect process model, is well designed, and given sufficient computational time. RL's generality, ease of use, and adaptive nature might create more value in industrial environments where engineers are time constrained and are tasked with assembling a *good enough* controller with limited hardware.

This section explores the generality and robustness of the RL algorithm, even when imperfectly designed and compares it to MPC for fault-tolerant control. The algorithm from this study was then simulated on the Wood-Berry distillation tower from the University of Alberta under different actuator faults.

5.6.1 Introduction

All process equipment such as sensors and actuators may malfunction or breakdown during their operational lifetime. Hence, it is desirable to have a fault-tolerant control system (FTCS) to ensure sufficient performance during these impending failures. The application of FTCS in an industrial environment results in increased operation robustness and safety, while reducing operating costs due to fewer plant-wide shut downs [108]. A typical FTCS contains two parts: i) Fault detection system (FDS) to identify the location and type of fault; ii) fault-tolerant controller (FTC) to operate the process safely during a fault.

Traditionally, a FTCS contains a variety of different controllers to handle different faults that may occur during online operations [108]. Furthermore, PID controllers are generally used for FTC and are specially tuned to handle each fault specifically [109]. The traditional approaches work well in terms of safety, but the sheer number of controllers command a high maintenance cost. Moreover, the controllers must be re-tuned periodically for optimal performance due to unavoidable process drifts caused by wear and tear [57].

In this study, a reinforcement learning (RL) FTCS is proposed where different system faults are detected and mediated using a general controller. Additionally, the FTC can automatically adapt to process drift and new operating conditions. The FTCS' set-up is designed to be general enough to learn different faults using the same algorithm and will reside on top of existing regulatory control systems. Furthermore, the controller does not suffer greatly due to poor tuning or model plant mismatch, a trait plaguing traditional optimal controllers [110]. The proposed FTCS is implemented onto continuous multiple-input multiple-output (MIMO) systems with input constraints subject to actuator faults. Various RL FTCS were previously proposed for discrete systems, but not in the continuous case where transition dynamics are explicitly considered [111], [112]. To demonstrate this approach, the system outputs are assumed to be measurable and the system dynamics are assumed to be stable using a pre-designed controller under the fault-free case. During normal operations, the FDS will learn the expected closed-loop behaviour of the system. Behaviours heavily deviating from the expected states are used to identify faults. The FTC is activated during faults to operate the system using the non-faulty components.

5.7 Preliminaries

5.7.1 System Description

A class of continuous-time MIMO systems with constrained manipulated inputs is considered in this study and described in state space form by:

$$\dot{x}(t) = f(x(t), u(t) + \tilde{u}(t)) \tag{5.36}$$

$$y = h(x(t)) + \varepsilon(t) \tag{5.37}$$

where $x(t) \in \mathbb{R}^m$ denotes the state vector at time t, $u(t)$ denotes the inputs at time t, $y = (y_1, y_2, ..., y_n) \in \mathbb{R}^n$ denotes the output variables, and $\tilde{u}(t)$ denotes the constrained manipulated input corresponding to actuator faults, which will be the focus of this study. Lastly, $\varepsilon \sim N(0, \sigma^2)$ denotes Gaussian noise in the measurements of the output variables.

5.7.2 The Reinforcement Learning Problem

Fig. 5.25 shows the RL paradigm. The overall RL problem was stated in detail in Chapter 1, and will only be briefly explained here. Starting from the top, the **environment** includes all factors the agent cannot arbitrarily change (the system in this study). The **agent** observes the **states** of the environment and performs control **actions** that transition the environment to new states while outputting a **reward** based on a desired performance metric. The reward guides the agent to the optimal policy. In control, reward is typically a function of the tracking error. The agent's decision making process is formalized in the Markov decision process.

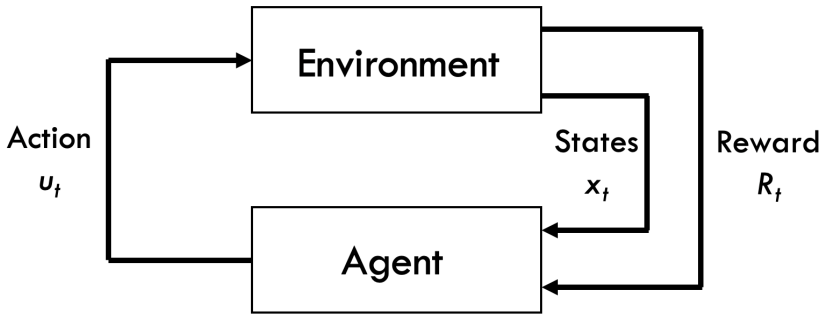


Figure 5.25: Paradigm of the reinforcement learning problem.

5.7.3 Markov Decision Process

The Markov decision process (MDP) is a discrete representation of the stochastic optimal control problem and a classical formulation of sequential decision making [7]. MDPs provide formalism to agents when rationalizing about planning and acting in the face of uncertainty. Many different definitions of MDPs exist and are equivalent up to small alternations of the problem. One such definition is that a MDP, $\mathcal{M}$, is a tuple $(\mathcal{X}, \mathcal{U}, P(x', r|x, u), \gamma, R)$ comprised of [22]:

- $x \in \mathcal{X}$: **State** space that describes the environment.
- $u \in \mathcal{U}$: **Action** space of the agent. $(\mathcal{U} \geq 2)$
- $R \in \mathbb{R}$: Expected **reward** from environment after agent performs u in x. $|R| \leq \mathcal{R}$
- $P(x', r|x, u)$: **State transition probabilities** of the environment. Given $x \in \mathcal{X}$, $u \in \mathcal{U}$, the probability of transitioning to x' and receiving r.
- γ: **Discount factor** associated with future uncertainty. $(0 \leq \gamma \leq 1)$

The agent starts in some initial states, x_0. At each time t, the agent picks an action u_t in accordance to the current policy π and transitions to some x_{t+1} while receiving r_{t+1} drawn from the $P(x_{t+1}, r_{t+1}|x_t, u_t)$. By repeating the above procedure many times, the agent is able to traverse through some sequence $x_t, u_t, r_{t+1}, x_{t+1}, u_{t+1}, r_{t+2}, ...$ and

accumulate:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5.38}$$

where G_t is the total discounted return along the sequence. Here, the discount factor, γ, captures the uncertainty of future rewards and keeps G_t bounded for non-terminating tasks. R_t is the reward received at time t. The objective of the agent is to find the optimal control policy π^*, that maximizes G_t. Optimal solutions for MDPs work well for discrete tasks when transition times are constant and dynamics of the system are disregarded. However, such systems are rare in the process industry.

Fig. 5.26 shows different cases of poorly designed controllers in the process industry. Controllers resulting in oscillations, large overshoot, or severe inverse response lead to faster equipment deterioration and are detrimental to process safety [57]. Additionally, transition time is often determined by the magnitude of change in the set-point in non-linear systems. Controllers typically require more time to track higher magnitude changes compared to ones of lower magnitude. One could design a controller that evaluates seldomly to guarantee successful transitions, though such excessively conservative designs may lead to economic disadvantages.

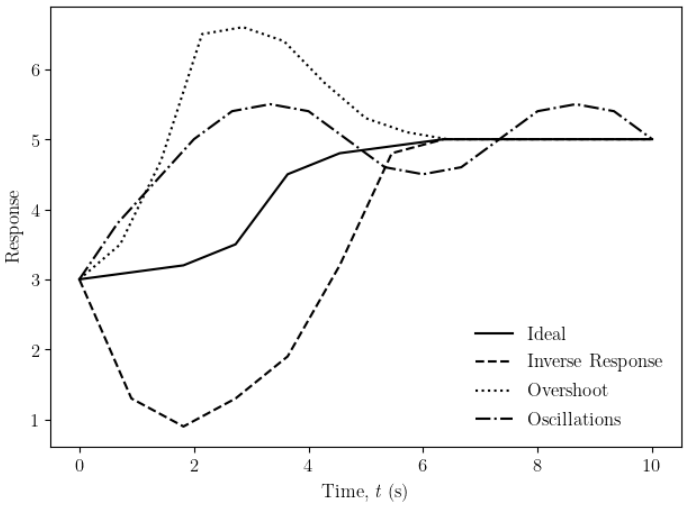


Figure 5.26: Symptoms of poorly designed controllers.

5.7.4 Semi Markov Decision Process

System dynamics and transition times are vital to successful process control; therefore, continuous control problems with unknown transition times are formalized using the semi-Markov decision process (SMDP). In SMDPs, the dynamics of the transition period are captured using the reward [25]:

$$R(x_t, x_{t+1}, u_t) = \int_0^\infty \int_0^t e^{-\beta s} \rho(x, \pi(x)) ds dF_{x,x_{t+1}}(t|\pi(x)) \tag{5.39}$$

where $\rho(x, \pi(x))$ and t are the average reward and transition time for the transition period from x_t to x_{t+1}. Additionally, $\beta \in [0, \infty)$ is the discount factor for SMDPs. High β values result in short-sighted agents. $F_{x_t,x_{t+1}}(t|\pi_t)$ is the probability distribution of the time required for the system to transition from x_t to x_{t+1} given π_t. The squared tracking error is calculated during intermediate transition periods, explicitly capturing transition dynamics during the search for π^*. Here, rewards for unknown transition time systems are corrected using:

$$\gamma(x_t, x_{t+1}, u) = \int_0^\infty e^{-\beta t} dF_{x_t,x_{t+1}}(t|\pi_t) \tag{5.40}$$

5.8 Proposed Fault-Tolerant Control System

Fig. 5.27 shows the proposed FTCS for any industrial process. The system contains three parts: i) Industrial process; ii) fault detection; iii) fault-tolerant control. The industrial process can be any arbitrary system (e.g., distillation tower, chemical reactor). A contextual bandit algorithm was used for the FDS. Subsequently, a tabular Q-learning approach was used for FTC. A bandit-based approach was selected for fault detection because the agent is not concerned with the long term reward (i.e., it is only concerned with the accuracy of its immediate classification) [7]. Contrarily, sequential decision making is critical for the success of an agent in control. Thus, an RL-based agent was used for FTC [113]. The tabular based

approach was selected for its simplicity and ease of implementation into industrial distributed control systems (DCS), much like how explicit MPCs are implemented into processes that demand fast computations on cheap hardware with little storage [60]. The flaws of the current algorithm are its lack of scalability and discrete nature.

For high dimensional industrial processes equipped with modern hardware, both scalability and discreteness can be eliminated by using deep Q-network (DQN) and deep deterministic policy gradient (DDPG) for the FDS and FTC, respectively. DQN with $\gamma = 0$ is an ideal algorithm for the FDS due to its continuous state space and discrete action space. Likewise, DDPG is ideal for control due to its ability to handle large continuous states and action systems. For the remainder of this study, the contextual bandit agent used for fault detection and the RL agent used for FTC will be denoted as the *prediction agent* and *control agent*, respectively.

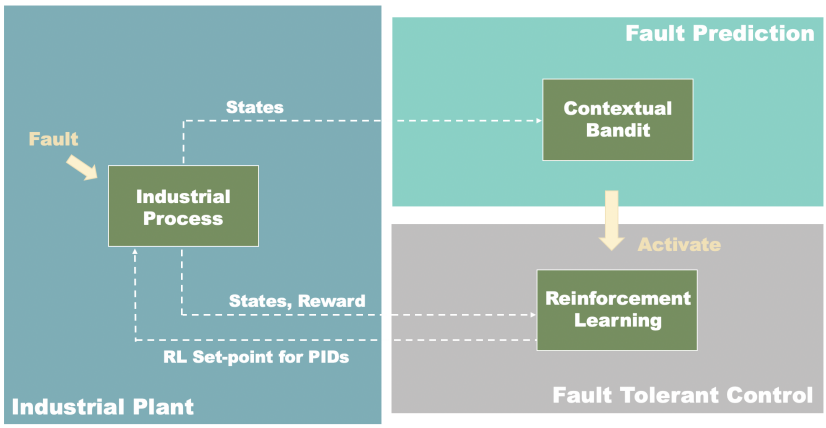


Figure 5.27: Overall set-up of the fault-tolerant control system.

In Fig. 5.27, the information flow is as follows. Initially, the industrial process is operating fault-free while the prediction agent is actively monitoring real time measurements for faults. When a fault is detected, the prediction agent will immediately activate the control agent to receive real time measurements from the process. Given the current process off-set, the control agent gives recommendations to the operators regarding solutions to mediate the situation. Recommendations can be new set-points for the regulatory controllers that are not at fault. Fig. 5.28 shows the information flow of the control agent.

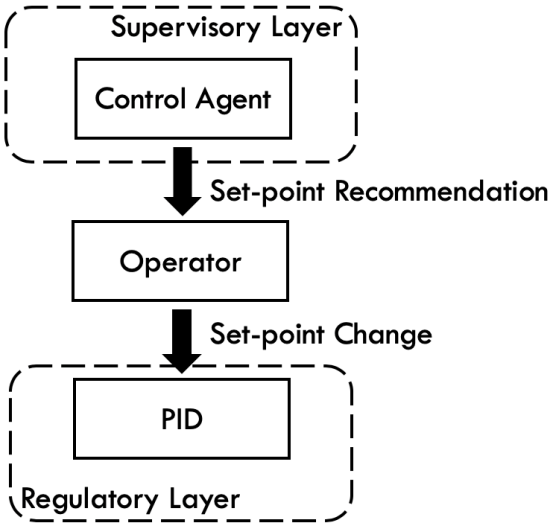


Figure 5.28: Information flow from the FTCS to the process.

5.8.1 Contextual Bandits Fault Detection

The fault detection system is used to identify faults in the process. FDS can be categorized as model-, knowledge- or prediction-based approaches [114]. Model-based approaches require an explicit model of the process. A fault is deemed active if the prediction of the model is drastically different from the real time sensor measurement. Knowledge-based approaches are based on subject matter expertise from process operators or equipment vendors and are usually rule-based (e.g., if $x > x_{max}$, then fault). Prediction-based approaches use historical data to identify a classification model for fault detection. The identified model would contain knowledge about normal operating boundaries regarding the process, and can be tuned by plant managers to be more conservative or aggressive. When the process conditions are outside the acceptable boundaries, a fault is deemed active.

Prediction-based methods are quickly becoming the forefront approach due to more readily available data. This study uses a contextual bandit prediction-based FDS to identify process faults. A contextual bandit algorithm was selected due to its ability to adapt to non-stationary problems. Furthermore, bandit-based algorithms are well suited for identification tasks because the agent is not concerned with long

term reward [7]. The objective of the agent is to identify if the current situation is faulty, given the current states x_t of the process.

Prediction Agent Algorithm

In contextual bandit problems, for each time t, the agent observes states x_t and picks one action $u_t \in \mathcal{U}$. After each action, a scalar reward feedback is sent to the agent as feedback to promote or discourage future similar state-action pairs. For each action in state $x \in \mathcal{X}$, there is an expected reward called *action value*, given by Equation (5.41).

$$q^*(x,u) = \mathbb{E}[R_t|X_t = x, U_t = u] \tag{5.41}$$

where $q^*(x,u)$ is the expected reward of taking u in x. Here, R_t is drawn from a distribution, $R_t \sim N(q_*(x,u), \sigma^2)$ [7]. The real action-value is unknown, but can be estimated from Equation (5.42) [113].

$$Q^{n+1}(x,u) \leftarrow Q^n(x,u) + \alpha_n(R_t - Q^n(x,u)) \tag{5.42}$$

where $Q(x,u)$ and n are the estimate of $q^*(x,u)$ and the number of times $Q(x,u)$ was estimated prior to the current estimate. α is the learning rate and is constant for adapting to non-stationary problems [7].

Table 5.17 shows the reward space for the prediction agent. Furthermore, $\mathcal{U}$ and $\mathcal{X}$ are given by $Actions = [Fault, No\ Fault]$ and $[x_1^r, x_2^r, ..., x_v^r]$, respectively. Superscript r and subscript v denotes the relevant states and the number of relevant states, respectively. To train the prediction agent, first, the historical data must be

Table 5.17: Reward for the prediction agent.

Process Fault	Action	Reward
Yes	Fault	1
Yes	No Fault	-1
No	Fault	-1
No	No Fault	0

labeled for faults. Next, the prediction agent will sample from the historical data

and update its internal action values with accordance to Table 5.17.

Contextual Bandit: *Learn* f*:* $\mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$
Require:
States $\mathcal{X} = \{x_1, x_2, ..., x_v\}$
Actions $\mathcal{U} = \{u_1, u_2\}$
Reward Function $R : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$
Learning rate $\alpha \in [0, 1]$
Procedure *Contextual Bandit* ($\mathcal{X}$, $\mathcal{U}$, R, α)
Initialize zero matrix $Q(x, u)_{\mathcal{X} \times \mathcal{U}}$
While Q not converged **do**
Sample state, x_t
Pick $\arg\max_{u_t} Q(x_t, u_t)^*$
Perform action u_t, observe R_{t+1}
$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha(R_{t+1} - Q(x_t, u_t))$
*Note: Ties broken randomly to avoid bias.

Once $Q(x, u)$ reaches convergence, real time process measurements are sent to the prediction agent to detect potential faults. Action selection is given by:

$$u_t = \arg\max_{u} Q_\pi(x, u), \ \forall \ x \in \mathcal{X} \tag{5.43}$$

The prediction agent will activate the control agent when a fault is deemed active.

5.8.2 Reinforcement Learning Fault-Tolerant Control

Once activated, the control agent provides recommendations to stabilize the process. Two strategies exist for the control agent: active FTC and passive FTC [115]. Active FTC refers to re-configurable control, whereas passive FTC uses robust control principles. Active FTCs are generally more economically advantageous because passive FTCs are relatively more conservative. A tabular Q-learning active FTC is used for this study because of its adaptive nature and ability to acknowledge future rewards [116].

Control Agent Algorithm

Reinforcement learning is similar to contextual bandits with the alteration that the long term trajectory is also considered. The Q-learning algorithm for MDPs is:

$$Q^{n+1}(x,u) \leftarrow Q^n(x,u) + \alpha_n(R_t + \gamma max'_u Q^n(x',u') - Q^n(x,u)) \tag{5.44}$$

where x' and u' are the next state and the action that maximizes the return in x'. By combining Equation (5.44) with Equations (5.39) and (5.40), SMDP Q-learning is given by:

$$Q^{n+1}(x,u) \leftarrow Q^n(x,u) + \alpha_n \left[\frac{1-e^{-\beta\tau}}{\beta} R_t + e^{-\beta\tau} max_{u'} Q^n(x',u') - Q^n(x,u) \right] \tag{5.45}$$

where R_t is given by Equation (5.39) and $\tau \leq \tau_{max}$ is the transition time from x_t to x_{t+1}. If $x_t \neq x_{t+1}$ at τ_{max}, the agent evaluates regardless. The reward is:

$$\rho = -\sum_{k=1}^{m} (y_k(t) - y_k^{sp}(t))^2 = -\sum_{k=1}^{m} e_k(t)^2 \tag{5.46}$$

where $y_k^{sp}(t)$ and $e_k(t)$ are the set-point and tracking error for y_k at t. The states and actions are discretized as:

$$\mathcal{X} = [(e_1^{min}, ..., e_m^{min}), ..., (e_1^{max}, ..., e_m^{max})]_{p^2 \times 1} \tag{5.47}$$

$$\mathcal{U} = [(u_1^{min}, ..., u_o^{min}), ..., (u_1^{max}, ..., u_o^{max})]_{q^2 \times 1} \tag{5.48}$$

where superscripts min and max denotes the min and max for each state error or action. $p, q \geq 2$ denotes the number of discretized values.

During training, exploration of the environment is mandatory to avoid locally optimal policies. Traditional exploration methods, such as ϵ-greedy, indiscriminately tries non-greedy actions (i.e., non reward maximizing actions given the current knowledge) with a fixed probability [7]. However, exploring in frequently visited states makes little sense. Instead, it be better to select non-greedy actions based

on their potential of being optimal. One such way to do this is to use UCB action selection [7]:

$$U_t = \arg\max_u \left[Q_t(x,u) + c\sqrt{\frac{ln\ t}{N_t(x,u)}} \right] \tag{5.49}$$

where $ln\ t$ and c are the natural logarithm of t and the exploratory factor, respectively. Large c values result in more exploration. $N_t(x,u)$ is the number of times u is picked in x prior to t. The square root term is the measure of uncertainty in the current Q values. Uncertainty is reduced each time u is selected by increasing $N_t(x,u)$.

The control agent is trained using a high-fidelity simulator for the system.

UCB Q-Learning: *Learn* $f: \mathcal{X} \times \mathcal{U} \rightarrow Q$

Require:

States $\mathcal{X} = \{x_1, x_2, ..., x_m\}$

Actions $\mathcal{U} = \{u_1, u_2, ..., u_o\}$

Reward function $R : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$

Learning rate $\alpha \in [0, 1]$

SMDP discount factor $\beta \in [0, \infty)$

Degree of exploratory $c \in [0, \infty)$

Procedure *UCB Q-learning* ($\mathcal{X}$, $\mathcal{U}$, R, α, β, c)

Initialize zero matrices $Q(x, u)_{\mathcal{X}\times\mathcal{U}}$, $N(x, u)_{\mathcal{X}\times\mathcal{U}}$

Initialize time, t_0

Observe initial state, x_0

While Q is not converged **do**

Pick $\arg\max_{u_t} Q(x_t, u_t) + c\sqrt{\frac{ln\, t}{N(x,u)}}^*$

Perform u_t, expect x_{t+1}

When $x \approx x_{t+1}$, observe $R(x_t, x_{t+1}, u_t), \tau$

$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha \left[\frac{1-e^{-\beta\tau}}{\beta} R \; + ... \right.$

$\left. e^{-\beta\tau} max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)\right]$

$N(x, u) \leftarrow N(x, u) + 1; \quad x_t \leftarrow x_{t+1}; \quad t \leftarrow t + 1$

*Note: Ties broken randomly to avoid bias.

After convergence of $Q(x, u)$, c is set to zero to stop exploration. Actions are picked using:

$$u_{i,t} = u_{i,t-1} + \Delta u_{i,t}, \quad u_i^{min} \leq u_{i,t} \leq u_i^{max} \tag{5.50}$$

where Δu_t is from Equation (5.43).

5.8.3 Stability and Convergence

The stability of RL is guaranteed assuming a Lipschitz continuous model and confining exploration to within the region of attraction, given a bounded input [117]. For convergence, given learning rates $0 \leq \alpha_n < 1$, bounded rewards $|r_n| \leq \mathcal{R}$ and:

$$\sum_{i=1}^{\infty} \alpha_i(x,u) = \infty, \quad \sum_{i=1}^{\infty} \alpha_i^2(x,u) < \infty, \forall x, u, \tag{5.51}$$

the tabular Q-learning values, $Q_n(x,u) \to Q^*(x,u)$ as $n \to \infty, \forall x, u$ with probability 1—the optimal result given such stochastic conditions [118].

Fig. 5.29 shows the steps to implementing the control agent into industrial control systems for mediating faults in **stable** processes where the above conditions are satisfied. The implementation consists of three phases: i) preliminary training, ii) calibration; iii) online monitoring.

- **Preliminary training:** A seed model of the process is first identified to allow for preliminary control agent training. Simulations using the seed model will establish a baseline performance for control agent. In this step, the control agent will learn to operate the process under a desired performance metric without using the commonly faulty equipment to gain fault tolerance.

- **Calibration:** The simulation-trained control agent will be implemented online where it will operate and adapt to the real process, overcoming any model plant mismatch. The control agent will perform minuscule exploratory moves while online to ensure optimality. Exploratory moves can be tuned by $u_{e,min} \leq u \leq u_{e,max}$, where $u_{e,min}$ and $u_{e,max}$ are the lower and upper bounds of the exploratory actions.

- **Online monitoring:** After sufficient performance is achieved, exploration will be terminated, and the control agent is ready to mediate process faults.

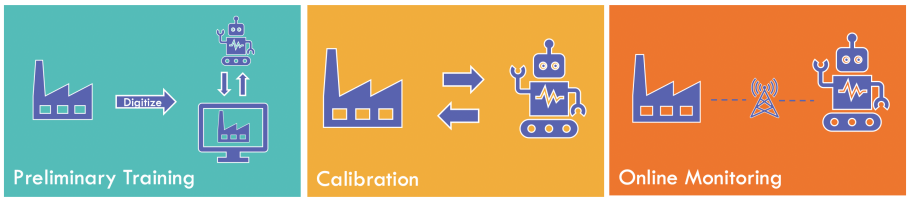


Figure 5.29: Steps on implementing the control agent.

The plant managers may choose to leave the control agent in calibrate mode during a fault so it can continue to identify more optimal control strategies. Such a strategy sounds risky in academia; however, it is indeed how state-of-the-art MPCs are implemented in industry.

5.8.4 Computational Complexity

The computation complexity was decomposed into training complexity and online evaluation complexity. The training complexity refers to the computational time to find the optimal policy. Likewise, the online evaluation complexity is the online evaluation time required to find the optimal input. Assuming *tabula rasa*, the computational complexity to reach the goal state for the first time during training is $O(p^3)$ [119]. Online evaluation complexity is approximately $O(mlog(p))$ and $O(mlog(p) + qlog(q))$ for the prediction and control bandit, respectively. Here, the $O(mlog(p))$ is associated with finding the index of the states using binary search. Similarily, $O(qlog(q))$ refers to sorting the value functions using heap sort to find the maximum value. For traditional optimal control solvers, the computational complexity is $O(N^3(p+q)^3)$, where N is the control horizon [100]. Comparatively, RL evaluates much faster online compared to traditional optimal control methods, but must first be trained. For tasks where long training time is feasible and demand fast online evaluation times, reinforcement learning may be the superior choice. Typically in the process control industry, training models offline is not a significant downfall; however, online evaluation time is incredibly scarce especially in highly complex plants due to hardware limitations. Therefore in terms of computation, RL may be the desired method.

5.9 Case Study

The proposed FTCS was simulated on a distillation tower to illustrate the pros and cons compared to traditional methods. Distillation towers are integral units in industrial processes that require the separation of mixtures of different components into products based on their relative volatility. Heavy oil upgrading facilities utilize distillation towers to separate feed mixtures into various products based on their specific gravity. For many chemical plants, the distillation tower can account up to 50% of the total operating cost, making optimization of the distillation tower a low hanging fruit for cost savings.

Flooding is a common and costly problem in industrial distillation towers. Flooding occurs when liquids are entrained in the vapour due to abnormally high vapour flow rates. Moreover, the excess pressure also causes liquid holdup in the higher plates of the distillation tower. Ultimately, this leads to significant reduction in separation efficiency causing a loss in production, wasted energy, and off-spec products. Flooding commonly occurs when the distillation tower heats up uncontrollable; typically due to actuator faults. In this case study, the proposed FTCS will be applied to the Woodberry distillation tower under different actuator faults. The FTCS will reside in the supervisory control layer, outputting recommended set-points for the regulatory controllers.

5.9.1 Process Description

Distillation is the process of separating a liquid or vapour mixture of two or more components into desirable purities through the addition or removal of heat. The fundamental theory of distillation is that low boiling point components are richer in the vapour of a boiling mixture, while the liquids would contain more of the less volatile components [120]. Liquids exit the bottom of the distillation tower and is sent to a reboiler, where heat is added to vaporize any straggling high volatility product to ensure maximum separation. Similarly, vapour from the top of the tower is sent to a condenser, where heat is removed and additional low volatility

components may be recovered. The condensed vapour is collected in the reflux drum, and will be recycled back into the distillation tower. Typically, distillation columns are large vertical drums with evenly spaced trays to enhance separation of the vapour and liquid components [121]. The tower is separated into two sections. The rectifying section is located between the feed tray and the top of the column and aims to concentrate light components in the vapour phase. Moreover, the stripping section is located between the feed tray and the column bottom and is used to concentrate the heavier components in the liquid phase [122].

The Wood-Berry distillation tower, shown in Fig. 5.30, contains one feed stream and two outlet streams. The feed stream containing methanol and water is characterized by the inlet mass composition Z_f. Methanol has a boiling point of 64.7 °C whereas pure liquid water has a boiling point of 100 °C [123], thus, making methanol the distillate and water the bottoms product. The control inputs are the reflux and steam flow rates, R (lb/min) and S (lb/min). Furthermore, the outputs are characterized by the distillate and bottoms methanol mass fraction, X_D and X_B, respectively. Objectively, the distillation column aims to achieve 100% X_D, while maintaining X_B at 0%. Additional detailed information about the operation and inner workings of distillation towers can be found in [122].

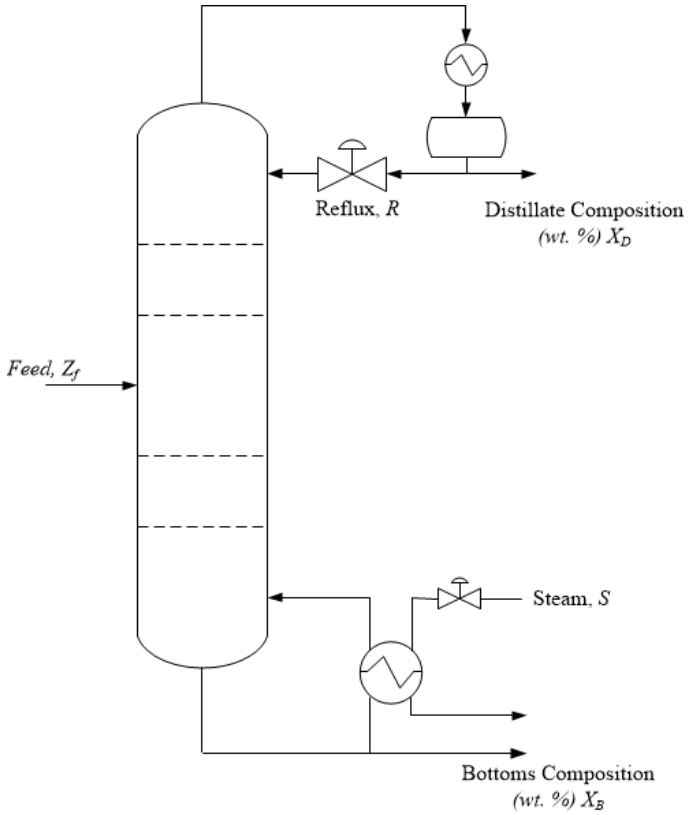


Figure 5.30: Wood-Berry distillation tower schematic.

The transfer function realization of the Wood-Berry distillation tower is given by Equation (5.52) [124].

$$\begin{bmatrix} Y_1(s) \\ Y_2(s) \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix} \tag{5.52}$$

where u_1 and u_2 are R and S, respectively. G_{ij} are:

$$\begin{aligned} G_{11} &= \tfrac{12.8e^{-s}}{16.7s+1} \quad G_{12} = \tfrac{-18.9e^{-3s}}{21s+1} \\ G_{21} &= \tfrac{6.6e^{-7s}}{10.9s+1} \quad G_{22} = \tfrac{-19.4e^{-3s}}{14.4s+1} \end{aligned} \tag{5.53}$$

Equation (5.53) was converted into state space form using the ss function in MATLAB and given by:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{bmatrix} = \begin{bmatrix} -0.06 & 0 & 0 & 0 \\ 0 & -0.09 & 0 & 0 \\ 0 & 0 & -0.05 & 0 \\ 0 & 0 & 0 & -0.07 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + I \begin{bmatrix} u_1(t-1) \\ u_1(t-7) \\ u_2(t-3) \\ u_2(t-3) \end{bmatrix} + I \begin{bmatrix} \tilde{u}_1(t-1) \\ \tilde{u}_1(t-7) \\ \tilde{u}_2(t-3) \\ \tilde{u}_2(t-3) \end{bmatrix} \tag{5.54}$$

$$\begin{bmatrix} X_D \\ X_B \end{bmatrix} = \begin{bmatrix} 0.8 & 0 & -0.9 & 0 \\ 0 & 0.6 & 0 & -1.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \end{bmatrix} \tag{5.55}$$

where I is the identity matrix and $\tilde{u}(t)$ denotes actuator faults. Initially, the system was at steady state where $X_D, X_B = 100, 0$ and initial states $x_0 = [251, 0, 103, 0]$. Measurement noises, ε_i, were sampled from $\varepsilon_i \sim N(0, 2)$. Applying the Popov-Belevitch-Hautus test to the system, $rank([B, AB, A^2B, A^3B]) = 4$, satisfying the controllability criterion [57]. Furthermore, it can be seen that X_D and X_B are controllable using either u_1 or u_2. Thus, even if one controller is faulty, the non-faulty controller can still guide one system output to the desired set-point. Finally, the system matrix contains only negative eigenvalues; therefore, the system is globally asymptotically stable with the region of attraction spanning the entire state space. Given a constrained input, the control agent in this study is guaranteed to be stable under any policy.

5.9.2 Tuning of Regulatory Control

Proportional-Integral (PI) controllers were used for regulatory control because its performance exceeds Proportional-Integral-Derivative (PID) controllers in the Wood-Berry distillation tower due to the slow dynamics of the system [109]. The discrete PI controller formulation is [57]:

$$u_t = u_{t-1} + K_p(e_t + e_{t-1}) + K_i e_t \tag{5.56}$$

where K_p and K_i are the proportional and integral parameters that must be tuned. A multi-loop tuning strategy using equivalent transfer functions and simplified internal model control was used to tune the PI controllers [125]. The controller parameters are given in Table 5.18:

Table 5.18: Parameters for the PI controllers

	u_1	u_2
K_p	1.31	-0.28
K_i	0.21	-0.06

5.9.3 Fault-Tolerant Control System

Integral wind-up is a common problem in PI controllers during actuator saturation or faults. Amid these events, the integral term accumulates a larger error, often resulting in excessive overshooting and irresponsiveness to errors in the opposite direction [57]. In this study, the prediction agent learned faults through large integral wind-ups. The states of the prediction agent is $\mathcal{X} = [\Delta y_1, \Delta y_2, \Delta u_1, \Delta u_2]$. The prediction agent will learn typical $\Delta y_{1,2}$ pairings with $\Delta u_{1,2}$; if large $\Delta u_{1,2}$ is observed without an equal change in $\Delta y_{1,2}$, a fault is deemed active.

Fig. 5.31 shows the normal and faulty controller input-output pairing for u_1 and X_D. Points within the dashed circles are expected states from the closed-loop system. Any points residing outside are faulty. A similar relation exists with all other input-output pairings. The prediction agent will deemed a fault active when ten consecutive points fall outside the boundaries. This condition was imposed to prevent false alarms caused by noisy process data.

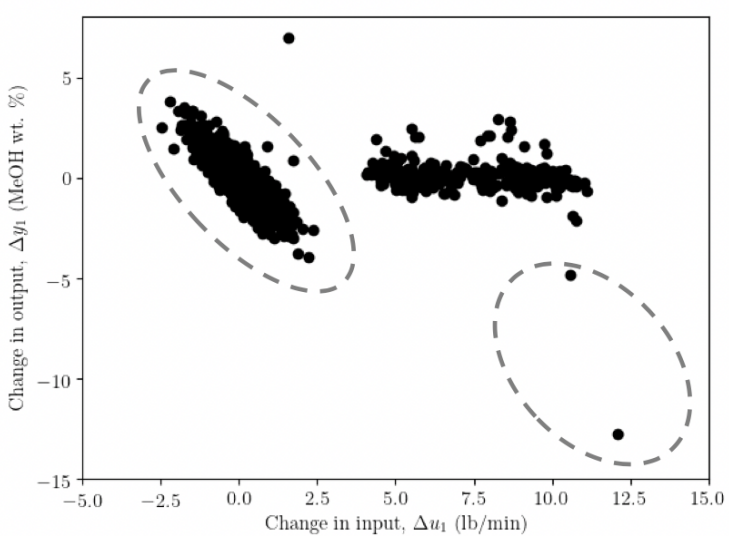


Figure 5.31: Relationship between X_D and u_1.

After a fault is identified, the control agent is activated to guide the system back to the fault free case, if possible. In this study, the control agent's state and actions are:

$$\mathcal{X} = [(-15, 15), (-15, -14), ..., (15, 15)]_{31^2 \times 1} \tag{5.57}$$

$$\mathcal{U} = [(-10, -10), (-10, -9), ..., (10, 10)]_{21^2 \times 1} \tag{5.58}$$

Initial learning rate a_0, discount factor β, and exploratory factor c were 0.5, 0.1 and 1.2, respectively. α is decayed as the agent gains experience, given by:

$$\begin{cases} \alpha_t = a_0, & N(x,u) < 25 \\ \alpha_t = \frac{\alpha_0}{1+N(x,u)}, & N(x,u) \geq 25 \end{cases} \tag{5.59}$$

where $\alpha_t \in [0.001, 0.5)$. The reward, $|R| \leq 900$, is bounded and given by Eq. 5.46. Learning rate decay and bounded reward are necessary for RL convergence [118].

5.9.4 Case Studies

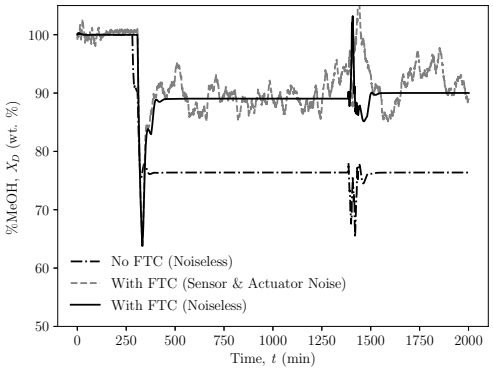
Table 5.19 shows the four case studies that were explored. The prediction and control agents were trained in simulation for 320,000 training steps for each case. A random actuator fault was introduced at the 150^{th} minute. τ_{max} was set to 30

minutes. Each episode was limited to a maximum of 2000 minutes before the system was reset. The PI controllers were evaluated every 4 minutes.

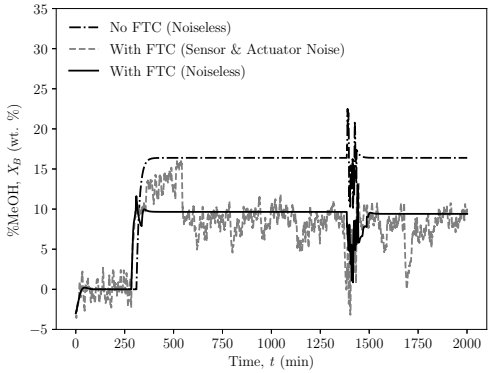
Table 5.19: Case studies for the FTCS

	Reward	Description
Case 1	- $e^2_{X_D}$	Set-point Change
Case 2	- $e^2_{X_B}$	Set-point Change
Case 3	- $0.8e^2_{X_D}$ - $0.2e^2_{X_B}$	Optimal Operation
Case 4	- $e^2_{X_D} \rightarrow -e^2_{X_B}$	Adaptation

The case study simulation results are shown in Figs. 5.32a, 5.32b, 5.33, and 5.34. In case 1, the operator changed the set-point from 100% to 90% for X_D at $t = 350$; however, the reflux valve became stuck. The FDS detected ten consecutive anomalous $(\Delta y_{1,2}, \Delta u_{1,2})$ pairs and activated the FTC which guided the system to the desired set-point successfully within 60 minutes. The FTC can also reject disturbances as shown at $t = 1400$. Without the FTCS, the system would have been stuck at $X_D = 76$. Additionally, the system is robust to large process uncertainty (as shown in the noisy measurements). Likewise, Fig. 5.32b shows a similar scenario for X_B where the steam valve became stuck. By training the FTCS for faults in X_B, the system can be easily re-stabilized using the *same* algorithm and hyper parameters.



(a) Fault in the reflux valve (Case 1).



(b) Fault in steam valve (Case 2).

In case 3, both X_D and X_B are considered at different degrees, with X_D being valued at four times greater than X_B. During operations, an actuator fault occurred

in the reflux valve, significantly impacting both X_D and X_B. With the remaining actuator, the system cannot be guided to the optimal set-points for both X_D and X_B. Here, the agent found an operating condition to minimize the overall loss.

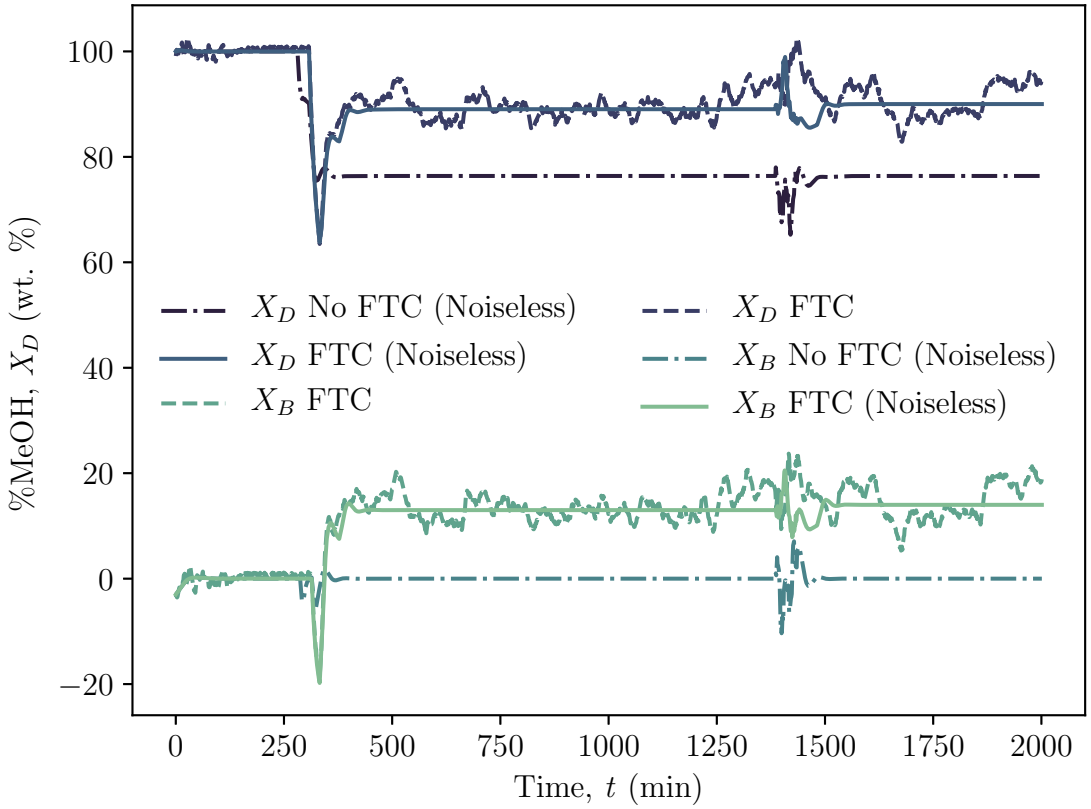


Figure 5.33: Trade-off between conflicting objectives (Case 3).

Case 4 was used to explore adaptability of the FTCS. A pre-trained control agent for regulating X_D to 100 was re-tasked to regulate X_B to 0. In Fig. 5.34, a fault occurred at $t = 300$ in the reflux valve. Originally, the control agent was tasked with using the steam valve to regulate X_D back to 100; however, the operating objective changed to regulating X_B to 0. Here, the control agent was able to completely adapt to the new operating objective in 90,000 training steps by solely experiencing the new reward function. Adaptation speed can also be controlled by tuning learning rate α.

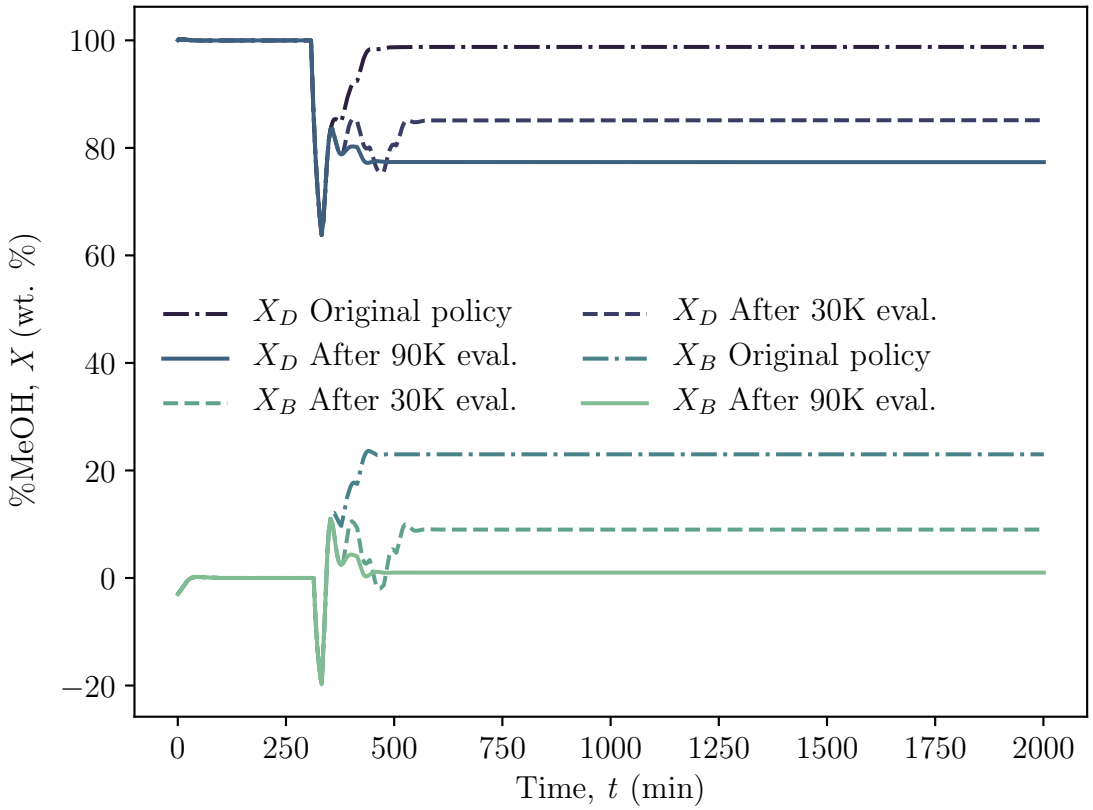


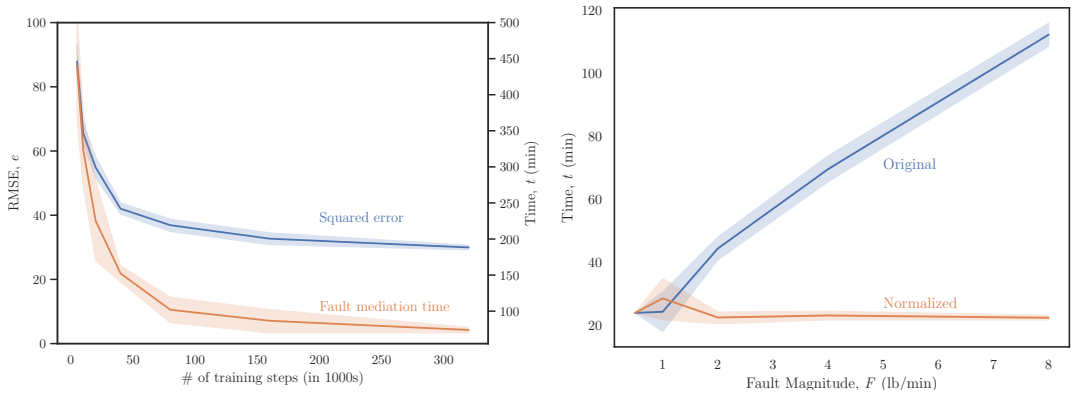
Figure 5.34: Adaptation of the FTCS (Case 4).

5.9.5 Learning Speed and Fault Mediation Time

Fig. 5.35b shows the time required to mediate faults of different magnitude. The fault mediation time was calculated as $t_s - t_f$, where t_s is when the control agent made its first action and t_f is when set points returned to 98% of its original values.

From Fig. 5.35b, the time required to mediate a fault increased linearly with magnitude; however, this was caused by larger magnitude faults requiring additional actions from the control agent. Moreover, the mediation time became constant after being normalizing by the minimum number of actions required to mediate the fault; the expected behaviour for linear systems. The control agent's actions can be increased to reduce mediation time during high magnitude faults. Variance was higher at smaller magnitudes due to noise being more dominant.

Fig. 5.35a shows the control agent's performance for mediating a constant fault after different training steps. It can be seen that the mean performance does not greatly increase after 160,000 training steps; however, variance of results reduce significantly until 320,000 training steps.



(a) Fault mediation time and error accumulation vs. # of training steps averaged over 30 episodes. Shaded regions correspond to one standard deviation.

(b) Time required to mediate faults of different magnitudes averaged over 1000 simulations. Shaded regions correspond to one standard deviation.

5.9.6 A Comparison of Optimal Control

In this section, the performance of the RL-FTCS was compared to MPC. A variety of factors relating to industrial implementation were explored for both MPC and RL under different situations. Factors include: performance, computational time, sensitivity to tuning, time required to implement, and robustness.

Figs. 5.36a and 5.36b show the X_D trajectories under different control strategies during a fault. The strategies provided are:

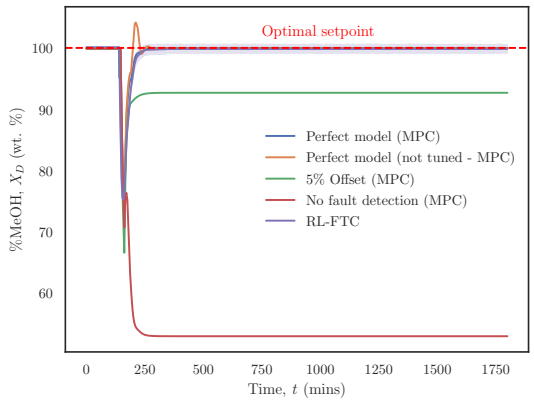
1. Classic MPC with no fault detection
2. MPC equipped with the proposed FDS to detect faults
   (a) using a perfect model
   (b) using a perfect model with un-tuned weighting matrices
   (c) MPC using a model with 5% mismatch
3. RL-FTCS

For the MPC control strategies, the internal states of the system are all assumed to be measurable. The MPC cost function is given by:

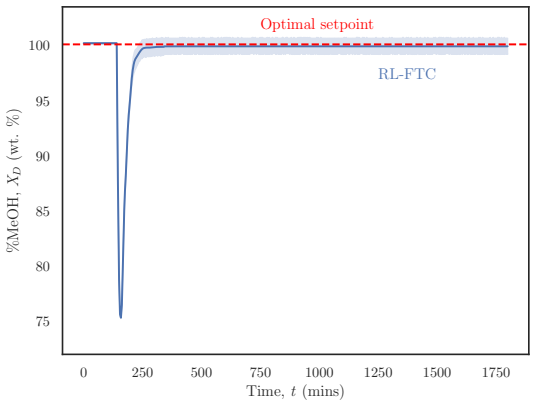
$$J = \sum_{i=1}^{\infty} \gamma^i x_i^T Q x_i \tag{5.60}$$

$$x_i = x_i - x_i^{sp}$$

where i denotes the stage number. $\gamma = 0.9$ is the discount factor to decay future costs; a strategy RL uses to emphasize near-term performance. Here, it was added to the MPC's cost function to ensure the objectives of both controllers are identical. The control and prediction horizons for the MPCs are ∞; however, stage costs beyond $i = 50$ are decayed by 99.5% due to γ. The MPC weighting matrix, $Q_{m\times m}$, is an identity matrix. In the un-tuned MPC case, $Q_{m\times m}$ is a random diagonal matrix. Furthermore, the inputs of the MPC are bounded by $|u_i| \leq 10$, an identical condition imposed on the RL controller. Overall, the MPC's objective was designed to be an exact replica of the RL's reward function to ensure both controllers are solving identical problems.



(a) Trajectories of X_D under different control strategies during a constant reflux valve fault. Shaded region correspond to one standard deviation.



(b) RL-FTC performance during a reflux valve fault averaged over 30 simulations. RL-FTC was trained on a model with 5% offset.

In this simulation, a fault occurred in the reflux valve at $t = 350$, causing a major disturbance in X_D. For the MPC without fault detection, X_D dropped drastically, and never recovered. With the FDS equipped, the MPC with the **perfect** model

was able to recover to pre-fault conditions very rapidly; however, the same MPC with a poorly tuned Q matrix results in overshooting and sub-optimal performance. Nevertheless, the fault was still rejected. However, for a MPC using a model with 5% offset (all values in the A matrix are increased by 5%), the MPC had a large offset and was never able to achieve pre-fault conditions due to the optimal trajectory calculated by the MPC being heavily reliant on the model itself.

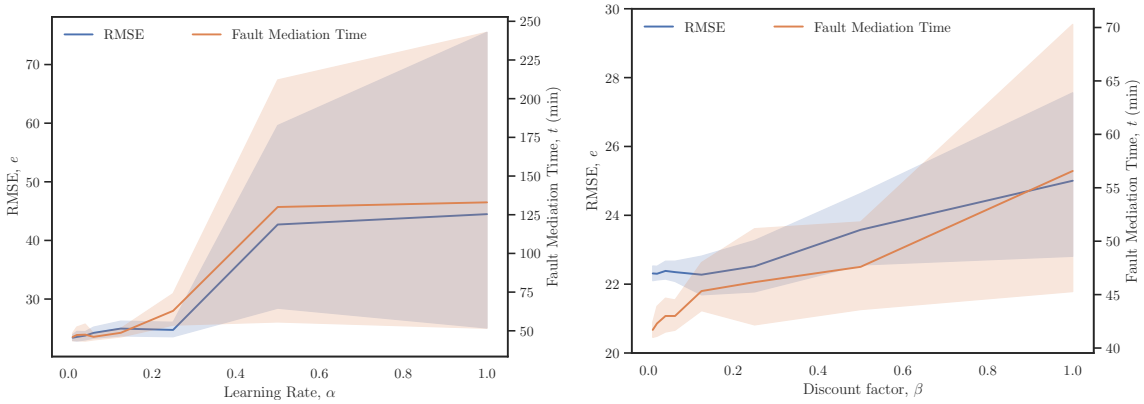
Here, RL can overcome this problem through the velocity implementation style and its *model-free* nature. The RL in Fig. 5.36a is trained on the 5% offset model. But, RL uses the model only for an initial policy. Afterwards, real-time feedback for RL is obtained in terms of an tracking error, e_t from the plant, and does not rely on the initial model for any control purposes. Taking e_t, RL will perform control action Δu with accordance to its current policy. Through this, RL was able to reject faults, even when trained on inaccurate models. Additionally, RL will update its policy online to adapt to process drift, and continue to improve.

Table 5.20 contains the performance metrics for the controllers shown in Fig. 5.36a. MPC with a perfect model is still the superior choice, resulting in the lowest RMSE and fault mediation time. On the contrary, if the weighting matrix is improperly tuned, the MPC's performance can suffer even using the perfect model. RL (trained on the 5% offset model) has higher RMSE and fault mediation time compared to MPC with a perfect model; however, RL performs better than all other MPC implementations, and will continue to improve when implemented online.

Table 5.20: Performance metrics for fault mediation using different control strategies.

		RMSE	Mediation Time (mins)
MPC	Perfect Model	21.6	46
	Perfect Model (un-tuned)	22.4	66
	5% Offset	N/A	∞
	No Fault Detection	N/A	∞
RL	RL-FTC	22.4	42

RL is also less prone to poor tuning as shown in Figs. 5.37a and 5.37b, and only requires the output of the system for control. For RL, the only hyper parameters that require tuning are α_{min} and β. But from Figs. 5.37a and 5.37b, it can be seen that RL is quite robust to poor hyper parameter tuning; a trait not exhibited by traditional optimal controllers. For α_{min}, any values below 0.25 results in relatively the same performance. Likewise for β, any values under 0.5 results in similar performance, with higher values only slightly depreciating performance. Nevertheless, higher values result in significantly higher variance.



(a) RMSE and fault mediation time as a function of different fixed α during control agent training.

(b) RMSE and fault mediation time as a function of different fixed β during control agent training.

Figure 5.37: Performance vs. α and β. Solid line represent average performance of 10 different agents. Shaded area represents one standard deviation.

The computational time of RL was also compared to MPC using IPOPT's linear

program. For a simulation lasting 2000 minutes, RL completed the simulation in 2.0 ± 0.2 **milliseconds** while MPC required 4.5 ± 0.1 seconds.

Table 5.21 summarizes the performance of RL-FTC compared to traditional optimal controllers under actuator faults.

Table 5.21: Summary of RL-FTC compared to MPC.

	RL	MPC
[1]RMSE	22.4	21.6
[1]Mediation time (mins)	42	46
[2]Computational time (s)	0.002 ± 0.0002	4.5 ± 0.1
Sensitivity to tuning	No	Yes
Robustness	Yes	Bad models cause offset
Online calibration	Exploratory moves	Exploratory moves
Requires offline training	Yes	No

[1]Lowest value achieved across all different simulations. For RL, this value is the average of at least 10 simulations to ensure reproducability.

[2]Computational time required to run the Wood-berry distillation for 2000 minutes.

5.10 Concluding Remarks on the FTCS

Eventually, all process equipment will reach the end of their operational lifetime and fail. Such failures are difficult to predict and may cause catastrophic damage; therefore, it is advantageous to proactively manage risks using a fault-tolerant control system (FTCS). This study proposed a general FTCS for continuous MIMO systems using reinforcement learning (RL). The FTCS was placed in the supervisory control layer and gave operating recommendations to process control systems. The FTCS was simulated distillation tower, showing its fault tolerant nature, robustness to uncertainties, and disturbance rejection capabilities all while being adaptive. The system was also evaluated from an industrial implementation perspective and compared to traditional optimal control methods similar to RL. Unsurprisingly, MPC was found to be the superior method if a perfect model was provided and sufficient computational time was given. However, MPC falls short during scenarios with model plant mismatch, or if the controller is poorly tuned. RL's performance is only slightly worse than MPC, and is robust to model plant mismatch due to its

model-free nature and velocity implementation style. Ultimately, RL may be the preferred method in an industrial environment where hardware is lacking, engineers being under time pressure to create solutions, and/or fast computational time is necessary.

Chapter 6

Review of RL for Process Control

This chapter starts with a literature review on the most famous RL applications in history. Applications in this section have all been widely covered by the media and are well known amongst researchers and industrial practitioners alike. Then, a review on RL applications catered towards the process control industry will be provided. Finally, this chapter is concluded with an impressive application of RL for power optimization. The main contribution of this chapter is the conducted literature view.

6.1 Renowned triumphs

The world witnessed, for the first time, artificial intelligence learning and playing games with a mere camera placed in front of the computer screen [49], [50] in 2013! The algorithm proposed was named the deep Q-learning network (DQN). In this application, a general RL agent *successfully* conquered various ATARI games using the camera images alone. However, such games are simple near-deterministic environments with sufficiently small state and action spaces, allowing even rules-based methods to be near-optimal (though previous algorithms did not *learn* the games, nor can they play multiple games with the same algorithm). Although DQN showcased the power and generality of DQN, previous methods were already near-optimal in such simple environments. To conquer a task never done before by computers,

Google DeepMind developed AlphaGo in 2016. AlphaGo was an RL algorithm built to conquer Go, a 2-player board game invented 3,000 years ago in China [126], [127]. Go is widely known as a near-impossible game for machines due to the dimensions of its state and action space (over 10^{170} possible states, a googol times larger than Chess), and the requirement to defeat opponents with stochastic strategies. State-of-the-art Go programs struggle against even amateur players; however, AlphaGo decisively defeated Ke Jie, the world's best Go player. The structure of AlphaGo employs *value networks* to evaluate the board state. Then, *policy networks* are used for optimal action selection. During initial training, the agent used supervised learning to gain fundamental knowledge from amateur level play. Afterwards, advanced strategies were developed by learning from expert level play. After surpassing the experts, the agent continued to *perfect* itself through conducting playing against itself, ultimately evolving into the world's best Go player in history [126], [127]. In terms of real world applications, these experiments demonstrate the potential of RL to identify new techniques and insights to advance modern engineering beyond what is already known.

Originally, AlphaGo contained human engineered features that were initially believed to enhance the agent's learning speed. Ironically, DeepMind thought the complete opposite. Instead, DeepMind believed that the features actually handicapped the agent's skill ceiling, leading to the development of AlphaGo Zero (zero refers to zero engineered features), a more natural version of AlphaGo that is free of human intervention [128]. In AlphaGo Zero, the states were simply the locations of the black and white stones. In terms of the algorithm, AlphaGo Zero combined the value and policy networks into one network, making it a more simple algorithm. After training for approximately 40 days starting *tabula rasa*, AlphaGo Zero was able to surpass AlphaGo through pure self-play, without human engineered features or learning fundamentals from human play. Furthermore, just 3 days was needed for AlphaGo Zero to achieve world championship level (i.e., the level required to decisively defeat Ke Jie). AlphaGo Zero was also much more efficient, using only 4

tensor processing units (TPUs) compared to the 48 used by AlphaGo resulting in a 90%+ reduction in energy usage.

In the latter half of 2017, AlphaGo Zero was perfected into AlphaZero, a general RL algorithm capable of teaching itself Chess, Shogi, and Go. Additionally, the agent was ultimately able to defeat the world champion program in each respective case [129]–[131]. Architecturally, AlphaZero uses a deep neural network $(\mathbf{p}, v) = f_\theta(x)$ where $\mathbf{p}$ represents a vector of action probabilities $p_u = Pr(u|x)$, θ are the parameters of the neural network, and $v \approx \mathbb{E}[z|x]$ where z denotes the expected game outcome [131]. For example, z = -1 for a loss, 0 for a tie, and 1 for a win. Magnus Carlsen, the world's best Chess player in history, had a peak FIDE ELO (skill evaluation assigned by FIDE, the world's most prestigious Chess organization) of 2882. In Chess programs using supervised learning to replicate Mr. Carlsen's playstyle, the ideal agent would be hard capped at 2882, a level representing zero replication error. Comparatively, AlphaZero achieved an ELO above 3300 from pure self-play in just 200,000 training steps. In 300,000 training steps (4 hours physical time), AlphaZero confidently surpassed Stockfish, the best Chess engine [132]. Comparing AlphaZero with Stockfish, Stockfish required *decades* of careful engineering and refinement by Chess and software experts. AlphaZero started knowing literally nothing, and after 4 hours of self-play, it was crowned the best Chess player in history. The most impressive accomplishment of AlphaZero in a RL literature contribution sense is the demonstration of RL's ability for long-term decision making. That is, AlphaZero played Chess like no other. The agent started by sacrificing many pieces in the early game to eventually obtain a significant advantage in the end game, some thirty steps in the future. Furthermore, AlphaZero only needs to search 10^4's moves per turn compared to traditional Chess engines, like Stockfish, where up to 10^7's moves are searched (over 1000 times more!). More impressively, AlphaZero was then used to learn Shogi and Go as well. Ultimately, the agent was able to defeat the respective best game engines, Elmo and AlphaGo Zero.

The achievements of DQN, AlphaGo, AlphaGo Zero and AlphaZero are all tech-

nologically amazing; however, all previous applications hold no true value in the real world. More specifically, the algorithms were all applied in a *perfect information* system where all system states are perfectly observable and without stochasticity. For example, you cannot keep the location of your pieces hidden from your opponent in Chess. Additionally, the applications did not require real-time decision making. Instead, the computers were given excessively long periods of time to provide an action. Comparatively, systems in the real world occasionally contain fast dynamics and often contain unobservable, and/or unreliable information. To demonstrate RL's ability to perform in stochastic and partially observable settings similar to the real world, AlphaStar was developed [133]. Here, the agent learned to play StarCraft II, a *real time* strategy game where the player acts as the general of an army. Here, the agent is tasked with optimally allocating sufficient resources for military and resource-generation needs in order to defeat the opponent. Compared to other games, StarCraft is a very difficult (most humans cannot properly play it), real time, and the opponent's moves are *hidden* and, often times, *stochastic* because the opponent is stochastic. The state and action spaces are also nearly infinite because of the wide range of available choices (much like a military general's job in real life). Here, the agent must respond and act fast enough to win real time battles while also managing the long term resource requirements of its army. In the past, ML methods were applied on *simpler* real time games such as Mario or Quake with heavy simplifications. Even with such modifications, no algorithms ever performed even remotely close to professional level play. In AlphaStar's case, the agent *decisively* defeated two of the best StarCraft II pros using pixel inputs alone and on the full game (with no modifications). Moreover, AlphaStar was not given any additional hidden information and was also constrained to be inline with human capabilities (e.g., the agent is not allowed to perform thousands of actions per second, etc.). Through AlphaStar, RL was shown to have the ability to react to unexpected situations in real time high dimensional environments. Additionally, RL was very successful in hierarchical long-term planning tasks, as shown by its abil-

ity to manage long-term resource needs. Such characteristics are vital in industrial process control, especially in applications regarding fault-tolerant control or high dimensional multi-variate optimal control.

AlphaStar used an *off-policy* actor-critic RL algorithm. Initially, AlphaStar learning fundamental strategies of StarCraft using supervised learning from previous game footage because the game is too difficult to learn *tabula rasa*. Afterwards, it conducted self play to perfect itself.

All applications above assumed a single agent environment. In industrial process control, the agent must also identify the consequences of its actions on the entire process. RL's capabilities in multi-agent partially observable settings was first confidently demonstrated by OpenAI on a game known as Defense of the Ancients (DotA) 2. DotA, like StarCraft II, is a real time high dimensional strategy game (more commonly known as multiplayer online battle arena) where each *team* tries to overcome the opponent. Unlike StarCraft, five players are on each team for DotA and all players must work together. In such a setting, the agent's interaction effects with other agents must be explicitly considered to identify the optimal policy. In DotA, the time horizon per game is also dramatically increased and can be up to 80,000. Comparatively, a game of Chess or Go typically ends within 150 turns [134]. In OpenAI Five, all agents use the proximal policy optimization algorithm and handles the system's partial observability using recurrent neural networks. At each time t, 20,000 continuous observations are provided to the agent. The agent then picks 1 action out of 1,000 different actions. The agents' reward function contains two parts: individual performance and team performance. The team performance reward function was used to enhance cooperation among the independent agents. In the reward function, a hyper parameter called *team spirit*, denoted here as ϕ, was used to imply the importance of individual and team performance. Throughout each game, team spirit was annealed from 0 to 1 to establish that in the end game,

only team performance matters. The reward function for each agent is:

$$r(x,u) = \phi \cdot \text{team reward function} + (1-\phi) \cdot \text{individual reward function} \quad (6.1)$$

On April 2019, OpenAI Five defeated the world's highest ranked DotA 2 teams [135].

Modern RL debuted in near-deterministic low dimensional video games, eventually transitioning to more complex systems that reflected the uncertain, unobservable, and stochastic nature of the real world. Throughout all these modern RL triumphs, RL agents was shown to have capabilities to optimally handle partially observable, long horizon, and high dimensional systems (better than humans). Additionally, RL has fast online evaluation times, allowing for quick reactions to unexpected situations. RL can also learn the optimal policy in multi-agent systems and is feasible for real time applications with exceptionally fast dynamics. Most critically, RL was shown, time and time again, to be a general algorithm with the ability to learn different things. Such a characteristic could significantly reduce R&D costs for advanced applications in industrial process control.

6.2 Simulated RL Applications

6.2.1 RL for Adaptive PIDs

Initially, senior management might hesitate to implement RL for direct closed loop control due to safety concerns. To gain initial approval, this section introduces RL methods to optimally tune PIDs. Proportional-Integral-Derivative (PID) controllers are widely used throughout many processes due to their simplicity, effectiveness, and ease of implementation. The general PID formulation is given by [57]:

$$u(t) = K_p\varepsilon(t) + K_i \int_{\tau=0}^{T} \varepsilon(\tau)dt + K_d\dot{\varepsilon}(t) \quad (6.2)$$

where $\dot{\varepsilon}(t)$ denotes the change in error at time t. K_p, K_i, and K_d are the PID parameters corresponding to the proportional, integral, and derivative gain, respectively. These parameters must be *well tuned* for optimal controller performance; however, the tuning process is time-consuming, especially in MIMO systems with many interacting control loops (i.e., proper tuning of one control loop ends up de-tuning another). Many traditional approaches exist for PID tuning. One popular method is the Ziegler-Nichols method. But PIDs tuned using these general approaches typically perform well below optimal [136]. Here, RL agents will be used to automatically and optimally tune the PID parameters, resulting in superior performance while reducing engineering work hours.

Perhaps the earliest study on automated PID tuning using RL concepts was published in 2000 by [136] (algorithm overview shown in Figure 6.1). Application-wise, the authors was able to successfully tune a Ford Motors Zetec engine. The algorithm was called Continuous Action Reinforcement Learning Automata (CARLA), and was typically used for *fine tuning* PIDs after initial parameters were set using methods like Ziegler-Nichols. Controllers tuned using CARLA resulted in a 60% reduction in the cost function compared to traditional tuning methods. CARLA is implemented as follows: Initially, each hyper parameter (K_p, K_i, K_d) corresponds to one CARLA. The output of each CARLA is the recommended new hyper parameter and is picked from the corresponding probability density functions, $f(x)$. For example, in a single PID system, there would exist three CARLAs corresponding to $K_p, K_i,$ and K_d. At each time step, the recommended parameters are outputted and is implemented into the PID. Afterwards, the cost function using the recommended parameters is evaluated. Costs lower than the mean will shift the distribution towards the recommended parameters, vice versa for higher costs. Exploration-wise, Gaussian white noise is added to the recommended parameters. Lastly, CARLA is typically applied to low dimensional settings due to its non-scalability nature. For more detailed information regarding CARLA, see [136].

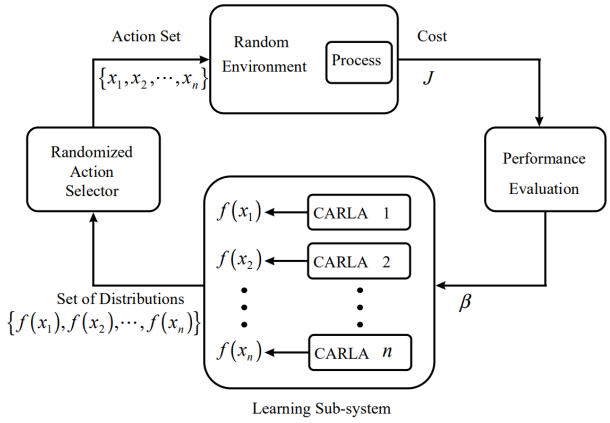


Figure 6.1: CARLA: An RL-powered automatic PID tuning algorithm

Six years later, [137] developed a more sophisticated PID tuning approach through an actor-critic algorithm. Here, the agent's states are given as:

$$x = [\varepsilon_t, \Delta\varepsilon_t, \Delta^2\varepsilon_t]$$

$$\Delta\varepsilon_t = \varepsilon_t - \varepsilon_{t-1}$$

$$\Delta^2\varepsilon_t = \varepsilon_t - 2\varepsilon_{t-1} + \varepsilon_{t-2}$$

and the actions are:

$$u = [K_i, K_p, K_d]$$

The states correspond to the integral, proportional, and derivative error terms of a discrete PID. Intuitively, the agent maps the current error, and the first- and second-order difference of errors to the optimal PID parameters at each t. Using the new parameters, the PID is re-parameterized and outputs Δu_t using:

$$\Delta u_t = K_i\varepsilon_t + K_p(\varepsilon_t - \varepsilon_{t-1}) + K_d(\varepsilon_t - 2\varepsilon_{t-1} + \varepsilon_{t-2}) \tag{6.3}$$

where K_i, K_p, and K_d are provided by RL and may change for each time t. From Δu_t, u_t is calculated using:

$$u_t = u_{t-1} + \Delta u_t$$

From [137], results showed the algorithm's capability to near-perfectly track complex

non-linear systems. By 2008, [138] applied this algorithm onto an industrial wind turbine experiment, yielding near perfect set point tracking. Applicational details are in [138]. In 2015, the algorithm was implemented on an under-actuated robotic arm [139]. Here, the system contains fast dynamics and lacks sufficient actuators for control. This study explores the RL tuning method's fault tolerant characteristics due to the under-actuated system. To test the PIDs, the robotic arm had to maintain proper formations and was exposed to many disturbances. Traditionally tuned PIDs overshoot and exhibit other undesired behaviours; however, the RL tuned PID showed significantly better performance for disturbance rejection and response time. By 2017, a Q-learning variant of this algorithm was used to tune a race track robot [140]. Compared to PIDs tuned using traditional approaches, the robots tuned using RL achieved up to 59% faster lap times.

In 2013, [140] introduced a new automated tuning strategy to tune robots playing soccer. This new method was comparable to the one used for the multi-PID soccer robot. The main difference was the agent's states. Instead of errors, the agent received the location of the robot in the soccer game. Intuitively, this gave the agent information regarding its current situation within the game, and allowed RL to tune its specifications accordingly. For example, the robot will require faster speed when sprinting down the soccer field compared to when it is ready to score a goal. Such a tuning method may be useful for an event triggered control system in industry. For example, if there is snow outside, the control system should act more conservatively and have less gain compared to normal ambient conditions. In [140], it was demonstrated that the RL tuned robots were vasting superior to robots tuned using the Ziegler-Nichols method.

RL was also used for a model-based PID tuning strategy where the controller's finite horizon cost was considered instead of its immediate tracking error. In the end, this method was highly successful and even worked on non-linear MIMO systems with arbitrary couplings. The method was validated in real life on a robot named Apollo. Apollo had imperfect low-level tracking controllers and unobserved

dynamics, but was able to perform adequately using the RL tuned PIDs. Advanced details regarding this implementation can be found in [141].

In recent literature, many more RL automated PID tuning methods were published, but were not listed here. Ultimately, all the "new" approaches are very similar to the ones presented with only slight alterations of the tuning set-up.

6.2.2 RL in Process Control

Studies where RL agents were applied solely for regulation or set-point tracking are still quite rare due the undeniable success of traditional methods. [142] was the first instance where RL was used for set-point tracking of an industrial process. Here, the authors tracked the set-point of a CSTR using a neural network based agent. In more recent literature, [143] was the first to show deep RL's (DDPG) capabilities in process control. Here, it was shown that RL can successfully control arbitrary SISO and MIMO systems as long as the reward function is properly established. In [143], the agents mapped $x = [y_t, y_{sp}]$ to $u = [u_t]$. Intuitively, the states provided the current tracking error to the agent while the action changed the control input to mitigate the error. In [144], an actor-critic agent was used to regulate the temperature of a building heating, ventilation, and air conditioning (HVAC) system. Ultimately, the agent resulted in a 2.5% reduction in energy consumption while achieving a 15% increase in thermal comfort. The HVAC system was optimized again in [145]. This time, a proximal actor-critic RL agent was used. All previous applications formulated the agent to perform set-point tracking; however, there already exists many highly capable controllers for set-point tracking such as PIDs and MPCs. RL's greatest advantage compared to previous methods are its flexibility and ease of use for optimal control (i.e., optimize an economic objective). More specifically, optimal control can be achieved by simply changing the reward function to be in terms of an economic objective. Furthermore, RLs are also shown to be effective in fault-tolerant control (FTC) [146]. A RL agent trained under different faults can identify a fault-robust optimal policy directly. Moreover, RL's *model-free*

nature and direct adaptive characteristics allow the agents to mediate faults even when trained on inaccurate process models and can also adapt to process drift.

6.2.3 RL for Anomaly Detection

Another field where RL gained traction is *time series* anomaly detection. In industry, anomaly detection is a proactive application to identify potential hazards *before* a loss incident occurs. Compared to previous methods, RL's ability to self-learn provide an attractive edge. [147] was perhaps the earliest paper to introduce RL-based anomaly detection (architecture shown in Figure 6.2). Here, the authors built an adaptive neural network agent to identify cyber threats. Comparatively, the architecture is nearly identical to the time-series anomaly detection introduced in Chapter 3. A POMDP was used to describe the system where the agent mapped observations $o = [x_{t-n}, x_{t-n+1}, ..., x_t]$ to actions $u =$ [Normal , Anomalous], guided by a reward signal based on the success or failure of its last prediction. The reward function is designed as follows: correct identifications yielded +1 reward while misclassification yielded -1 reward. Notice that the observations were augmented past states. By doing so, the agent is provided with time-series information. More recently, Zighra (a online security company) deployed the algorithm from [147] into production through a software called SensifyID. Although the concepts here was originally proposed for networks; the same concept was shown in [146] to work as a general fault detection tool for industrial process control as well.

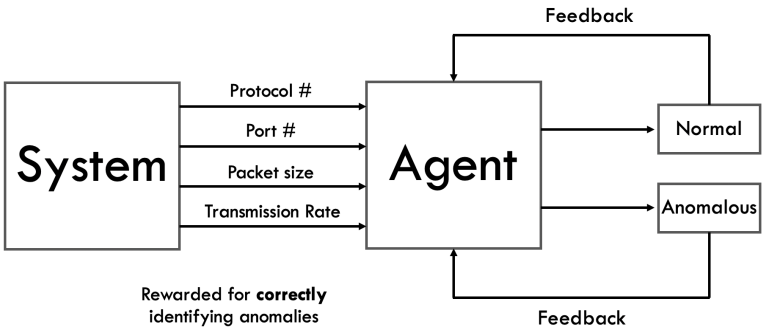


Figure 6.2: A sample anomaly detection architecture.

In early 2010, [148] extended the original RL anomaly detection concepts by

representing the system as a partially observable Markov *reward* process (MRP). The states in this new algorithm remained $o = [x_{t-n}, x_{t-n+1}, ..., x_t]$. The agent had no actions since a MRP was used to represent the system. Instead, the agent learned the probabilities of each state transitioning into an anomalous state. Mathematically, this is given as:

$$P_a(x) = P\{o_{t+1} \in \mathcal{A} | o_t\} \tag{6.4}$$

where $P_a(x)$ represents the probability of transitioning into an anomalous state $a \in \mathcal{A}$ given observation o_t where $\mathcal{A}$ is the set of anomalous states. There exists another hyper parameter, μ_a, that denotes the anomaly threshold. High values reduce false positives but increase false negatives, while low values increase true positives but also increase true negatives. At any time when $P_a(x) > \mu_a$, an anomaly was deemed imminent. The value function of this approach is represented as:

$$V(x) = \sum_{i}^{n} P(o_{t+1} \in \mathcal{A} | o_t) \cdot r(o_t) \tag{6.5}$$

where $r(o_t)$ is the reward received given o_t. If $o_t \in \mathcal{A}$, $r(o_t) = 1$, otherwise 0. Notice here that higher value states have higher chance of being anomalous. As validation, the authors compared the RL anomaly detection algorithm to other popular classification algorithms such as support vector machines. In the end, RL anomaly detection resulted in the highest detection accuracy, although all algorithms scored accuracies above 99.8% on the selected data sets, even linear methods such as logistic regression [148].

In 2018, Huang el al. introduced a recurrent neural network (RNN) based RL anomaly detection algorithm without needing to tune μ_a [149]. The algorithm was similar to [147] where $\pi(u, x)$ mapped states to actions $u = [\text{Normal} , \text{Anomalous}]$. In this representation, μ_a was not required because the classification is binary (i.e., not a probability). Compared to [147], the new algorithm was still a POMDP; however, it used a long short term memory (LSTM) recurrent neural network (RNN) to memorize previous states rather than augmenting the previous states directly.

The performance advantages between the two approaches have yet to be explored in literature, but the LSTM RNN should require significantly more training. The algorithm was validated on the Yahoo anomaly detection benchmark data set [150] and successfully identified all anomalies with no false alarms.

6.3 Google's success story

A world-changing implementation of RL was demonstrated by Google when a RL agent[1] showed the capabilities to autonomously controlling a *live* data center, reducing electricity usage by up to 40%. This also indirectly reduced the carbon footprint of all individuals using Google's services, which encompasses a large part of the world. Google's data centers generate enormous amounts of heat through powering services such as Google Search, Gmail, and YouTube. Hence, the data centers' primary energy usage is for cooling. Cooling industrial processes are accomplished by equipment such as heat exchangers, pumps, and cooling towers—even at Google. Modelling such a complex, non-linear systems poses several difficulties, rendering traditional methods ineffective [151]:

1. Complex, high-dimensional environment with uncountable non-linear interactions rendering modern system identification methods infeasible. Additionally, experienced human operators simply cannot comprehend the countless interactions.

2. Highly dynamic internal and external building conditions (such as ambient temperature, server load, etc.) rendering rules- and non-adaptive methods intractable.

3. All data centers have unique layouts and set-ups. This non-consistency demands custom-tuned models for each individual data center, assuming tradi-

[1]Google DeepMind did not explicitly state the technology used to achieve the savings, only machine learning. However, DeepMind is a company that focuses on reinforcement learning approaches and there were many mentions of creating a general algorithm for all the data centers in the article; therefore, it was assumed reinforcement learning was used. More specifically, meta-RL was most likely used due to the construction of many simulators and the agent's adaptation speed.

tional approaches were used; however, such a dilemma could be adequately overcame through artificial general intelligence where one algorithm can learn many different scenarios.

To overcome these difficulties, DeepMind researchers first identified neural network models corresponding to different operating conditions by leveraging historical operating data from different data centers. The inputs to the neural network models were sensor information such as temperature, pump speeds, ambient temperature, etc. The model output was the power usage effectiveness (PUE) given by:

$$PUE = \frac{\text{Total building energy usage}}{\text{IT energy usage}} \tag{6.6}$$

Here, the neural networks were used as training simulators for the data centers. RL was applied on said simulators to learn a control policy to minimize the PUE. Different agents were trained on different data centers, during different operating conditions. When implemented, the ideal agent would be picked based on the current operating condition. Initially, the control actions provided by the agent were only recommendations. The PUE with and without implementing the agent's recommendations is shown in Figure 6.3.

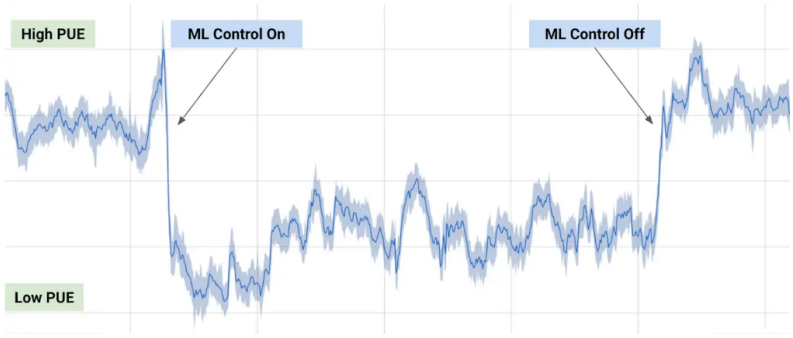


Figure 6.3: Power usage effectiveness with and without ML control. Original figure from [151].

By 2018, the agent was given full access to the data center control system after safety constraints were added.

As a summary, the RL agents sample measurements from the sensors in each

data center every five minutes and outputs the optimal control actions that satisfy a robust set of safety constraints [152]. The local control operators then verify the provided inputs to ensure that the system will remain within constraint boundaries. In the first few months, the agent consistently reduced electricity consumption by an average of 30% and is expected to improve as it continues to learn. In the end, the agent reached an optimal policy that resulted in the lowest PUE ever seen, far surpassing human operation—an event only achievable through RL.

Chapter 7

Concluding Remarks

7.1 Concluding Remarks

Machine learning methods are becoming infinitely more powerful as the world continues its transition to a digital era overloaded with data. Unfortunately, the lack of individuals with knowledge in both the process engineering and machine learning is holding back value creation in the process industry. This thesis introduced simple and economically efficient machine learning algorithms for prediction, monitoring, and control.

Chapter 2 introduced the ML models for prediction—the most widely used machine learning technique. Here, ML is used to predict the expected process variables given other measurements. Applications in the process industry include soft sensors, digital twins, and production forecasting. Combined with mathematical programming, data-driven MPCs and real-time optimization applications can also be built.

Chapter 3 focused on ML applications for safety and risk management. More specifically, anomaly detection/prediction and alarm management applications were introduced. Anomaly detection/prediction applications served as multi-variate alarms, *proactively* identifying deviations in process variables to safeguard people, the environment, company assets, and production capabilities. The ML alarm management system was used to reduce nuisance alarms and provide alarm prioritization during alarm flood scenarios.

In Chapter 4, ML was used for process control and optimal control and was compared to model predictive control. More specifically, the focus transitioned to RL, a branch of ML that can actually surpass previous knowledge and push the boundaries of research. Here, RL was applied for continuous control of processes. Firstly, a tutorial on implementing RL for continuous process control was introduced. Then, RL's optimality was validated against model predictive control, a mathematical programming approach to optimal control. Afterwards, the two methods' pros and cons were compared. To test the limits of RL, it was used in simulation to perform optimal control on an industrial wastewater treatment plant with 145 states and 14 disturbances. The chapter was concluded by showcasing RL's abilities to serve as a general fault detection and fault-tolerant control system.

In Chapter 5, a literature review of RL applications in the process industry was conducted to provide motivation for future innovation. Here, it was discovered that RL can be used for adaptive PIDs tuning, real-time optimal control, and anomaly detection applications.

Table 7.1 shows the pros and cons of reinforcement learning. RL truly shines in extremely fast, non-linear processes where model predictive controllers cannot provide solutions in adequate time. Furthermore, RL also does not need process models after initial training due to its *model-free* nature and can naturally adapt to process drifts. Currently, the biggest disadvantages of RL are simply due to the embryonic nature of the field. In literature, meta-RL studies a set of RLs that can adapt quickly to unseen environments; thus, overcoming the requirement of accurate simulators. Inverse RL began to tackle the reward design issue. Lastly, stability theory and state constraints are being studied in the field of safe RL.

Table 7.1: Most influential advantages and disadvantages of reinforcement learning.

Advantages	**Disadvantages**
Online computation time	Accurate simulator required
Can learn many tasks	Reward design can be difficult
Direct adaptive optimal control	Stability theory lacking
Engineered features not needed	State constraints are difficult

As a final remark, the truly unique characteristic of RL that makes it the closest thing to real artificial intelligence is its *general nature*, allowing for learning of many things through a general algorithm. Although modern RL still faces many challenges, the most interesting fact about artificial general intelligence is that it is *eventually* scientifically achievable. Unlike galactic teleporters or other wild fantasies from science fiction literature, artificial general intelligence is proven to exist, currently within us! The last step is *merely* to reverse engineer human psychology. And when such a task if finally conquered, the concepts reinforcement learning will, *without a doubt*, reside as its central algorithm.

7.2 Future Extensions

7.2.1 RL-MPC - An Unified Approach

In terms of control, one possible future project would be to combine RL and MPC into one unifying algorithm. Currently, RL is a newer field of research and lacks industrial support. Therefore, most plant managers are skeptical of its performance in direct closed-loop control. On the other hand, linear MPCs have many applications in process control but the applications of its non-linear counterpart is still relatively scarce. From an engineering prospective, non-linear MPC is vastly superior because linear processes do not exist in the real world. One factor barring non-linear MPCs from implementation is its much higher computational burden. Mathematically, both linear and non-linear programming are solved in an iterative

approach and the convergence is dependent on the initial guess. By leveraging RL to provide initial guesses to the non-linear MPC, the computational time should be substantially faster, leading to the viability of non-linear MPCs. Theoretically, the initial guess provided by a perfectly trained RL should be exactly the optimal solution, greatly reducing the iterative procedure required by non-linear programming.

7.2.2 Meta-learning in reinforcement learning

Initially, RL agents must be trained on the desired task to obtain optimal performance. Due to the low data efficiency of modern RL algorithms, thousands of interactions may be required before the agent learns something meaningful. Such a requirement is infeasible in real life applications; thus, a representative simulator of the environment is first identified to pre-train the agent in simulation. Then, the agent is implemented to the real process for control and optimization purposes. Without a doubt, there will be off-sets between the identified model and the real process. Hence, RL will also need an initial calibration period to directly adapt its learned policy from the simulator onto the real process. The length of this adaptation period is dictated by the accuracy of the simulator. Unfortunately, there are many processes that are nearly impossible to identify accurately. In such scenarios, the calibration period itself may be in-feasibly long. Meta reinforcement learning is a new field that aims to significantly reduce this calibration time.

In meta reinforcement learning, many *different* simulators of the environments are built to capture model uncertainty. For example in a refinery, one model is built for winter ambient temperatures while another is built for the summer. There could also be different models built for different compositions of crude oil. The goal of the agent is to adapt to new similar environments quickly, even when the exact same task was not trained for. One future project could be to explore training an agent on many models with modeling errors, and then ultimately implementing the agent onto the real system to identify its adaptation speed. If successful, such a application holds many implications for RL in process control because one general

agent could be used to control many different similar systems.

Bibliography

[1] OilPrice.com, *Oil price charts*, `https://oilprice.com/oil-price-charts`, Accessed: 2019-02-03.

[2] D. C. Panagiotis, R. Scattolini, D. M. Pena, and J. Liu, "Distributed model predictive control: A tutorial review and future research directions," *Computers and Chemical Engineering*, vol. 51, no. 5, pp. 21–41, 2013.

[3] R. S. Sutton, A. G. Barto, and R. J. Williams, "Reinforcement learning is direct adaptive optimal control," *American Controls Conference*, 1991.

[4] G. Armstrong and P. Kotler, *Marketing: An introduction*. Upper Saddle River, New Jersey: Pearson Prentice Hall, 2007.

[5] PwC, *Sizing the prize: What's the real value of ai for your business and how can you capitalise?* `www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf`, Accessed: 2019-09-05.

[6] S. Russel and J. Norvig, *Artificial Intelligence: A Modern Approach. 3rd Edition*. Upper Saddle River, New Jersey, USA: Prentice Hall, 2009.

[7] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. 1998.

[8] G. Hinton and T. Sejnowski, *Unsupervised Learning: Foundations of Neural Computation*. Cambridge, Massachusetts, USA: The MIT Press, 1999.

[9] Z. Ge, Z. Song, S. X. Ding, and B. Huang, "Data mining and analytics in the process industry: The role of machine learning," *Institute of Electrical and Electronics Engineers Sensors Journal*, pp. 20 590–20 616, 2017.

[10] D. Mayne and J. B. Rawlings, *Model Predictive Control: Theory and Design. 2nd Edition*. Wisconson, USA: Nob Hill Publishing, 2017.

[11] R. Bellman, *Dynamic Programming*. New Jersey, USA: Princeton University Press, 1957.

[12] M. Mes and A. P. Rivera, *Approximate Dynamic Programming by Practical Examples*. New York, USA: Springer, 2017.

[13] E. Thorndike, "Animal intelligence," *Darien*, 1911.

[14] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, pp. 285–294, 1933.

[15] W. R. Thompson, "On the theory of apportionment," *American Journal of Mathematics*, pp. 450–457, 1934.

[16] H. Robbins, "Some aspects of the sequential design of experiments," *American Journal of Mathematics*, pp. 450–457, 1952.

[17] R. Bellman, "A problem in the sequential design of experiments," *American Journal of Mathematics*, pp. 221–229, 1956.

[18] A. Barto, R. S. Sutton, and P. S. Brouwer, "Associative search network: A reinforcement learning associative memory," *Biological Cybernetics*, vol. 40, no. 3, pp. 201–211, 1981.

[19] M. A. Thathachar and P. S. Sastry, "A new approach to the design of reinforcement schemes for learning automata," *IEEE Transactions on Systems, Man, and Cybernetics*, pp. 168–175, 1985.

[20] Borel, "Les probabilites denombrables et leurs applications aritbmttiques," *Rendiconti del Circolo Matematico di Palermo 27*, pp. 247–271, 1909.

[21] R. Bellman, "A markov decision process," *Journal of Mathematics and Mechanics*, vol. 6, pp. 679–684, 1957.

[22] A. Ng, *Shaping and Policy Search in Reinforcement Learning, Ph.D Dissertation*. California: University of California Berkeley, 2003.

[23] C. Lusena, M. Mundhenk, and J. Goldsmith, "Nonapproximability results for partially observable markov decision processes," *Journal of AI Research*, vol. 14, pp. 83–103, 2001.

[24] S. K. Chenna, Y. K. Jain, H. Kapoor, R. S. Bapi, N. Yadaiah, A. Negi, V. S. Rao, and B. L. Deekshatulu, "State estimation and tracking problems: A comparison between kalman filter and recurrent neural networks," *ICONIP*, pp. 275–281, 2001.

[25] S. J. Bradtke and M. O. Duff, "Reinforcement learning methods for continuous-time markov decision processes," *Neural Information Processing Systems*, pp. 393–400, 1994.

[26] A. S. Poznyak, *Advanced Mathematical Tools for Automatic Control Engineers: Deterministic Techniques*. Oxford, United Kingdom: Elsevier, 2008.

[27] D. Silver, *Planning by dynamic programming*, Class Lecture, COMPGI13, Imperial COllege London, London.

[28] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[29] B. Reinaldo, C. Riberiro, and A. Costa, "Heuristically accelerated q–learning: A new approach to speed up reinforcement learning," *SIBA*, pp. 245–254, 2004.

[30] A. Garivier and E. Moulines, "On upper-confidence bound policies for non-stationary bandit problems," *arXiv:0805.3415*, 2008.

[31] P. J. Huber, "Robust estimation of a location parameter," *Annals of Statistics*, vol. 53, no. 1, pp. 73–101, 1964.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. London, England: The MIT Press, 2015.

[33] B. Grosman, E. Dassau, H. C. Zisser, L. Jovanovic, and F. J. Doyle, "Zone model predictive control: A strategy to minimize hyper- and hypoglycemic events," *NCBI*, vol. 4, no. 4, pp. 961–975, 2010.

[34] H. Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," *Neural Information Processing Systems*, pp. 1–9, 2016.

[35] P. Fleming and R. Purshouse, *Genetic Algorithms for Control Systems Engineering*. 2001.

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

[37] G. Yuan, C. Ho, and C. Lin, "Understanding the difficulty of training deep feedforward neural networks," *International Conference on Artificial Intelligence and Statistics*, vol. 9, pp. 249–256, 2010.

[38] K. He, X. Zhang, and J. S. S. Ren, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *arXiv:1502.01852*, 2015.

[39] A. Ng, *Machine Learning Yearning*. 2018.

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Conference for Learning Representations*, 2014.

[41] L. Bottou, "Large-scale machine learning with stochastic gradient descent," *COMPSTAT*, pp. 177–186, 2010.

[42] A. Y. Ng, "Feature selection, l1 vs. l2 regularization, and rotational invariance," *ICML*, p. 78, 2004.

[43] G. Yuan, C. Ho, and C. Lin, "An improved glmnet for l1-regularized logistic regression," *Journal of Machine Learning Research*, vol. 13, pp. 1999–2030, 2012.

[44] W. Mendenhall, R. Beaver, B. Beaver, and S. Ahmed, *Introduction to Probability and Statistics*. Toronto: Nelson College Indigenous, 2013.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[46] S. Ioffee and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv:1502.03167*, 2015.

[47] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971*, 2015.

[48] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *ICML*, pp. 1–9,

[49] V. Mnih, K. Mavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *Neural Information Processing Systems*, 2013.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Ridjeland, G. Ostrovski, S. Peterson, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[51] L. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, pp. 293–321, 1992.

[52] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv:1511.05952*, 2015.

[53] B. Huang, Y. Qi, and A. Murshed, *Dynamic Modeling and Predictive Control in Solid Oxide Fuel Cells: First Principle and Data-based Approaches*. Berlin, Germany: John Wiley & Sons, 2013.

[54] G. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Physical Review*, vol. 36, 1930.

[55] I. Karatzas and S. E. Shreve, *Brownian Motion and Stochastic Calculus. 2nd Edition*. Berlin, Germany: Springer-Verlag, 1991.

[56] J. M. Knudsen and P. G. Hjorth, *Elements of Newtonian Mechanics: Including Nonlinear Dynamics*. New York, USA: Springer, 2013.

[57] D. Seborg, D. Mellichamp, and T. F. Edgar, *Process Dynamics and Control*. Hoboken, New Jersey: Wiley, 2010.

[58] N. W. Schuck and Y. Niv, "Sequential replay of nonspatial task states in the human hippocampus," *Science*, vol. 364, pp. 1–11, 2019.

[59] N. Khaled and B. Pattel, *Practical Design and Application of Model Predictive Control*. Amsterdam, Netherlands: Elsevier, 2018.

[60] A. Bemporad, *Topic: Explicit MPC: Basics, Fast Implementations, Advantages and Limitations*. Zurich, 2013.

[61] A. E. Huesman, O. H. Bosgra, and P. M. J. Vanhoff, "Integrating mpc and rto in the process industry by economic dynamic lexicographic optimization; an open-loop exploration," *AICHE*, 2008.

[62] M. Ellis, H. Durand, and P. D. Christofides, "A tutorial review of economic model predictive control methods," *Journal of Process Control*, vol. 24, pp. 1156–1178, 2014.

[63] G. D. Souza, D. Odloak, and A. C. Zanin, "Real time optimization (rto) with model predictive control (mpc)," *Computer and Chemical Engineering*, vol. 34, pp. 1999–2006, 2010.

[64] R. L. Mott, *Applied Fluid Mechanics*. London: Pearson, 2014.

[65] S. Firdaus and M. A. Uddin, "A survey on clustering algorithms and complexity analysis," *International Journal of Computer Sciences*, vol. 12, pp. 62–85, 2015.

[66] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *ICLR 2017*, 2017.

[67] M. McCloskey and N. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," *The Psychology of Learning and Motivation*, vol. 24, pp. 109–164, 1989.

[68] S. Grossberg, "Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world," *Neural Networks*, vol. 37, pp. 1–47, 2013.

[69] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[70] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," *CoRR*, vol. abs/1710.06574, 2017. arXiv: `1710.06574`. [Online]. Available: `http://arxiv.org/abs/1710.06574`.

[71] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[72] P. Wawrzynski, "Real-time reinforcement learning by sequential actor–critics and experience replay," *Neural Networks*, vol. 22, pp. 1484–1497, 2009.

[73] V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *AAAI*, pp. 2094–2100, 2016.

[74] J. Cocchio, G. Winkel, and L. White, *ENGG 404: Engineering Safety and Risk Management*. 2015.

[75] R. Barandela, R. M. Valdovinos, J. S. Sanchez, and F. J. Ferri, "The imbalanced training sample problem: Under or over sampling?" *SPR and SSPR*, pp. 806–814, 2004.

[76] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," *International Joint Conference on Neural Networks*, pp. 1322–1328, 2008.

[77] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[78] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," *International Joint Conference on Neural Networks*, pp. 1322–1328, 2008.

[79] C. P. Peng, K. L. Lee, and G. M. Ingersoll, *An Introduction to Logistic Regression Analysis and Reporting*. 2002.

[80] C. Manning, *Introduction to Information Retrival*. 1930, vol. 36.

[81] A. Altmann, L. Tolosi, O. Sander, and T. Lengauer, "Permutation importance: A corrected feature importance measure," *Bioinformatics*, vol. 26, no. 10, pp. 1340–1347, 2010.

[82] D. Becker, *Permutation importance*, `https://www.kaggle.com/dansbecker/permutation-importance?utm_medium=email&utm_source=mailchimp&utm_campaign=ml4insights`, Accessed: 2019-03-02.

[83] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

[84] D. Becker, *Partial plots*, `https://www.kaggle.com/dansbecker/partial-plots?utm_medium=email&utm_source=mailchimp&utm_campaign=ml4insights`, Accessed: 2019-03-07.

[85] D. Becker, *Shap values*, `https://www.kaggle.com/dansbecker/shap-values?utm_medium=email&utm_source=mailchimp&utm_campaign=ml4insights`, Accessed: 2019-03-17.

[86] S. Lundberg and S. Lee, "A unified approach to interpreting model predictions," *Neural Information Processing Systems*, 2017.

[87] B. R. Hollifield, E. Habibi, and J. Pinto, *Alarm Management: A Comprehensive Guide. Second Edition.* International Society of Automation, 2010.

[88] J. Alex, L. Benedetti, J. Copp, K. V. Gernaey, U. Jeppsson, I. Nopens, M. N. Pons, L. Rieger, C. Rosen, J. P. Steyer, P. Vanrolleghem, and S. Winkler, "Benchmark simulation model no.1 (bsm1)," *International Water Association*, pp. 1–62, 2008.

[89] J. Folmer and B. Vogel-Heuser, "Computing dependent industrial alarms for alarm flood reduction," *International Multi-Conference on Systems, Signals & Devices*, 2012.

[90] C. T. Maravelias and C. Sung, "Integration of production planning and scheduling: Overview, challenges and opportunities," *CCE*, vol. 33, pp. 1919–1930, 2009.

[91] J. Shin, T. A. Badgwell, K. Liu, and J. H. Lee, "Reinforcement learning, overview of recent progress and implications for process control," *CCE*, vol. 127, pp. 282–294, 2019.

[92] R. S. Sutton, A. G. Barto, and R. J. Williams, "Reinforcement learning is direct adaptive optimal control," *ACC*, 1991.

[93] T. Moriyama, G. D. Magistris, M. Tatsubori, T. Pham, A. Munawar, and R. Tachibana, "Reinforcement learning testbed for power-consumption optimization," *AsiaSim*, 2018.

[94] L. Raji, R. S. Milton, S. Suresh, and S. Sankar, "Reinforcement learning in adaptive control of power system generation," *Procedia Computer Science*, pp. 202–209, 2015.

[95] H. Fan, C. Yao, J. Guo, and X. Lu, "Deep reinforcement learning for energy efficiency optimization in wireless networks," *ICCCBDA*, 2019.

[96] T. Technologies, *Pumps & process automation lab*, `www.turbinetechnologies.com/educational-lab-products/pump-lab-with-automation`, Accessed: 2019-07-02.

[97] E. Meijering, "A chronology of interpolation: From ancient astronomy to modern signal and image processing," *IEEE*, vol. 90, pp. 319–342, 2002.

[98] R. M. Felder and R. W. Rousseau, *Elementary Principles of Chemical Processes. 3rd Edition*. Wiley, 2004.

[99] A. Zhang, X. Yin, S. Liu, J. Zeng, and J. Liu, "Distributed economic model predictive control of wastewater treatment plants," *Chemical Engineering Research and Design*, vol. 141, pp. 144–155, 2019.

[100] S. Richter and C. N. J. amd M. Morari, "Computational complexity certification for real-time mpc with input constraints based on the fast gradient method," *IEEE Transactions on Automatic Control*, vol. 57, pp. 1391–1403, 2012.

[101] J. C. Dunn and D. P. Bertsekas, "Efficient dynamic programming implementations of newton's method for unconstrained optimal control problems," *Journal of Optimization Theory Applications*, vol. 63, pp. 319–342, 1989.

[102] S. J. Wright, "Applying new optimization algorithms to model predictive control," *Chemical Process Control*, vol. 90, pp. 147–155, 1997.

[103] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," *Proc. 17th World Congress*, 2008.

[104] G. Pannocchia, M. Gabiccini, and A. Artoni, "Offset-free mpc explained: Novelties, subtleties, and applications," *IFAC*, vol. 48, pp. 342–351, 2015.

[105] AspenTech, *Aspen dmcplus*, `https://www.aspentech.com/en/products/pages/aspen-dmcplus`, Accessed: 2019-08-08.

[106] AspenTech, *Aspen dmc3*, `https://www.aspentech.com/en/products/msc/aspen-dmc3`, Accessed: 2019-05-08.

[107] D. Gorges, "Relations between model predictive control and reinforcement learning," *IFAC*, vol. 50, pp. 4920–4928, 2017.

[108] P. Mhaskar, J. Liu, and C. D. Panagiotis, *Fault-Tolerant Process Control*. London, UK: Springer, 2013.

[109] S. A. Lawal and J. Zhang, "Actuator fault monitoring and fault tolerant control in distillation columns," *International Journal of Automation and Controlling*, vol. 6, 2016.

[110] L. D. Tufa and C. Z. Ka, "Effect of model plant mismatch on mpc performance and mismatch threshold determination," *Procedia Engineering*, vol. 148, pp. 1008–1014, 2016.

[111] F. Farivar and M. Ahmadabadi, "Continuous reinforcement learning to robust fault tolerant control for a class of unknown nonlinear systems," *Applied Soft Computing*, vol. 37, pp. 702–714, 2015.

[112] D. Zhang and Z. Gao, "Reinforcement learning based fault-tolerant control with application to flux cored wire system," *Measurement and Control*, vol. 51, pp. 349–359, 2018.

[113] A. Barto, R. S. Sutton, and P. S. Brouwer, "Associative search network: A reinforcement learning associative memory," *Biological Cybernetics*, vol. 40, pp. 201–211, 1981.

[114] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques part i: Fault diagnosis with model-based and signal-based approaches," *Industrial Electronics*, vol. 62, pp. 3757–3767, 2015.

[115] J. Jiang and X. Yu, "Fault-tolerant control systems: A comparative study between active and passive approaches.," *Annual Reviews in Control*, vol. 36, pp. 60–72, 2012.

[116] R. Sutton, A. Barto, and R. Williams, "Reinforcement learning is direct adaptive optimal control," *IEEE Control Systems Magazine*, vol. 12, pp. 19–22, 1992.

[117] F. Berkenkamp, M. Turchetta, A. Schoellig, and A. Krause, "Safe model-based reinforcement learning with stability guarantees," *Neural Information Processing Systems*, pp. 908–919, 2017.

[118] C. Watkins and P. Dayan, "Technical note, q-learning," *Machine Learning*, pp. 279–292, 1992.

[119] S. Koenig and R. G. Simmons, "Complexity analysis of real-time reinforcement learning," *Association for the Advancement of Artificial Intelligence*, pp. 99–105, 1992.

[120] J. Stichlmair and J. R. Fair, *Distillation: Principles and Practices*. New York: Wiley, 1998.

[121] S. Rajendran and S. Mathew, "Design and development of model predictive controller for binary distillation column," *International Journal of Science Education*, vol. 2, pp. 445–451, 2013.

[122] H. Z. Kister, *Distillation Operation*. Hoboken, New Jersey: McGraw-Hill, 1990.

[123] R. E. Sonntag, C. Borgnakke, and G. J. V. Wylen, *Fundamentals of Thermodynamics*. Hoboken, New Jersey: John Wiley et Sons, Inc., 1998.

[124] R. K. Wood and M. W. Berry, "Terminal composition control of a binary distillation tower," *Chemical Engineering Science*, vol. 28, pp. 1707–1717, 1973.

[125] C. Rajapandiyan and M. Chidambaram, "Controller design for mimo processes based on simple decoupled equivalent transfer functions and simplified decoupler," *Industrial and Engineering Chemistry Research*, vol. 51, pp. 12 398–12 410, 2012.

[126] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go

with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[127] DeepMind, *Alphago*, `https://deepmind.com/research/case-studies/alphago-the-story-so-far`, Accessed: 2019-08-02.

[128] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, A. Chen, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, and G. Driessche, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[129] DeepMind, *Alphazero: Shedding new light on chess, shogi, and go*, `https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go`, Accessed: 2019-05-02.

[130] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kuraran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv:1712.0181*, 2017.

[131] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kuraran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, 2018.

[132] C. Chabris, "The real kings of chess are computers," *Wall Street Journal*, 2015.

[133] DeepMind, *Alphastar: Mastering the real-time strategy game starcraft ii*, `https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii`, Accessed: 2019-08-01.

[134] OpenAI, *Openai five*, `https://openai.com/blog/openai-five/`, Accessed: 2018-11-12.

[135] OpenAI, *How to train your openai five*, `https://openai.com/blog/how-to-train-your-openai-five/`, Accessed: 2019-05-18.

[136] M. N. Howell and M. C. Best, "On-line pid tuning for engine idle-speed control using continuous action reinforcement learning automata," *Control Engineering Practice*, vol. 8, pp. 147–154, 2000.

[137] X. Wang, Y. Cheng, and W. Sun, "A proposal of adaptive pid controller based on reinforcement learnin," *China Univ Mining & Technol*, vol. 17, pp. 40–44, 2006.

[138] M. Sedighizadeh, A. Rezazadeh, and W. Sun, "Adaptive pid controller based on reinforcement learning for wind turbine control," *International Journal of Electrical and Information Engineering*, vol. 2, pp. 124–130, 2008.

[139] A. Akbarimajd, "Reinforcement learning adaptive pid controller for an under-actuated robot arm," *International Journal of Integrated Engineering*, vol. 7, pp. 20–27, 2015.

[140] A. Hakim, H. Hindersah, and E. Rijanto, "Application of reinforcement learning on self-tuning pid controller for soccer robot multi-agent system," *rICT & ICeV-T*, 2013.

[141] A. Doerr, D. Nguyen-Tuong, A. Marco, S. Schaal, and S. Trimpe, "Model-based policy search for automatic tuning of multivariate pid controllers," *arXiv:1703.02899*, 2017.

[142] J. C. Hoskins and D. M. Himmelblau, "Process control via neural networks and reinforcement learning," *Computers and Chemical Engineering*, vol. 16, pp. 241–251, 1992.

[143] S. P. K. Spielberg, R. B. Gopaluni, and P. D. Loewen, "Deep reinforcement learning approaches for process control," *AdCONIP*, vol. 10, pp. 201–207, 2017.

[144] Y. Wang, K. Velswamy, and B. Huang, "A long-short term memory recurrent neural network based reinforcement learning controller for office heating ventilation and air conditioning systems," *Processes*, vol. 5, pp. 1–18, 2017.

[145] Y. Wang, K. Velswamy, and B. Huang, "A novel approach to feedback control with deep reinforcement learning," *IFAC*, pp. 31–37, 2018.

[146] R. Nian, J. Liu, B. Huang, and T. Mutasa, "Fault tolerant control system: A reinforcement learning approach," *SICE*, pp. 1010–1015, 2019.

[147] J. Cannadey, "Next generation intrusion detection: Autonomous reinforcement learning of network attacks," *NISSC*, pp. 1–12, 2000.

[148] X. Xu, "Sequential anomaly detection based on temporal-difference learning: Principles, models and case studies," *Applied Soft Computing*, vol. 10, pp. 859–867, 2010.

[149] C. Huang, Y. Wu, Y. Zuo, K. Pei, and K. Min, "Towards experienced anomaly detector through reinforcement learning," *AAAI*, vol. 10, pp. 8087–8088, 2018.

[150] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," *ACM SIGKDD*, vol. 7, pp. 1939–1947, 2015.

[151] DeepMind, *Deepmind ai reduces google data centre cooling bill by 40%*, `https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40`, Accessed: 2019-05-01.

[152] DeepMind, *Safety-first ai for autonomous data centre cooling and industrial control*, `https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control`, Accessed: 2019-02-15.

[153] J. Conca, "Pick your poison for crude – pipeline, rail, truck or boat," *Forbes*, 2014.

[154] CIA. (2016). The world factbook - central intelligence agency, Central Intelligence Agency, [Online]. Available: `www.cia.gov` (visited on 09/06/2016).

[155] S. García, J. Luengo, and F. Herrera, *Data Preprocessing in Data Mining*. New York, USA: Springer, 2015.

[156] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," *Conference on Knowledge Discovery and Data Mining*, pp. 226–231, 1996.

[157] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," *International Conference on Machine Learning*, pp. 807–814, 2010.

[158] C. W. J. Granger, "Non-linear models: Where do we go next - time varying parameter models?" *Studies in Nonlinear Dynamics & Econometrics*, vol. 12, pp. 1558–3708, 3 2008.

[159] N. R. Canada. (2017). Crude oil facts, [Online]. Available: `https://www.nrcan.gc.ca/energy/facts/crude-oil/20064`.

[160] E. I. Administration. (2019). Field production of crude oil, [Online]. Available: `https://www.eia.gov/dnav/pet/hist/LeafHandler.ashx?n=PET&s=MCRFPUS2&f=M`.

[161] K. Gillingham and J. H. Stock, "The cost of reducing greenhouse gas emissions," *Journal of Economic Perspectives*, pp. 1–31, 2018.

Appendix A

Process Monitoring and Optimization of an Industrial Pipeline

Pipelines are critical for safe and efficient transportation of fluids across long distances. For example, pipelines are used by utility companies to transport clean water and natural gas to homes for heating and living purposes. Furthermore, pipelines are used in agriculture to transport irrigation water to hydrate crops. Moreover, pipelines are used by energy companies for transporting energy-rich hydrocarbons to fuel the world's transportation and manufacturing needs. In the United States, over 70% of petroleum products are shipped by pipeline. In Canada, this number increases to 97% [153]. Data in 2014 estimates that there are approximately 3,500,000 kms of operational pipelines across 120 countries [154]. Due to the world's dependency on pipelines for transporting their basic needs, ensuring its reliability and efficiency has a global-scale impact.

Typical pipelines have hundreds of digital measurements per minute and are hard to analyze; however, machine learning methods benefit greatly from large amounts of data. Thus, an opportunity was discovered where machine learning methods can be applied to pipelines used to transport petroleum products. The objectives of this

This project was sponsored by Mitacs through the Mitacs Accelerate Program.

project were to leverage machine learning to identify anomalous pipeline behaviour, and to build a real time optimization tool to automate, normalize and enhance pipeline operation.

In this chapter, a classification machine learning algorithm will first be introduced for process monitoring and to detect anomalous activity within any pipeline equipment. Then, a real time optimization tool will be shown. Due to confidentiality agreements, all information presented here-forth will be masked, and all parties of this project will remain anomalous.

A.1 Process Introduction

Two separate pipelines, Line A and Line B, were analyzed. Line A is a simplistic pipeline with few operating variables. Due to the lack of operational complexity, the data was used to construct an anomaly detection monitoring tool. Line B was more complex and had many degrees of freedom. Due to the additional complexity, the pipeline operators were unsure about the optimal operations of the pipeline. Thus, a real time optimization tool was built for this line.

A.1.1 Line A

The schematic of Line A is shown in Figure A.1. For Line A, the objective was to build an anomaly detection tool to predict unexpected shut downs of its pumps.

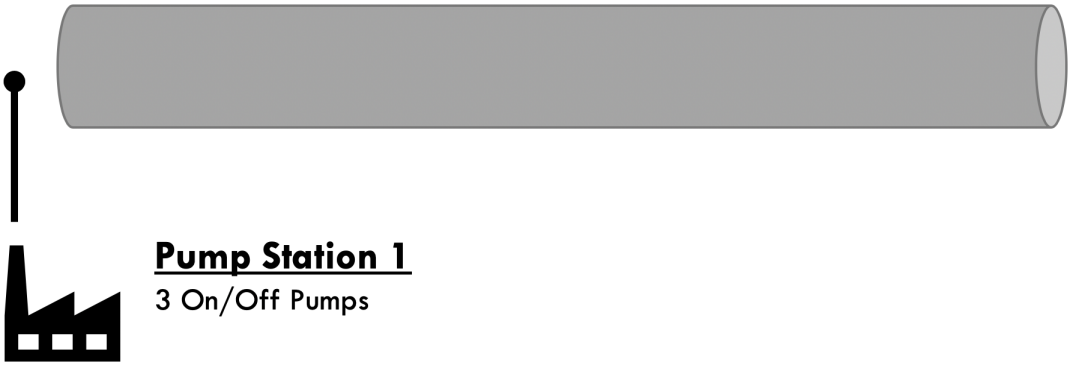


Figure A.1: Schematic diagram of Line A.

A.1.2 Line B

A schematic of Line B is shown in Figure A.2. Line B is a complex pipeline spanning over 100 kms and carries two products, a lighter product and a heavier product. The two products are batched (i.e., rotate between sending each product) and each product is sent for approximately eight hours before switching to the other product. The American Petroleum Institute (API) gravity for the lighter and heavier products are roughly 40 and 20, respectively. For the rest of this chapter, the lighter and heavier product will be referred to as *light crude* and *heavy crude*, respectively. The pipeline is typically operated between 1800 bbl/h to 3050 bbl/h.

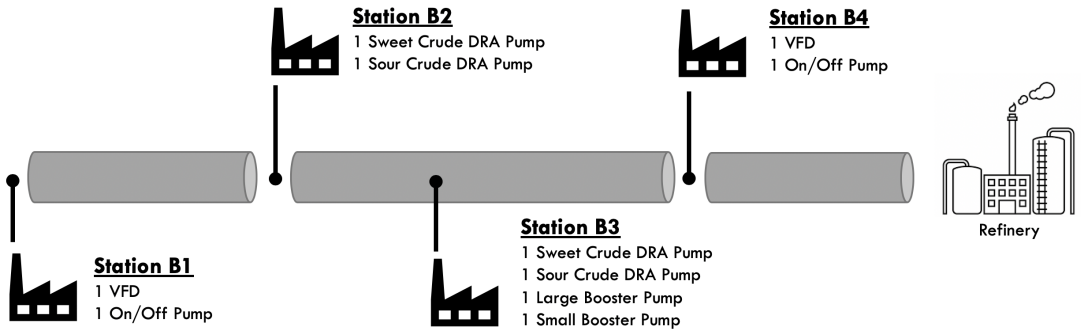


Figure A.2: Schematic diagram of Line B.

Equipment wise, Line B boasts eight pumps spread across four pump stations. Two pumps are variable frequency drives (VFD), while the rest are on/off pumps. Additionally, there are four drag reducing agent (DRA) injection pumps situated at the second and third pump stations. Each pump station contains a heavy crude and light crude DRA pump because the different crudes use different types of DRA. The DRA is injected based on the product present at the pump station.

A.2 Real Time Optimization

The hierarchical structure of a typical process control system is shown in Figure A.3. Modern control systems typically consists of three layers: real time optimization (RTO), supervisory control, and regulatory control. From the top, real time optimization is evaluated the least frequently, and performs a steady state optimiza-

tion of the process. The outputs of RTO are the ideal set points for all equipment given an operating objective. Next, the supervisory control layer performs dynamic optimization to identify the most efficient input trajectory to achieve the set points from RTO. Supervisory control is evaluated faster than RTO, but slower than regulatory control. Model predictive control (MPC) and economic model predictive control (EMPC) are typical supervisory control frameworks. More recently, reinforcement learning (RL) can also be retrofitted into this control layer. Finally, the regulatory controllers actuate the physical equipment and follows the input trajectory given by the supervisory control layer while maintaining stability. Common regulatory controllers are proportional-integral-derivative (PID) controllers.

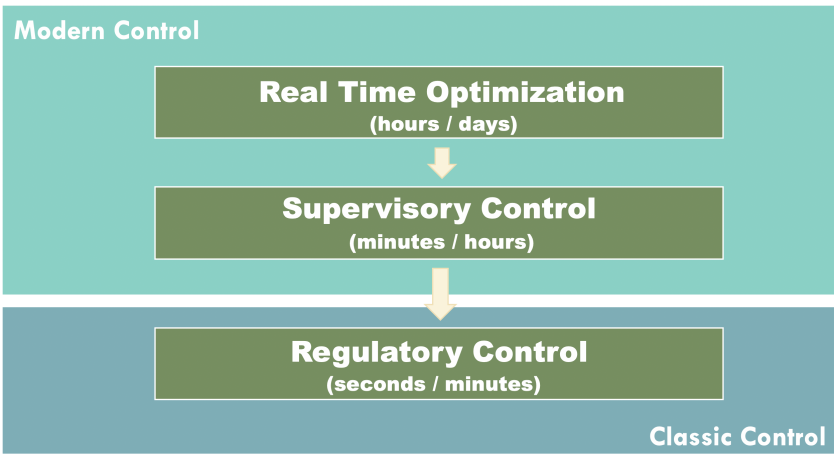


Figure A.3: Hierarchy of a typical control system.

A.2.1 Problem Description

Figure A.4 shows the traditional communication framework for operating a pipeline. The goal of RM06A is to meet the demands of the downstream refinery. To do so, a schedule with desired flow rates are sent to the operators from the scheduling team, and the operators are tasked to operate the pipeline at the given flow rate. Due to the complexity of this pipeline, different operators operate the pipeline differently depending on their own experience. This difference introduces turbulence and unnecessary wear-and-tear onto the pipeline, increasing maintenance costs. Moreover, some operators are less experienced and operate the pipeline sub-

optimally.

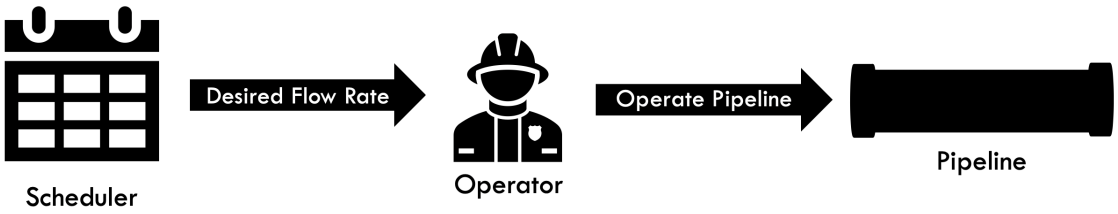


Figure A.4: Communication framework for operating Line B.

To overcome this problem, machine learning was used to identify a data driven model of the pipeline. Then, a steady state optimization tool was built using mixed integer linear programming (MILP) to give operators the **optimal** set-points for each equipment given a desired flow rate. This system achieves the following three objectives: i) Introduces uniformity in operator performance for desired set-points; ii) semi-automation of the pipeline, freeing up operators' time for other tasks; iii) training tool for new operators in a non-safety critical environment (comparable to flight simulators used for pilots).

The new ML-assisted communication framework for Line B is shown in Figure A.5. Here, the desired flow rate is sent to both the operator and optimization tool. The tool will then identify the most efficient set points to achieve the desired set point based on given cost metrics and recommend them to the operators. The tool is fully capable of automating the pipeline in open loop; however, initial performance skepticism and lack of closed loop feedback may introduce unforeseen safety concerns.

The rest of this section is organized as follows. First, the data pre-processing step will be shown. Then, the model identification phase will be introduced. Following that, the optimization algorithm and constraints for the real-time optimization are presented. Finally, the section is concluded with some conceptual software design regarding its implementation into a supervisory control and data acquisition (SCADA) system and the overall project impact will also be shown.

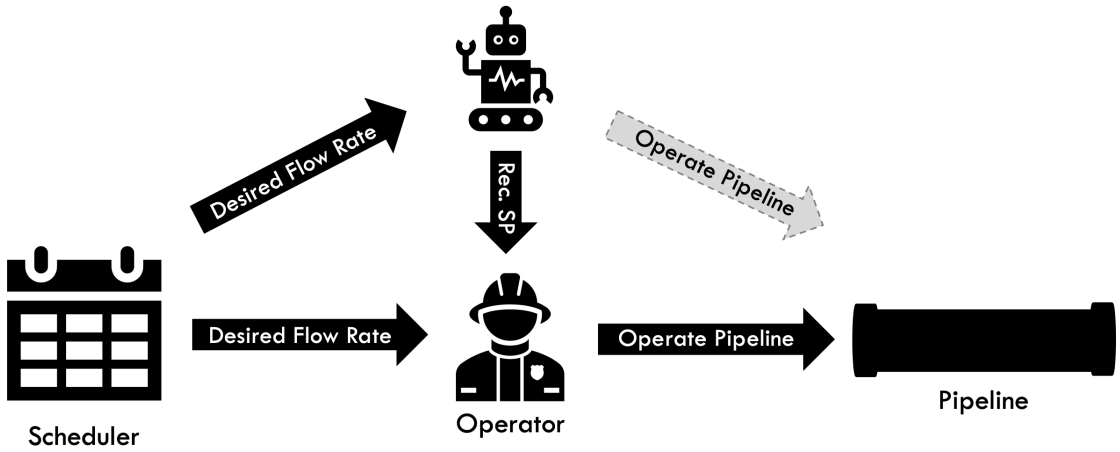


Figure A.5: Proposed communication framework for operating Line B.

A.2.2 Data Pre-processing

Two data sets were initially provided by the industrial sponsor. The details are shown in Table A.1. Model identification and optimization evaluations were conducted for both data sets; however, the steps are very similar. Because the second data set was used for the algorithm for live implementation whereas the first data set was used primarily as a proof of concept, only the steps for the second data set will be shown in detail.

Table A.1: Data details.

Date of Collection	Data Dimension
Date 1 - Date 2	$525,601 \times 899$
Date 3 - Date 4	$159,851 \times 738$

Data pre-processing can be broken down into three phases: Pre-processing by subject matter experts, automated data pre-processing, and manual data pre-processing. An iterative procedure followed phase three where the subject matter experts worked alongside the machine learning scientists to give suggestions on which variables should be included/excluded in the final model.

Filtering by Subject Matter Experts

The first phase of data pre-processing was conducted by subject matter experts from industry. The original data set contained all data corresponding to the pipeline.

Variables such as alarm limits, fire detector status, monitor on/off status, etc., have low predictive power and were removed. After this phase, the number of variables reduced from 738 to 124. The distribution of the remaining variables along the pipeline is shown in Table A.2.

Table A.2: Distribution of variables along Line B after phase 1 data pre-processing.

	B1	B2	B3	B4	Refinery	Other
# of Variables	24	21	21	33	22	3

Automated Data Pre-processing

Next, the data set was automatically filtered using the following methods:

- **Missing data removal**: Remove *rows* of data containing missing values.
- **Data imbalance analysis**: Remove boolean variables that contain 97% or more of a single class. Heavily imbalanced variables create model biases towards the majority class [155].
- **Collinear analysis**: Identify variables that are correlated over 90%. Correlation, r_{xy} is given in Equation A.1. After correlated variables are identified, one variable is kept while the rest are removed to avoid redundancy [155].

$$r_{xy} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}} \tag{A.1}$$

After performing the above methods, the data set reduced from 124 to 65. The distribution of the new data set is shown in Table A.3.

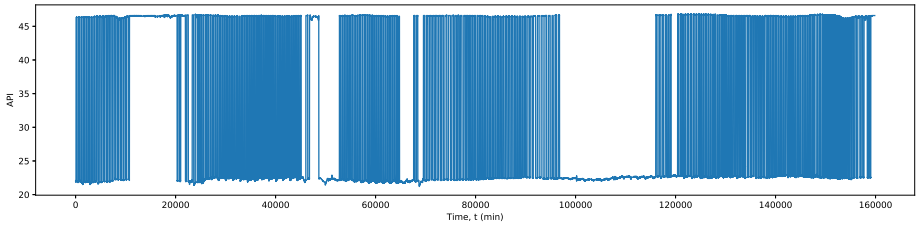
Table A.3: Distribution of variables along Line B after phase 2 data pre-processing.

	B1	B2	B3	B4	Refinery	Other
# of Variables	10	11	11	18	12	3

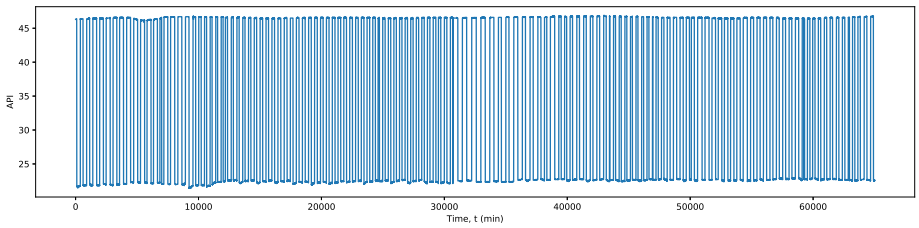
Manual Data Pre-processing

The data set is then manually pre-processed to remove or modify rows due to badly behaving sensors or irregular operating conditions. In this phase, only the number of training examples are reduced.

For this pipeline, both light and heavy crude are transported in a cyclical fashion due to the hydraulic dynamics of the pipeline. Otherwise, the heavy crude is too heavy to be transported for sustainable periods. However, downstream demand for each product can disrupt this operating cycle. From Figure A.6, it can be seen that there were extended periods of time where only light or heavy crude were being transported, and was caused by the lack of demand downstream. Such scenarios deviate from normal operations and were removed from the data.



(a) API data before abnormal condition removal.



(b) API data after abnormal condition removal.

Figure A.6: API data before and after removing abnormal operating conditions.

Moreover, there is a time delay for the flow rate to react to a DRA set point change because the new DRA concentration must be permeated throughout the line before its effect can be fully realized. DRA was assumed to be catastrophically destroyed when passing through a pump[1]; thus, DRA is only required to coat the pipeline between pump stations for its full effect to be exploited. For Station B2, it must coat the pipeline between Station B2 to Station B3. For Station B3, the

[1]According to the industrial sponsors.

pipeline spanning between Station B3 and Station B4 must be coated. Given the flow rate of the pipeline, it will take approximately ten hours to sufficiently coat the majority of the pipeline. Hence, data corresponding to transitional periods are removed. At times, transitional times may take longer; however, removing additional data will reduce the available data for model identification.

Pre- and post-processed DRA parts per million (ppm) measurements are shown in Figure A.7. DRA ppm is measured continuously for the control of the DRA injection pumps. However, the measurement is unreliable and corrupted with noise. Because DRA set points are rarely changed, an exponentially weighted moving average (EWMA) was applied to the DRA ppm readings for increased measurement reliability. The EWMA formula and bias correction are given in Equations 3.1 and 3.2, respectively.

The objective of the machine learning model was to predict the flow rate at refinery. However, there is a natural time delay between the time an equipment status changed and the corresponding impact on downstream flow rate. Because the pipeline is fully loaded and the product is incompressible, pressure changes upstream will be propagated downstream at close to the speed of sound [64]. Table A.4 shows the time required for pressure to propagate down the pipeline starting from each pump station. The pump data for each pump station was shifted accordingly to account for this time delay[2].

Table A.4: Time required for pressure changes at each pump station to be realized at refinery.

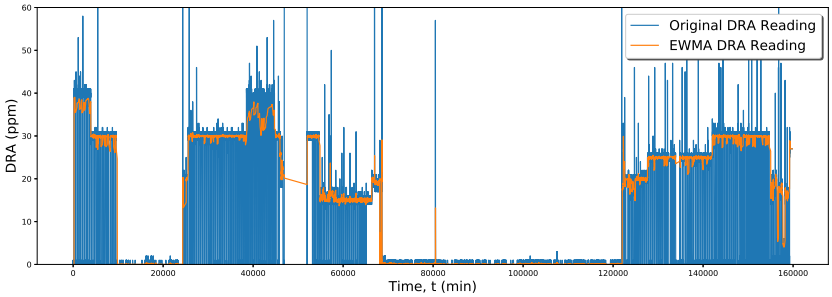
	B1	B2	B3	B4
Time to refinery at speed of sound in liquids (1480 m/s) [64]	2.0 min	2.0 min	1.0 min	1.0 min

A comprehensive list of the manual data pre-processing procedures is as follows:

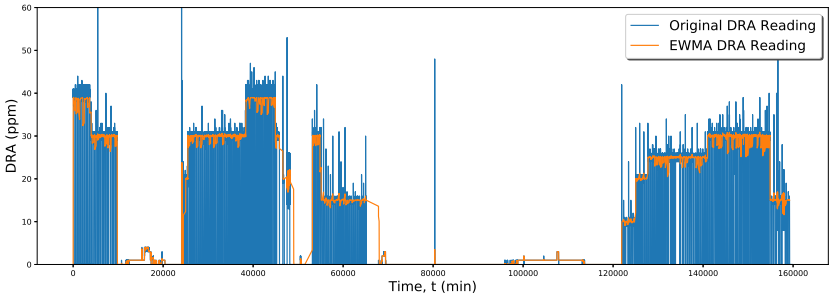
1. Shift data to accommodate the time delay at refinery.

2. Smooth DRA data using EWMA given in Equation 3.1.

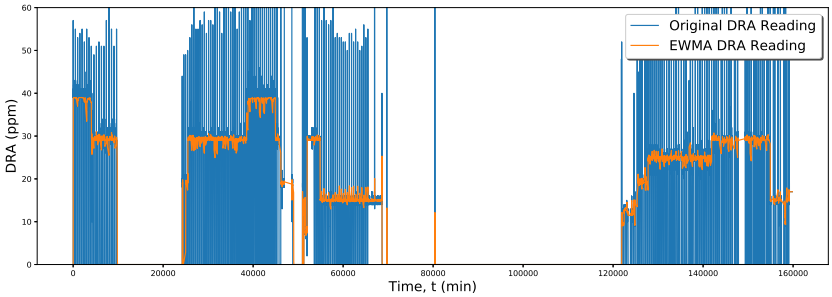
[2]Time delays are significantly rounded for confidentiality purposes.



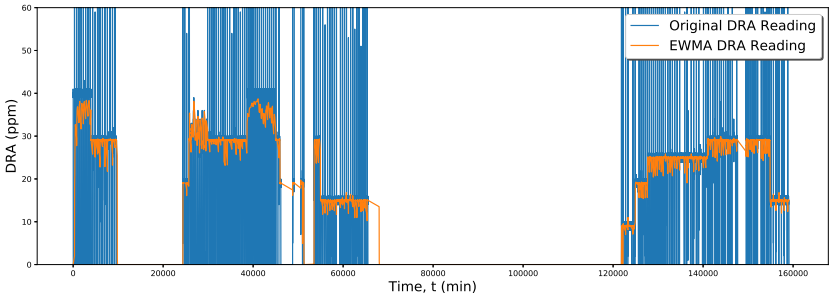
(a) Station B2 heavy DRA sensor reading.



(b) Station B2 light DRA sensor reading.



(c) Station B3 heavy DRA sensor reading.



(d) Station B2 light DRA sensor reading.

Figure A.7: Pre- and post-processed DRA sensor readings.

3. Remove first 10 hours data corresponding to DRA set point changes.

4. Remove data points where flow is under 800 bbl/hr.

5. Remove data when *only* light or heavy crude was sent through the pipeline.

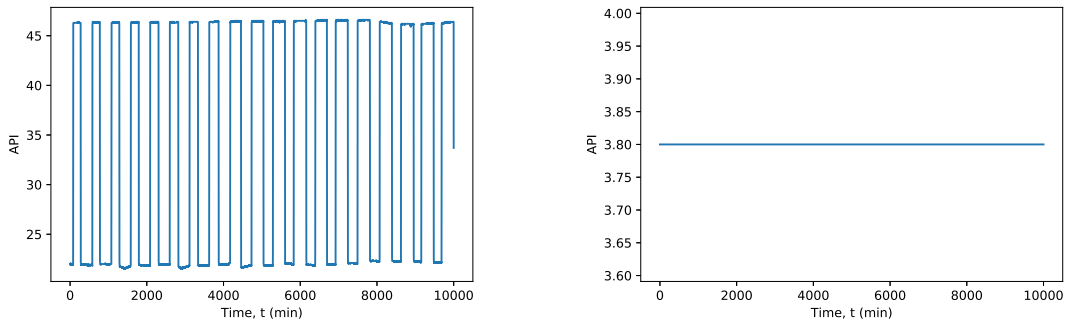
The final data set contained 65 variables and 97,470 data points.

A.2.3 Model Identification

Feature Selection

For each pump station, there was a variety of sensors measuring the same process variables. For example, VFD pumps have four readings each: On/off status, RPM, HZ, and current. Many variables relating to one equipment is redundant; thus, only one variable was selected when redundancy existed. Additionally, some sensors were behaving abnormally.

In normal operations, the density fluctuates between 10 - 50 API, depending on the crude present in the pipeline. After analysis, the Station B4 densitometer was behaving abnormally compared to other densitometer and is shown in Figure A.8. After confirming with the industrial that the densitometer was behaving abnormally, the variable was removed.



(a) Station B1 API data for 10,000 mins. (b) Station B4 API data for 10,000 mins.

Figure A.8: Comparison of normal and abnormal density readings.

Table A.5 shows the features selected for each pump station. The predicted variable was the flow rate (bbl/h) at refinery.

Table A.5: Features selected for machine learning models.

Station B1	Station B2	Station B3	Station B4	Refinery
u_5: Boos. Pump Status	u_1: Light DRA (ppm)	u_3: Light DRA (ppm)	u_8: Boos. Pump Status	x_8: Inlet Temp. (°C)
u_9: VFD Current (Amp)	u_2: Heavy DRA (ppm)	u_4: Heavy DRA (ppm)	u_{10}: VFD Current (Amp)	
x_4: Inlet Temp. (°C)	x_5: Inlet Temp. (°C)	u_6: Small Pump Status	x_7: Inlet Temp. (°C)	
x_1: API	x_2: API	u_7: Large Pump Status		
		x_6: Inlet Temp. (°C)		
		x_3: API		

Feature Scaling

Figure A.9 shows the contour of a normalized and non-normalized cost function. It can be seen that the optimization of a non-normalized cost function can be significantly hindered depending on where the optimization is initialized.

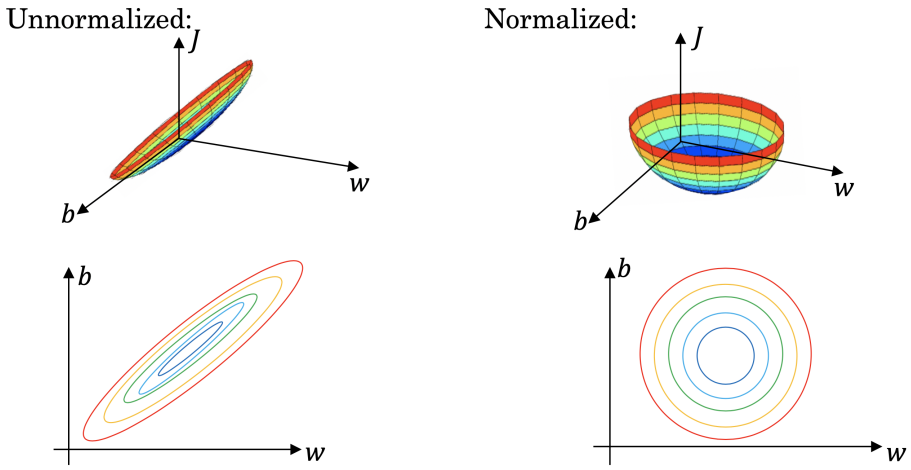


Figure A.9: Coutour of normalized and non-normalized cost functions. Original images from [39].

The min-max normalization was applied to each variable to avoid this problem and is given by:

$$x^{norm} = \frac{x - x^{min}}{x^{max} - x^{min}} \tag{A.2}$$

where x^{norm} is the normalized values. Here, x^{min} and x^{max} are the minimum and maximum values of each variable, respectively. By applying this normalization, all data will be bounded between $x_i \in [0, 1]$ and the elongation issue of the cost function was resolved.

Exploratory Data Analysis

Exploratory data analysis was then conducted to gain insights into the data set. First, the distribution of the flow rate was explored and shown in Figure A.10. It

can be seen that the flow rate follows a bi-modal distribution corresponding to two different operating strategies: a high demand strategy and a low demand strategy.

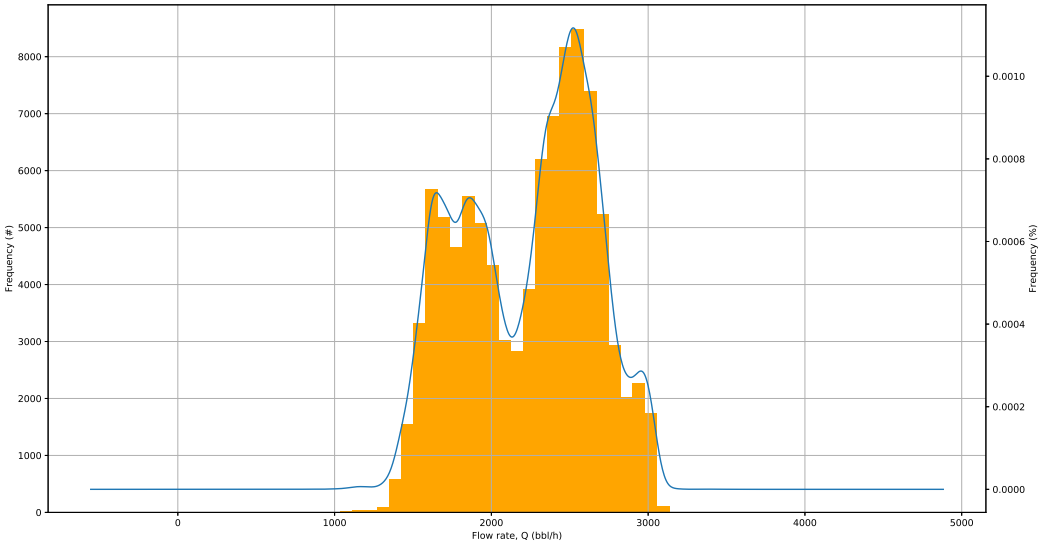


Figure A.10: Flow rate distribution of the pre-processed data set.

To segregate the two Gaussian distributions, DBSCAN was used. DBSCAN is a density-based algorithm for discovering clusters in large spatial data [156]. DBSCAN also scales much better to big data compared to affinity propagation or Gaussian mixture models due to the latter being iterative methods. DBSCAN contains two hyper parameters, ϵ and *min points*. The steps of DBSCAN is as follows:

1. Normalize the data using Equation A.2 so each variable is weighted similarily.

2. Create an $n-$dimensional sphere of radius ϵ around an initial data point. Euclidean distance was used for the distance metric and is given by:

$$distance = \sqrt{(x_1^{(1)} - x_1^{(2)})^2 + (x_2^{(1)} - x_2^{(2)})^2 + ... + (x_m^{(1)} - x_m^{(2)})^2} \tag{A.3}$$

   where superscripts 1 and 2 denotes the first and second data points.

3. If there are more than *min points* in this sphere, then all points within this sphere belong to the same cluster.

4. Expand the cluster by recursively applying the above criteria to the edge points of the cluster.

5. If the cluster can no longer be expanded, apply steps 2 - 4 to a new data point currently not belonging to a cluster.

6. If there are less than *min points* in this sphere, then the data point is ignored and we proceed to the next data point.

7. Outlier data points are ones that fail to belong to any cluster.

The resultant segregation created by DBSCAN using hyper parameters 1.13 and 10,000 for ϵ and *min points* is shown in Figure A.11. The first cluster (black) contained 56,738 data points while the second cluster (green) contained 36,779 data points. The remaining 3953 data points (blue) were identified as outliers.

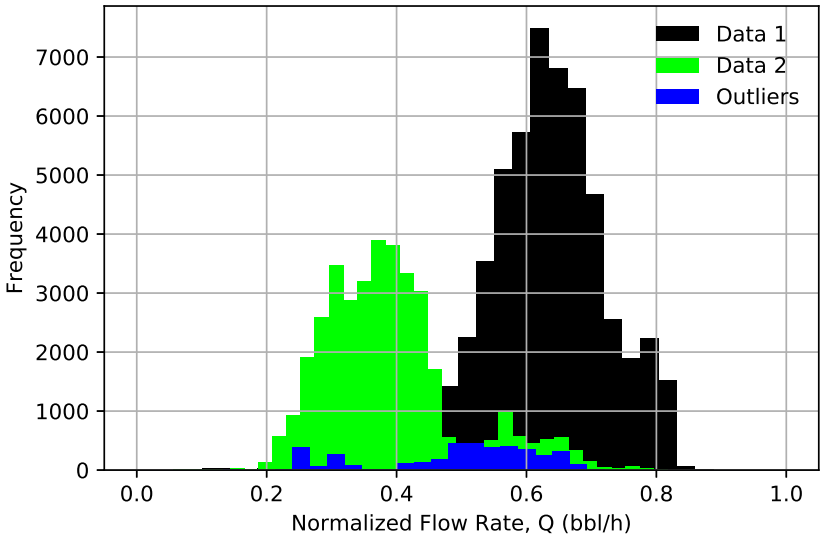


Figure A.11: Clusters identified after applying the density-based scan.

Average operating conditions of each cluster is shown in Figure A.12. The main differences are the flow rate, DRA usage, and pump usage. Cluster 1 had 32% higher average flow rate. Cluster 1 also used substantially more DRA compared to cluster 2, where almost no DRA was used[3]. Furthermore, the Station B1 booster pump,

[3]Actual values masked to ensure confidentiality

	Feature	Cluster 1 (Black)	Cluster 2 (Green)
	Flow Rate (bbl/h)	2500	1900
DRA	DRA – Station B2 Heavy (ppm)	High	Low
DRA	DRA – Station B2 Light (ppm)	High	Low
DRA	DRA – Station B3 Heavy (ppm)	High	Low
DRA	DRA – Station B3 Light (ppm)	High	Low
On/off Pump	Station B1 Booster Pump (% on)	100%	0%
On/off Pump	Station B3 Booster Pump 1 (% on)	0%	100%
On/off Pump	Station B3 Booster Pump 2 (% on)	95%	0%
On/off Pump	Station B4 Booster Pump (% on)	63%	0%
VFD	Station B1 VFD (amps)	225	184
VFD	Station B4 VFD (amps)	30	0

Figure A.12: Average operating variables for the two operating conditions.

Station B3 booster pump 1, Station B4 boooster pump, and Station B4 VFD were only used in cluster 1. Station B3 booster pump 2 was only used in cluster 2.

Data Partitioning

The data set was split into three sections for machine learning: training, validation, and testing. The partition and description of each section is shown in Table A.6. The training data set was used to identify the machine learning model(s). Then, the model was validated on unseen data via the validation data set (sometimes called development data). The error of the model on the validation data set, $e_{validation}$, was then evaluated and compared to the training data error, e_{train}. If the difference is large, the model was rebuilt using different data pre-processing techniques and features. This step was repeated until $e_{train} \approx e_{validation}$ to ensure that the model did not overfit to the training data. Finally, the model was tested on the testing data to explore the performance of the model in live production. Testing data was always the last 5% of the data set.

Table A.6: Description of each data partition.

	% of Data	Description
Training	90%	Identify the ML model
Validation	5%	Tune ML model performance on unseen data
Testing	5%	Test ML model performance on proxy live data

Cost Function

The mean squared error (MSE) cost function was used for all predictive models and is given by:

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{A.4}$$

where n is the number of samples in the current optimization step. $\hat{y}_i$ and y_i are the i^{th} predicted and actual labels, respectively. Here, J is the loss. The MSE cost function was selected due to its convex nature [39].

To ensure adequate performance on the validation and testing data, the model must avoid overfitting to the training data. This was done by reducing the model complexity through removing or reducing individual variables' impact on the model. This study used a ridge regularization to reduce model complexity:

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^{n} W_j^2 \tag{A.5}$$

where λ is the regularization penalty. Here, as $\lambda \to \infty$, $W \to 0$. That is, the larger λ is, the stronger large weights are penalized.

Model Optimization

The adaptive momentum (ADAM) gradient descent optimizer was used to update the weights and bias of the models. The general gradient descent formulation is given by Equation A.6 [40].

$$\theta_j^{m+1} \leftarrow \theta_j^m - \alpha\frac{\partial J}{\partial \theta_j} \tag{A.6}$$

where θ_j is the j^{th} weight of the model. Here, m represents the m^{th} update of gradient descent and α is the learning rate. ADAM improves upon Equation A.6 by computing an adaptive learning rate for each parameter. To do so, the exponentially decaying average of the past gradients and squared gradients of the weights and

biases are computed and stored using Equations A.7 to A.10.

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW \tag{A.7}$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \tag{A.8}$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2 \tag{A.9}$$

$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \tag{A.10}$$

where V and S are the estimates of the gradient and squared gradients, respectively. V and S are typically initiated as zero vectors and are heavily biased towards zero at initial steps. Hence, the biases for the initial terms are corrected using:

$$V_{dW}^{corrected} = \frac{V_{dW}}{1 - \beta_1^t} \tag{A.11}$$

$$V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t} \tag{A.12}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1 - \beta_2^t} \tag{A.13}$$

$$S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t} \tag{A.14}$$

Combining the above equations, the weights and biases are updated by:

$$W_j \leftarrow W_j - \alpha \frac{V_{dW}^{corrected}}{S_{dW}^{corrected} + \epsilon} \tag{A.15}$$

$$b \leftarrow b - \alpha \frac{V_{db}^{corrected}}{S_{db}^{corrected} + \epsilon} \tag{A.16}$$

where ϵ is a small scalar to avoid division by zero. The authors proposed values of 0.9, 0.999 and 10^{-8} for β_1, β_2, and ϵ, respectively.

Due to the size of the data, batch gradient descent where all data are used to compute the gradient at each step is computationally infeasible. Thus, mini-batch gradient descent was used where smaller batches of data were sampled from the

original data set to perform stochastic updates at each step.

Performance Assessment

The model performance were assessed using the following three ways:

1. Root mean squared error (RMSE):

$$J = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{A.17}$$

2. Mean absolute error (MAE):

$$J = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{A.18}$$

3. Coefficient of determination (R^2):

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y_i})^2} \tag{A.19}$$

Table A.7 shows the advantages and disadvantages of each assessment metric.

Table A.7: Pros and cons of different model performance assessment methods.

Method	Advantages	Disadvantages
RMSE	Useful for identifying large errors	Smaller errors are muted
MAE	Easy to interpret as all errors have the same weight	Inferior to RMSE when large errors are undesirable
R^2	Easy to understand, $-\infty \leq R^2 \leq 1$	Valid only for linear relationships

Linear Modelling

Linear Regression

Linear regression was the first regression method to be explored, and was selected as the benchmark due to its simplicity and linear nature. The model structure of

linear regression is given as:

$$\hat{y} = W_1^T x + W_2^T u + b \tag{A.20}$$

where $x \in R^8$ is a vector of states, $u \in R^{10}$ is a vector of inputs and superscript T denotes the transpose operation.

Hyper parameters of the linear regression are shown in Table A.8. The performance assessment of the least squares model is shown in Table A.9. The overall performance of the linear model was good. Error on the test data was 5.8% higher compared to the training and validation data.

Table A.8: Hyper parameters for linear regression.

Hyper Parameter	Value
Epochs	800
Minibatch size	8192
Learning rate, α	0.001
Regularization, λ	0.001

Table A.9: Performance assessment for the linear regression.

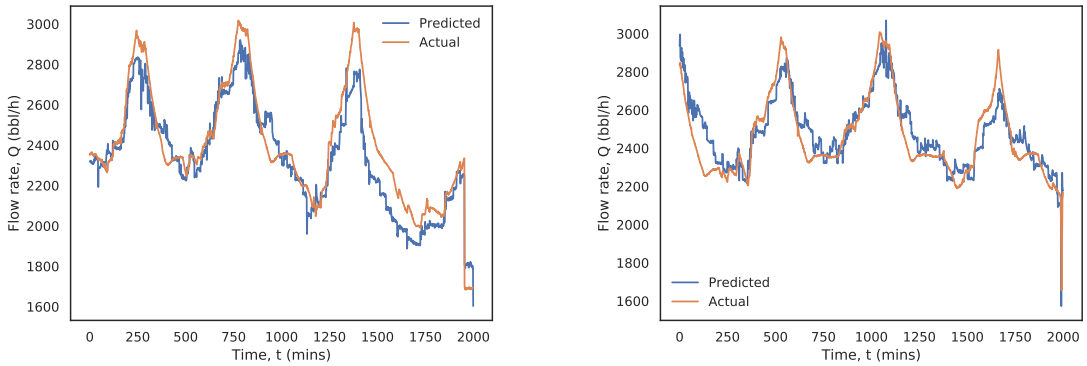
	Training data	Validation data	Test data
MAE	98	98	102
RMSE	127	127	135
R^2	0.91	0.91	0.70

The models' performance on the validation and test data sets are shown in Figures A.13a and A.13b.

The linear regression model is given in Equation A.20. Weights for $x_1 - x_4$ were very small and were omitted.

$$\begin{aligned}\hat{y} = {} & 0.10u_1 + 0.15u_2 + 0.13u_3 + 0.04u_4 + 0.04u_5 + 0.09u_6 + 0.12u_7 - 0.01u_8 \\ & + 0.49u_9 + 0.02u_{10} + 0.09x_1 - 0.18x_5 + 0.30x_6 - 0.05x_7 + 0.04x_8\end{aligned} \tag{A.21}$$

From Equation A.20, it can be seen that turning on the booster pump at Station



(a) Predicted vs. actual flow rate for the validation data set.

(b) Predicted vs. actual flow rate for the test data set.

Figure A.13: Linear regression validation and test plots.

B4 results in a decrease in flow rate. Theoretically, this is impossible and is most likely caused by noise in the data. To increase the model's ability to reflect reality, engineering knowledge was injected into the model via constraining the weights of $u_1 - u_{10}$ to be strictly positive.

Constrained Linear Regression

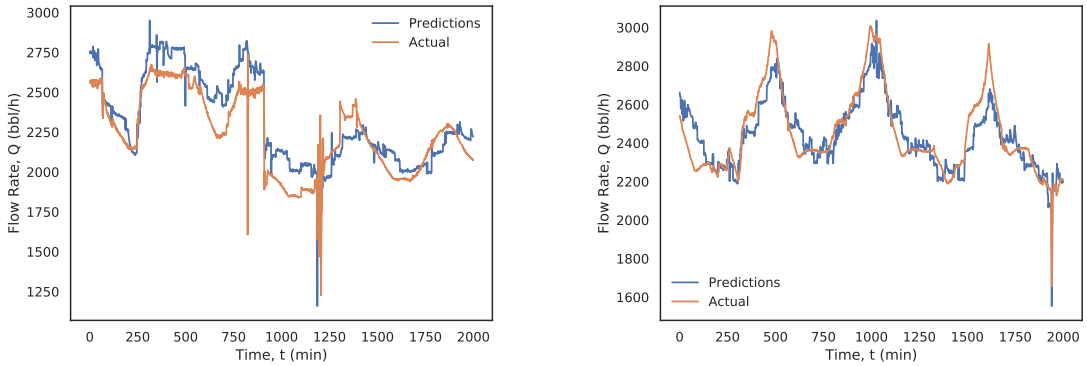
The constrained linear regression used the same hyper parameters as shown in Table A.8. Performance assessment of the constrained linear regression is shown in Table A.10. Compared to the original linear regression, RMSE increased by 0.8% for the training and validation data sets. However, performance on the test set was improved by 8%. The constrained linear regression performance on the validation

Table A.10: Performance assessment for the constrained linear regression.

	Training data	Validation data	Test data
MAE	98	98	94
RMSE	128	129	123
R^2	0.91	0.91	0.74

and test data sets are shown in Figures A.14a and A.14b. The weights were nearly identical to the unconstrained model; however, all negative weights on operating variables were removed.

The constrained linear regression is given in Equation A.22. Weights for $x_8, x_{11} -$



(a) Predicted vs. actual flow rate for the validation data set.

(b) Predicted vs. actual flow rate for the test data set.

Figure A.14: Constrained linear regression validation and test plots.

x_{14} were very small and were omitted.

$$\hat{y} = 0.10x_1 + 0.15x_2 + 0.13x_3 + 0.04x_4 + 0.04x_5 + 0.09x_6 + 0.11x_7 + 0.49x_9 + 0.02x_{10} - 0.18x_{15} + 0.30x_{16} - 0.05x_{17} + 0.03x_{18} \quad \text{(A.22)}$$

Non-linear Modelling

To further increase the accuracy of the models, the following non-linear methods were explored for modelling the pipeline flow rate:

- Polynomial models
  - Quadratic model
  - Square-root model
- Feed-forward neural networks
  - Small neural network (3 layers, 20 nodes per layer)
  - Medium neural network (6 layers, 30 nodes per layer)
  - Large neural network (8 layers, 40 nodes per layer)
- Linear parameter-varying model

Performance assessment of the non-linear models will use MAE and RMSE. In non-linear models, R^2 is not valid due to $SS_R + SS_E \neq SS_{Total}$ [44].

Polynomial Models

The quadratic and square-root model structures are given by Equations A.23 and A.24, respectively:

$$\hat{y} = W_1^T X^2 + W_2^T X + b \tag{A.23}$$

$$\hat{y} = W_1^T X^{1/2} + W_2^T X + b \tag{A.24}$$

where $W_1 \in R^{18}$ are the weights for the squared and square rooted variables for the quadratic and square root models, respectively. Furthermore, $W_2 \in R^{18}$ are the weights for the original variables. The hyper parameters for both models are shown in Table A.11.

Table A.11: Hyper parameters for polynominal regression.

Hyper Parameter	Value
Epochs	1000
Minibatch size	8192
Learning rate, α	0.001
Regularization, λ	0.001

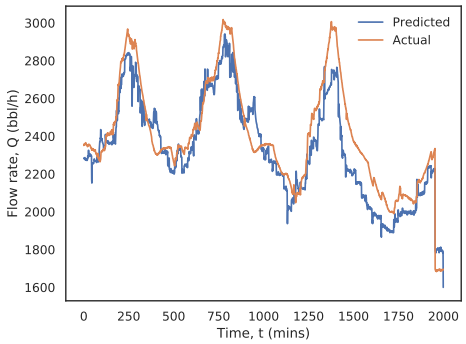
The performance assessment of the quadratic and square root models are shown in Table A.12. Compared to linear regression, MAE and RMSE reduced by up to 10% and 9% when using the polynomial models, respectively. Moreover, performance of the square root model was about 3.5% better than the quadratic model.

Table A.12: Performance assessment for the quad. and sqrt. model.

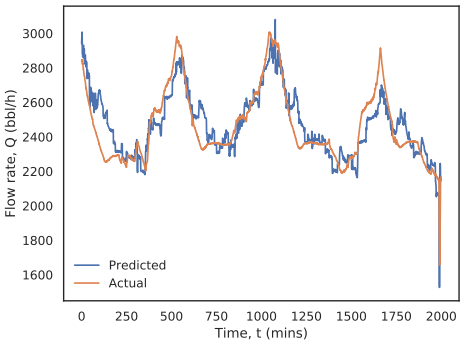
	Training data		Validation data		Test data	
	Quad	Sqrt	Quad	Sqrt	Quad	Sqrt
MAE	92	89	92	89	89	91
RMSE	121	118	121	117	120	115

The polynomial models' performance on the validation and test data are shown in Figures A.15a to A.15d. Overall, the model performances increased by introduc-

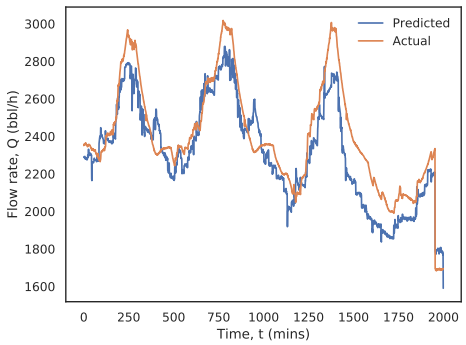
ing non-linearities. To further increase accuracy, heavily non-linear neural network models were explored.



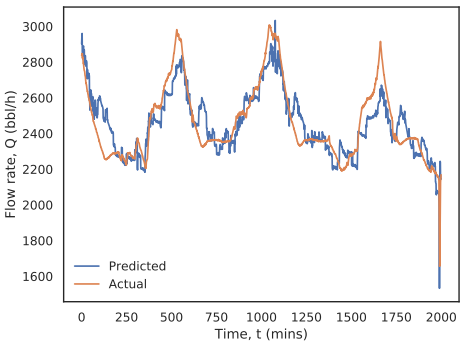
(a) Predicted vs. actual flow rate for validation data using the quad. model.



(b) Predicted vs. actual flow rate for the test data using the quadratic model.



(c) Predicted vs. actual flow rate for the validation data using the sqrt. model.



(d) Predicted vs. actual flow rate for the test data using the sqrt. model.

Figure A.15: Polynomial regression validation and test plots.

Neural Network Models

Neural networks are highly non-linear models that explore the individual and interaction effects of each variable with all other variables. The general structure of a neural network is shown in Figure A.16. Neural networks are comprised of an input layer, some hidden layer(s), and an output layer. The input layer consists of the input data, while the hidden layer(s) and output layer consists of fitted parameters, $W_{n_x \times n_b}$ and $b_{n_b \times 1}$. Here, n_b and n_x denotes the batch size and the dimension of the input layer, respectively. In Figure A.16, x_m denotes the m^{th} input variable. The superscript and subscript of a denotes the hidden layer number and the node number in the corresponding layer, respectively. Subscript m_1 to m_r denotes the number of nodes in hidden layers 1 to r, respectively. Finally, superscript o denotes

the output layer.

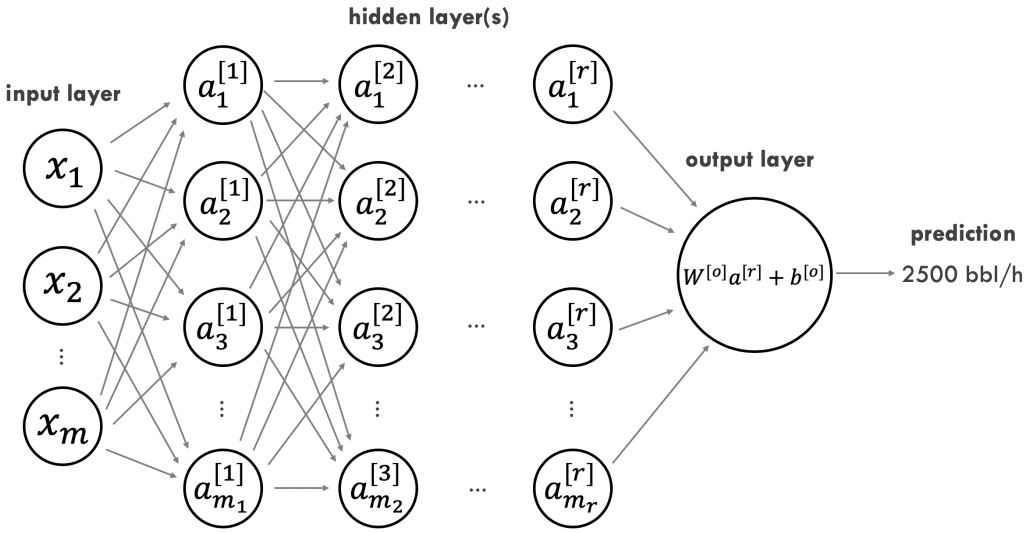


Figure A.16: Structure of a general neural network.

The details within a hidden layer's node is shown in Figure A.17. First, the outputs from the previous layer's nodes are inputted and multiplied by the weights of the current node. The current node's bias is then added. If the current node is in the first layer, the outputs from the previous layer is replaced with the input variables. Afterwards, the output is sent to a rectified linear unit (ReLU) activation function given by:

$$a_j^{[i]} = \begin{cases} y, & \text{if } y \geq 0. \\ 0, & \text{otherwise.} \end{cases} \tag{A.25}$$

where i and j denotes any hidden layer and any node number, respectively.

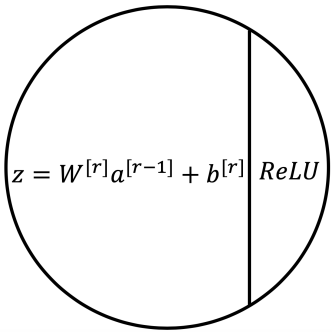


Figure A.17: Inside a hidden layer's node.

Mathematically, for one example x:

$$z_j^{[1]} = W^{[1]}x + b^{[1]}$$
$$a_j^{[1]} = ReLU(z_j^{[1]})$$
$$z_j^{[2]} = W^{[2]}a_j^{[1]} + b^{[2]}$$
$$a_j^{[2]} = ReLU(z_j^{[2]})$$
$$\dots$$
$$z_j^{[r]} = W^{[r]}a_j^{[r-1]} + b^{[r]}$$
$$a_j^{[r]} = ReLU(z_j^{[r]})$$
$$y = W^{[o]}a_j^{[r]} + b^{[o]}$$

The hyper parameters for each neural network is given in Table A.13. For highly complex models such as large neural networks, overfitting was inevitable; thus, λ was increased as the neural network got more complex to reduce overfitting. The ReLU activation function was chosen for computational efficiency and avoiding the exploding/vanishing gradient problem [157].

Table A.13: Hyper parameters for the feed-forward neural network.

Hyper Parameter	Small NN	Med. NN	Large NN
Epochs	700	1000	1200
Minibatch size	8192	8192	8192
Learning rate, α	0.001	0.001	0.001
Regularization, λ	0.001	0.003	0.005
Number of layers	3	6	8
Neurons per layer	20	30	40
Activation function for hidden layers	ReLU	ReLU	ReLU
Activation function for hidden layers	Linear	Linear	Linear

Table A.14 shows the performance assessment of the small, medium and large neural networks. In all cases, the training and validation data set performance was significantly better compared to the linear model; however, the performance was only slightly better on the test data set. On the training and validation data, the error went down by up to 61%. On the test data, error went down by up to 19%. The difference may be caused by the test data being different from the training data. Due to the complexity of neural network models, they perform exceptionally well on data that shares similar characteristics as the training data, but perform

poorly otherwise. To close the gap in performance, a higher λ could be used to reduce model complexity. Moreover, smaller neural networks could also be explored to reduce model complexity. Additionally, a significant amount of data is lacking for the application of neural networks to this data set because only winter months data was collected; however, the models were tested on summer months data where temperatures have increased by up to 10° C, significantly reducing viscosity of the shipped crude.

Table A.14: Performance assessment of the neural network models.

	Training Data			Validation Data			Test Data		
	Sm.	Med.	Lar.	Sm.	Med.	Lar.	Sm.	Med.	Lar.
MAE	48	42	38	50	45	37	87	87	91
RMSE	66	58	57	69	61	56	107	117	118

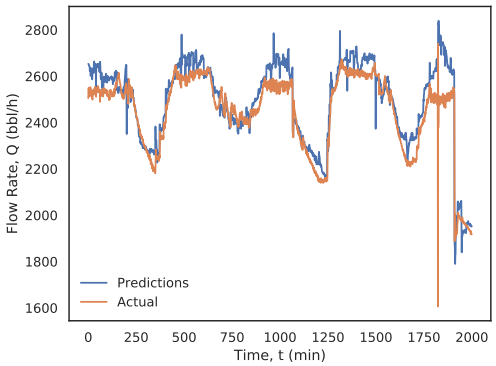
The comparison of actual and predicted flow rates on the validation and test data for the neural nets are shown in Figures A.18a to A.18f.

Linear Parameter-varying Models

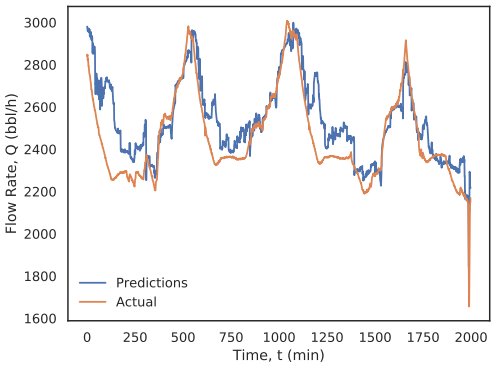
Lastly, the linear parameter-varying model (LPV) was explored to model the pipeline. It is clear that the process is non-linear due to the large increase in accuracy when switching to a non-linear model structure. LPV models were selected due to their non-linear nature while still retaining the interpretability of linear models. Furthermore, any non-linear model can be approximated by a set of linear models [158]. The LPV model is given by:

$$\begin{aligned} \hat{y} &= W_{1,1}^T x + W_{1,2}^T u + b_1 \\ \hat{y} &= W_{2,1}^T x + W_{2,2}^T u + b_2 \\ &\dots \\ \hat{y} &= W_{n,1}^T x + W_{n,2}^T u + b_n \end{aligned} \tag{A.26}$$

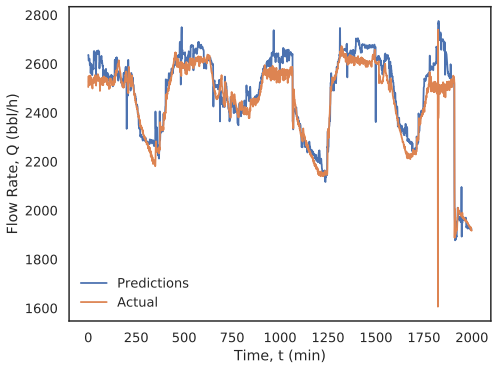
where $n \geq 1$ represents the number of linear models used to capture the data set. Here, W_n and b_n are the weights and biases corresponding to the n^{th} model, respectively. For this study, $n = 2$. The models corresponded to the two clusters



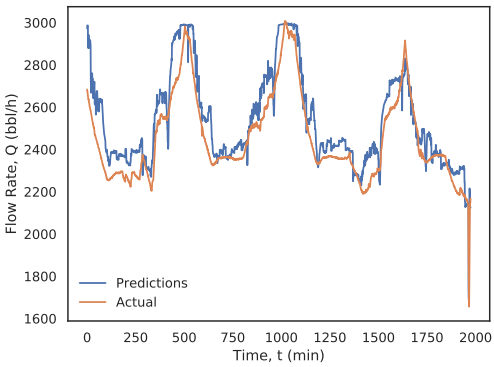
(a) Validation data for the small neural net.



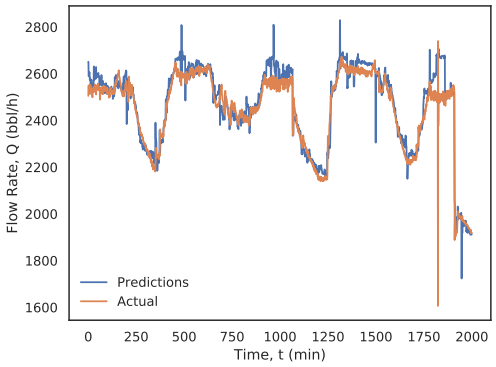
(b) Test data for the small neural net.



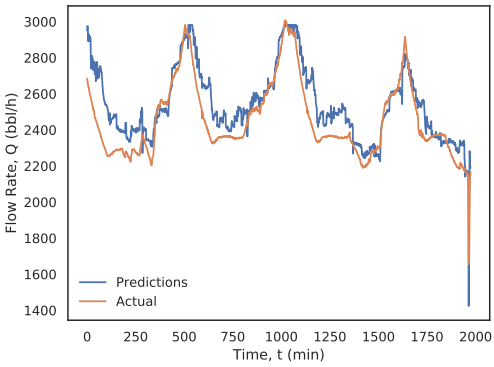
(c) Validation data for the med. neural net.



(d) Test data for the med. neural net.



(e) Validation data for the large neural net.



(f) Test data for the large neural net.

Figure A.18: Predicted vs. actual flow rates for the feed-forward neural networks.

identified in Figure A.11. Models 1 and 2 are identified from clusters 1 and 2, respectively. The hyper parameters for models 1 and 2 are identical to the previous linear models, and are shown in Table A.8. During online implementation, the model will be selected based on the Euclidean distance between the features of the new data and the centroid of the two clusters. However, if the distance exceeds 1.15 in both cases, the data will be labeled as anomalous.

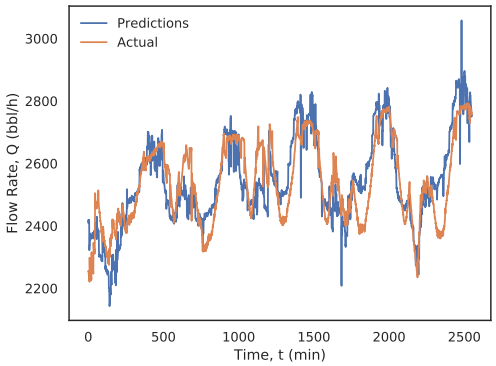
The performance assessment of the two LPV models are shown in Table A.15. Overall, the LPV models were able to reduce modelling error compared to normal linear regression. Model 1 was only used to predict high flow rate scenarios. Nevertheless, its MAE and RMSE were still 8% lower compared to the linear regression model used to predict for all data. For model 2, the MAE and RMSE were up to 31% lower.

Table A.15: Performance assessment for clusters 1 and 2 regression models.

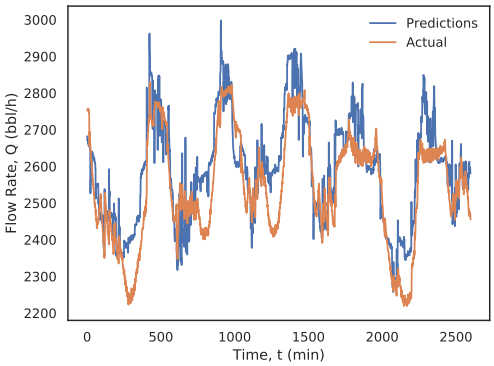
	Training data		Validation data		Test data	
	Cl. 1	Cl. 2	Cl. 1	Cl. 2	Cl. 1	Cl. 2
MAE	90	66	90	67	96	85
RMSE	115	91	116	92	120	110
R^2	0.87	0.90	0.86	0.89	0.78	0.57

The comparison of actual and predicted flow rates on the validation and test data for the LPV models are shown in Figures A.19a to A.19d. In Figures A.19c and A.19d, it can be seen that model 2 performs poorly on the testing data. This might be caused by some unobserved variables that were only used during low flow rate operations.

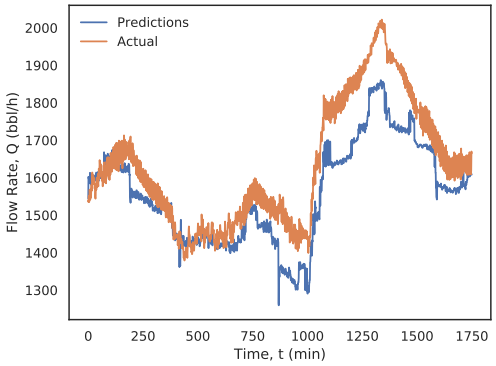
Ultimately, the LPV model structure was selected to model the pipeline. The LPV model has the interpretability of linear models, while having the predictive capabilities of non-linear models. Furthermore, the LPV model identifies a separate set of parameters for the two distributions, making the optimization algorithm more representative of live operations. For example, because no DRA was used for lower flow rates, the optimization algorithm will have explicit constraints on model 2 to not use DRA as well. A similar example would be using only Station B3 booster



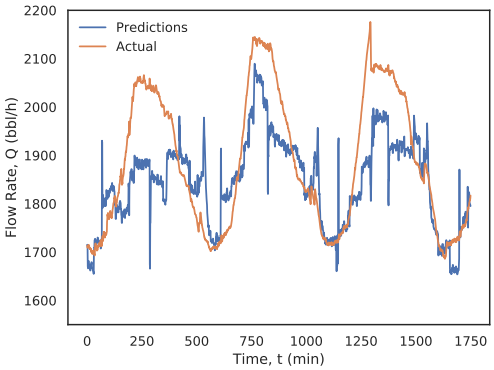
(a) Validation data using model 1.



(b) Test data using model 1.



(c) Validation data using model 2.



(d) Test data using model 2.

Figure A.19: Predicted vs. actual flow rate for the linear parameter-varying models.

1 for cluster 2, while using Station B3 booster 2 for cluster 1. The advantages of these constraints are twofold: i) More realistic to live operations. ii) Avoid model extrapolations (since no DRA data was used to identify model 2, the optimizer should not be able to use it for optimization).

Time-series Modelling

All previous models are static (i.e., $y_{ss} = f(x, u)$) and are used for the real time optimization layer in Figure A.3. To completely automate the pipeline, a dynamic model was identified for the application of supervisory control. In dynamic modelling, temporal correlations within the time-series data are exploited to improve model accuracy. Moreover, y_t is measured at each time and can be used to further

improve model accuracy. The time-series model will have the following structure:

$$y_{t+1} = f(x_t, x_{t-1}, ..., x_{t-p}, u_t, u_{t-1}, ..., u_{t-q}, y_r) \tag{A.27}$$

Here, subscripts p and q denote the number of previous states and inputs to be considered in the model, respectively. Subscript $r = max(p, q)$. Least squares will be used to identify the time-series model with data augmented as $[x_t, u_t|x_{t-1}, u_{t-1}|...]$. After exploring a variety of p's and q's, the final hyper parameters used for the time series model is shown in Table A.16.

Table A.16: Hyper parameters for the time-series least squares model.

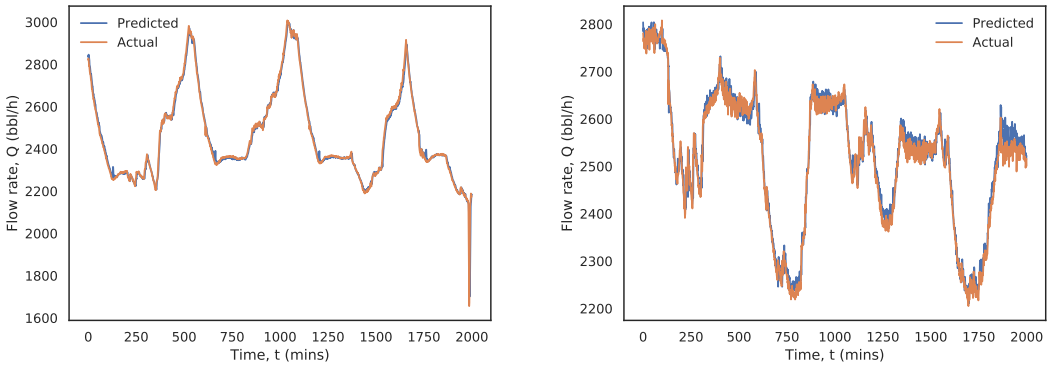
Hyper Parameter	Value
Epochs	1000
Minibatch size	8192
Learning rate, α	0.001
Regularization, λ	0.001
# of previous states, p	2
# of previous inputs, q	2

The performance metrics of the time-series model is shown in Table A.17. Compared to static models, dynamic models are significantly more accurate; error metrics went down by up to 87%.

Table A.17: Performance assessment for the time-series least squares model.

	Training data	Validation data	Test data
MAE	22	17	25
RMSE	41	34	65
R^2	0.99	0.99	0.86

The comparison of actual and predicted flow rates on the validation and test data for the time-series least square model are shown in Figure A.20. Here, it can be seen that the test data was much more noisy. Nevertheless, the model was still able to predict at a high level of accuracy.



(a) Validation data.

(b) Test data.

Figure A.20: Predicted vs. actual flow rate using the time-series model.

Model Applicability Range

Extrapolation while using data-driven models introduce risks to operations. Thus, Table A.18 show the range of states where the model is most effective. The ranges were identified from $\mu - 2\sigma \leq x, u \leq \mu + 2\sigma$ to ensure the range captures 95.5% of the data while omitting outliers. Usage of the models beyond the normal range may be ineffective.

Table A.18: Applicable states of the machine learning models.

Variables	Valid Range
Flow Rate (bbl/h)	2012 - 2913
Station B1 Density (API)	22 - 47
Station B2 Density (API)	15 - 45
Station B3 Density (API)	15 - 45
Station B1 Temp (°C)	35 - 50
Station B2 Temp (°C)	41 - 51
Station B3 Temp (°C)	43 - 54
Station B4 Temp (°C)	48 - 60
Refinery Temp (°C)	40 - 58

In addition to the state ranges provided above, the recommended set points are also limited to avoid extrapolation. The set point ranges are shown in Table A.19.

Table A.19: Applicable inputs of the machine learning models.

Variables	Valid Range
Station B1 VFD (Amps)	105 - 322
Station B4 VFD (Amps)	78 - 134
Station B2 & B3 Light DRA	Values
Station B2 & B3 Heavy DRA	Values

A.2.4 Mixed Integer Linear Programming

The information flow of the optimization algorithm is shown in Figure A.21. First, a desired flow will be inputted by the operators depending on the demand from refinery. Then, the machine learning model along with equality and in-equality constraints will be provided to the mixed integer linear program (MILP). Given the objective function, decision variables, and the current costs of operation, the MILP will output the optimal set points.

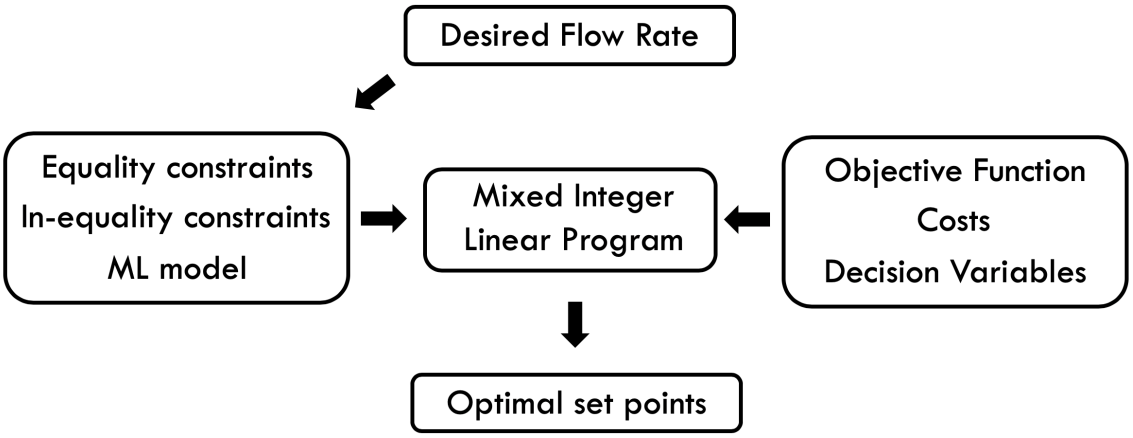


Figure A.21: Optimization information flow chart.

Mixed Integer Linear Program

The binary and continuous variables of the optimization problem are shown in Table A.20. MILP was used to perform steady state optimization on the linear model because the decision variables included binary and continuous variables. In total, there were 10 decision variables; 4 binary and 6 continuous.

Constraints

The equality and in-equality constraints are shown in Table A.21. From the data provided, the Station B1 pump was on 95% of the time, and at least one Station

Table A.20: Binary and continuous decision variables.

Binary	Continuous
On/off Station B1 Pump	Station B1 & Station B4 VFD
2× On/off Station B3 Pump	2× light & 2× heavy DRA
On/off Station B4 Pump	

B3 on/off pump was on; therefore, constraints were incorporated to reflect this. Additionally, constraints were used to ensure proper DRA was injected into the pipeline for the different crudes. There were also pressure constraints on the outlet pressure of the pumps to ensure safe operations. Before a recommendation is made to the operators, the recommended set points were used to predict for the outlet pressures at each pump to ensure they do not exceed the maximum allowable working pressure (MAWP). Finally, the final flow rate is given a range to ensure optimization feasibility. Specifying exact flow rates sometimes led to infeasible solutions because there may not exist a combination to get an *exact* flow rate.

Table A.21: List of equality and inequality constraints

Equality Constraints	In-equality Constraints
On/off Station B1 Pump = On	LB $\leq$ Flow rate $\leq$ UB
Station B3 Pump 1 and/or 2 = On	$P_{Outlet}^{Chey} \leq 1480$ psi
$API^{StationB2} \geq 30 : DRA_{light}^{StationB2}$ = On	$P_{Outlet}^{StationB3} \leq 1613$ psi
$API^{StationB2} \leq 30 : DRA_{heavy}^{StationB2}$ = On	$P_{Outlet}^{FL} \leq 1613$ psi
$API^{StationB3} \geq 30 : DRA_{light}^{StationB3}$ = On	LB ppm $\leq DRA \leq UB$ ppm
$API^{StationB3} \leq 30 : DRA_{heavy}^{StationB3}$ = On	105 Amps $\leq VFD^{Chey} \leq 322$ Amps
Station B1 On/off = 28 Amps	78 Amps $\leq VFD^{FL} \leq 134$ Amps
Station B3 1 Pump = 190 Amps	
Station B3 2 Pump = 278 Amps	
Station B4 Pump = 117 Amps	

Pressure models were built to ensure MAWP was not exceeded when the recommended set points are implemented. In the end, three linear regression pressure models were identified; each at the outlet of a pump station (Station B1, Station B3, Station B4) where pressure is expected to be the highest. The inputs to the

Table A.22: Inputs to the pressure constraint models.

B1 Outlet	B3 Outlet	B4 Outlet
Inlet pressure	Inlet pressure	Inlet pressure
B1 VFD	B1 VFD	B1 VFD
B1 on/off pump	B1 on/off pump	B1 on/off pump
B1 temperature	B3 on/off pump 1	B2 DRA ppm
	B3 on/off pump 2	B3 on/off pump 1
	B2 DRA ppm	B3 on/off pump 2
	B1 temperature	B3 DRA ppm
	B3 temperature	B4 VFD
		B4 on/off pump
		B1 temperature
		B3 temperature
		B4 temperature

pressure models were all upstream of where the pump station was located and is given in Table A.22.

The performance assessment of the Station B1, Station B3, and Station B4 pressure constraint models are shown in Table A.23. The Station B1 pressure prediction is very accurate; however the accuracy decreases as we traverse down the pipeline. The primary reason is because an accurate inlet pressure is required to predict for the outlet pressure. But the inlet pressures beyond Station B1 are unknown before the set points implemented in the pipeline (Ex: the inlet pressure at Station B3 is unknown because the recommended Station B1 VFD set points are not implemented yet; thus, the predicted output pressure will be inaccurate). Furthermore, predicting the outlet pressure at Station B3 is especially difficult because a mechanism exists to limit the pressure to 1377 psi. The mechanism's data is not captured, and causes the regression curve to be hard constrained at an upper bound. Consequently, this causes many inputs to result in the same output, and hinders learning.

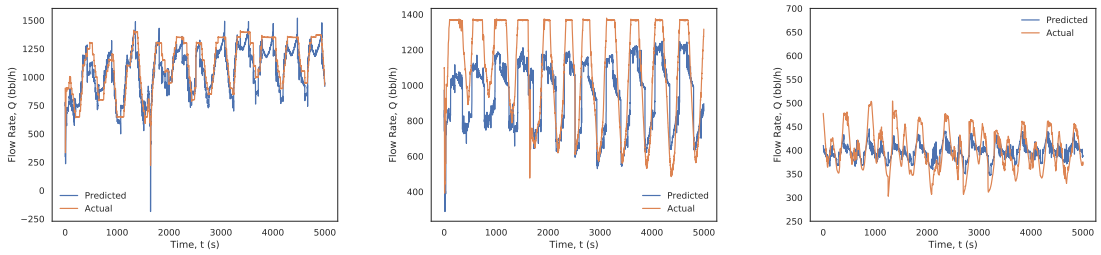
The performance of the pressure constraint models on the validation data is shown in Figures A.22a to A.22c.

Objective Function, Costs and Decision Variables

The objective is to operate the pipeline at the desired flow rate with the minimum

Table A.23: Performance assessment of the Station B1, Station B3, and Station B4 pressure constraint models.

	Training Data			Validation Data		
	B1	B3	B4	B1	B3	B4
MAE	75	179	25	72	179	27
RMSE	101	217	32	96	217	35
R^2	0.88	0.5	0.47	0.89	0.49	0.46



(a) Station B1 (b) Station B3 (c) Station B4

Figure A.22: Pressure constraint model performance on validation data.

possible operating cost. The optimization is given by:

$$
\begin{aligned}
\min_{J} \quad &= \left[\alpha_i \sum_i \mu_i \cdot DRA_i + \beta \sum_j \nu_j \cdot I_j V_j \right] \\
\text{s.t.} \quad & y = f(x, u), \\
& \nu_{StationB31} + \nu_{StationB32} \geq 1, \\
& \nu_{Chey} = 1, \\
& 20 \leq DRA_i \leq 40, \ i \in I, \\
& \gamma_j^{min} \leq I_j \leq \gamma_j^{max}, \ j \in J
\end{aligned}
$$

where α denotes the cost of DRA per ppm, β denotes the cost of one kWh, μ denotes the on/off status of the DRA pumps, and ν denotes the on/off status of the pumps. DRA_i is the DRA ppm set point of the i^{th} DRA injection pump. Moreover, V_j and I_j are the voltage and amperes corresponding to the j^{th} pump. Here, γ_j are the upper and lower bounds of the VFDs' currents and are provided in Table A.21. For on/off pumps, γ_j is fixed. The voltages of the pumps are provided in Table A.24.

The power cost per kWh for the pipeline was $0.09 USD according to the local commercial electricity costs[4].

Table A.24: Voltages for the pumps.

Pump	Voltage
VFD^{Chey}	Large
VFD^{FL}	Large
On/off $Pump^{Chey}$	Small
On/off $Pump^{StationB31}$	Large
On/off $Pump^{StationB32}$	Large
On/off $Pump^{FL}$	Small

The cost curves for DRA is shown in Figures A.23. The cost to increase DRA ppm is dependent on the flow rate of the pipeline. Higher flow rates require more DRA to be injected to achieve a desired parts per million reading; hence a higher cost. Given a flow rate, the cost to increase the DRA ppm is given by Equations A.28 and A.29 for the light and heavy DRA, respectively[5].

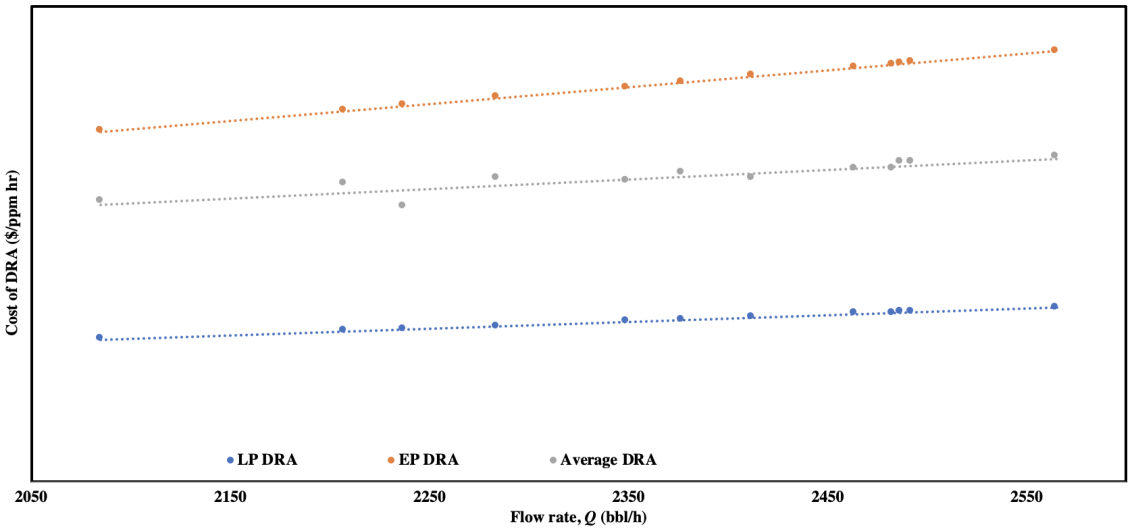


Figure A.23: Cost of DRA as a function of flow rate.

$$\$/ppm_{light} = W \cdot Q + b \tag{A.28}$$

$$\$/ppm_{heavy} = W \cdot Q + b \tag{A.29}$$

[4]Location masked for confidentiality.
[5]Costing information are highly confidential and are also masked.

where Q denotes the volumetric flow rate in barrels per hour.

A.2.5 Conceptual Software Design

The conceptual software design for the optimization tool is shown in Figure A.24. From the right, the users would first populate the costs corresponding to each equipment. The pump costs are on a \$/kWh basis, and the DRA cost is based on \$/ppm. Operating costs can also be incorporated by multiplying the costs by a fixed factor (e.g. $1.1 \cdot$ \$/kWh). Currently, DRA costs are calculated automatically based on the historical data using Equations A.28 and A.29. After specifying costs, the user would provide the maximum steady state operating pressure (MSSOP) for the pipeline, and enter the desired flow rate. Finally, the "OPTIMIZE" button would be pressed and all the boxes outlined in green will be populated with the optimal set point values. When a set point is no longer deemed optimal, the outline turns red. An example of such a scenario would be when the batch switches at the Station B2 or Station B3 stations. During maintenance activities, the user can click the "ADVANCED..." button to specify special constraints and eliminate the optimization from using certain equipment. For example, if the Station B1 VFD is under maintenance, $\nu_{CheyVFD}$ can be constrained to 0, preventing the optimization from using it.

A flow diagram of the tool's internal structure is shown in Figure A.25. Starting from the left, the operator would provide the constraints (equipment maintenance, MSSOP, etc.), desired flow rate, and operating costs to the optimization tool. Likewise, the optimization tool also receives live temperature and density readings from the SCADA system. Given the densities at Station B2 and Station B3, the proper VFD pumps are turned on. From the flow rate and DRA pump status, DRA costs are computed from Equations A.28 and A.29. Next, the costs, densities, and temperatures are all fed into the machine learning model and the MILP is ran to find the optimal set points to reach the desired flow rate. The set points are then sent to the pressure constraint models to validate that no outlet pressure constraints are

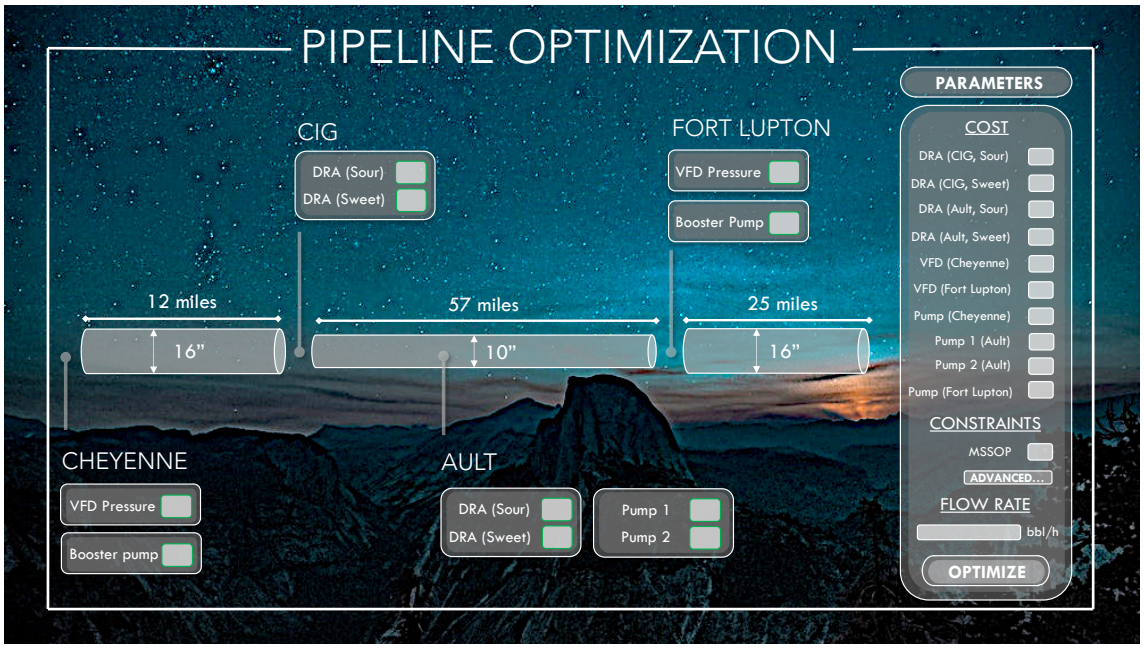


Figure A.24: Conceptual software design for the optimization tool.

violated. If none are violated, the optimal set points are recommended to the operators. To ensure implementability, the VFD amperes are first converted to pressure set points because the operators do not know how to operate pumps by amperes.

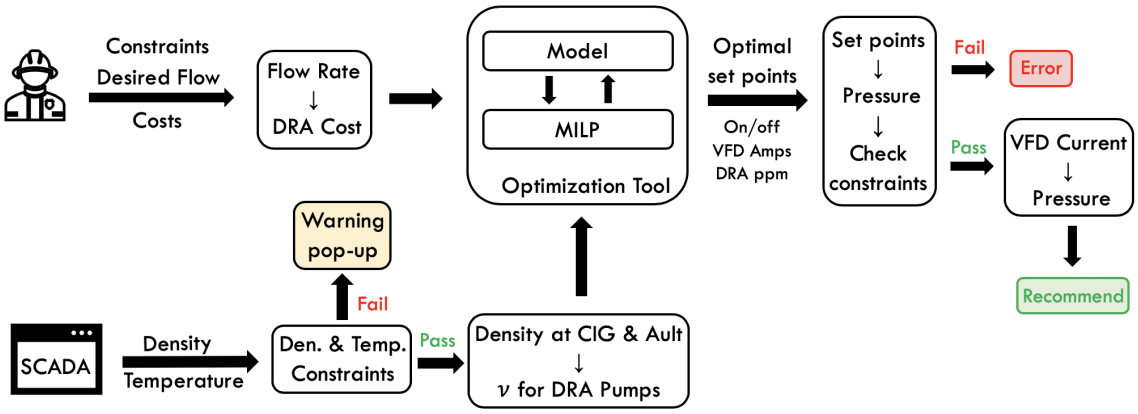


Figure A.25: Internal flow diagram of the product.

A.2.6 Cost Savings and Impact on Society

Operating costs for running the pipeline using the optimization tool was compared to historical values to identify the potential cost savings. In 2019, the pipeline was plagued with unexpected shut downs and maintenances; thus, the data set ranging from January 2018 - May 2018 was used for this comparison due to the stability

of operations. Both the actual and optimal costs are calculated using the same formulation to ensure a fair comparison.

First, the total monthly DRA cost was calculated by:

$$Cost_{DRA} = \sum_{i=1}^{43{,}200} \frac{1}{60} \left(f_{light}(Q, ppm) + f_{heavy}(Q, ppm) \right) \tag{A.30}$$

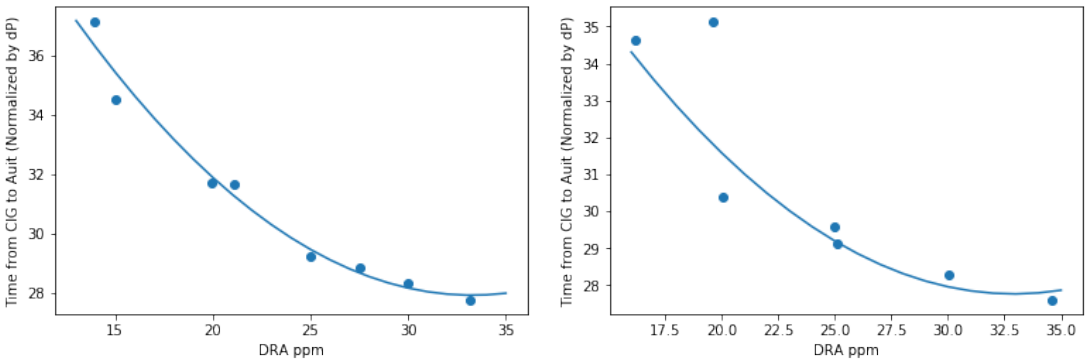
where the functions represent Equations A.28 and A.29 for computing the DRA cost per hour. Here, i denotes the i^{th} minute in a particular month. The costs are divided by 60 to obtain the per minute costs and are summed over 43,200 entries representing the total minutes in a 30 day month. Likewise, the monthly power costs are calculated by:

$$Cost_{power} = \frac{p}{1000} \sum_{i=1}^{720} \sum_{j=1}^{6} I_j \cdot V_j \tag{A.31}$$

where i denotes the i^{th} **hour** in a particular month and j denote the j^{th} pump. The 720 corresponds to the amount of hours per 30 day month. p denotes the power cost in kWh and is divided by 1000 for unit consistency. Additionally, I_j and V_j denotes the current and voltage for the j^{th} pump.

The actual and optimal costs for DRA and power are shown in Table A.25[6]. It can be seen that the total savings range between -3.7% to 17.6%. The savings were realized by using less DRA while running the pumps harder. For this initial study, interaction effects between pumps and DRA were not explicitly considered. Additionally, the effectiveness of the VFDs (i.e., flow rate as a function of current) were assumed to be linear in the region of study; this may not be the case in actual production because pumps' efficiency changes with accordance to the pump curve [64]. Likewise, DRA effectiveness was also assumed to be linear. Preliminary data analysis showed a quadratic relationship between flow rate and increased DRA (Figs. A.26a and A.26b and below for detailed DRA study); however, a MILP was used for the first release of this product so a linear curve was used instead. From Table

[6]Actual costs are removed for confidentiality; however, percentage savings are still present.



(a) Light DRA (b) Heavy DRA

Figure A.26: Flow rate as a function of DRA ppm.

A.25, it can be seen that the optimization tool shows great promise in terms of cost savings. A loss was incurred in April because the industrial sponsor used only 10 - 15 ppm of DRA for a large part of the month. In the optimization, minimum DRA was constrained to 20 ppm. In all other months, the algorithm was able to achieve a cost savings.

Figures A.26a and A.26b shows the flow rate change (normalized by dP) as a function of DRA ppm. The flow rates were normalized by pressure to ensure only the DRA's effect was explored. The flow rate was normalized by:

$$Time_{norm} = \frac{Time}{P_2 - P_1} \tag{A.32}$$

where P_1 and P_2 are the pressures at the starting and ending pump stations, respectively. A similar relationship is expected to exist for pumps due to the pump curves. To further improve the accuracy of the optimization tool, the weights of the model can be switched to functions.

A detailed analysis of the cost savings for January - May 2018 are shown in Table A.26. The savings per barrel at higher flow rates were more significant compared to savings at lower flow rates.

Ultimately, expected operating cost savings were estimated to be between 7% to 9%. Average saving per barrel was $ – (8%), increasing as the flow rate increased. Moreover, savings for transporting one barrel of crude one km was $ – USD. The

Table A.25: Optimal vs. actual costs ($USD 000's) for January - May 2018.

		January	February	March	April	May
Actual	DRA cost	-	-	-	-	-
	Power cost	-	-	-	-	-
	Total cost	-	-	-	-	-
Optimal	DRA cost	-	-	-	-	-
	Power cost	-	-	-	-	-
	Total cost	-	-	-	-	-
Diff.	Dollars	**-**	**-**	**-**	**(-)**	**-**
	% Improvement	**8.0**	**17.6**	**7.8**	**(3.7)**	**9.7**

Table A.26: Cost savings per barrel of crude shipped.

	January	February	March	April	May
Ave. flow rate (bbl/h)	2195	2400	2256	2248	2575
Saving / bbl per hour ($)	-	-	-	(-)	-
Saving / bbl / hour / km ($)	-	-	-	(-)	-

total expected savings per year for Line B was estimated to be $ – USD. However, the results are only realized if the pipeline is operating within the ranges provided in Tables A.18 and A.19.

Benefits for Society

On the financial side, this optimization tool has shown to reduce operating costs of transporting one bbl/h/km by $0.04 USD[7] for an industry leading energy company. Cost savings can be even more significant for lower tier, in-experienced companies. In Canada alone, 175,000 barrels of crude is produced per hour and transported at an average velocity of 9.5 km/h [159]. At $0.04 USD savings per barrel/km, the annual savings result to approximately $583M USD ($784M CAD). Expanding to North America, where approximately 742,000 barrels of crude is produced per hour, the savings jump up to $2.5B USD ($3.4B CAD) [160]. Moreover, the savings here does not include inter-refinery and inter-market pipelines; only produced crude were included. The savings can be reinvested into greener products, resulting in signifi-

[7]A significantly rounded and slightly altered value to maintain confidentiality. An estimated value was provided in order to quantify the impact to society.

cant reduction of green house gases (GHG). In 2018, the social cost of carbon was valued at $46 USD per metric ton [161]; $2.5B results in a 54.3M metric ton of reduced GHG.

On the operational level, this tool standardizes the operation of complex pipelines. Given a set of inputs, $I = \{Q, Costs, x\}$, the output of the optimization tool will be identical each time due to the linear model. Thus, operators on different shifts and with different experience levels will operate very similarly if this tool is used. Additionally, this normalization reduces variance in operations which reduces depreciation of process equipment. Furthermore, the free cash flow can be reinvested to obtain higher throughputs, which in turn will increase revenue and increase tax dollars. Finally, the tool will also increase operational safety, which may result in the Alberta Energy Regulator (AER) allowing higher throughputs on pipelines currently operating under capacity.

A.3 Pipeline Project Conclusion

An adaptive outlier-robust optimization tool based on linear parameter-varying (LPV) models was presented in this study. Objectively, the project aims to: 1) introduce standardization in operations to reduce turbulence in the pipeline while increasing efficiency of less experienced operators. 2) Open-loop automation of the pipeline; reducing repetitive and painful tasks for the operators while freeing up their time for more attention demanding tasks. First, exploratory data analytics was performed to pick the most relevant features for the model. It was identified that the flow rate followed a bi-modal distribution. The data was clustered into the two operating conditions where cluster 1 averaged 2500 bbl/h and cluster 2 averaged 1900 bbl/h. Then, a LPV model consisting of two models was built; one model for each cluster. The models predicted the output flow rate of the pipeline as a function of 10 inputs and 8 states. The MAE, RMSE and R^2 for model 1 on the testing data set was 96, 120, 0.78, respectively. Likewise, the performance metrics were 85, 110, and 0.57 for model 2. These models were then used for optimization via a mixed

integer linear program where the outputs were the optimal steady state set inputs based on specified operating costs. By comparing the operating costs incurred from the actual operation with the optimal set points, cost savings of $ – USD could have been realized in 2018. The average cost reduction was $8%, with an average savings margin of $ – USD per barrel/km transported. A preliminary impact study was also conducted and found that $583M USD in cost savings can be realized if this tool was applied to all exporting pipelines in Canada. Intangible benefits include increased efficiency for lower experience level operators, increased stability in the pipeline, and reduced wear-and-tear from turbulence.