NTNU

Norwegian University of Science and Technology

TKP4170 Specialization Project

# Application of Reinforcement Learning and Neural Network in Process Control

**Vinh Phuc Bui Nguyen**

Supervisor: Sigurd Skogestad

Co-supervisor: Saket Adhau

December 17, 2020

# Contents

# List of Figures

# Acknowledgement

I would like to express my gratitude to my supervisor, Professor Sigurd Skogestad, for his guidance and the fruitful discussions we had. I would like to thank my co-supervisor Saket Adhau for his patience and great advice on Neural Networks. I also would like to express my appreciation to Postdoc Jose Otavio Assumpcao. The project would not have been possible without his excellent support and valuable feedback.
Finally, I would like to thank my family and my friends for their encouragement, especially my girlfriend, Le Thi Dieu Thao, and Nguyen Phung Phuong Nguyen.

# Abstract

The project consists of two distinct parts. In the first part, the applicability of Reinforcement Learning in various Chemical Engineering problems (regulatory control, plant economics optimization, etc.) has been analyzed and evaluated. In the second part, Neural Network has been studied to create a nonlinear process model for the use of steady-state Real-Time Optimization. The process studied is the one in a Gas-Lift lab rig.

# 1 Introduction

The field of Machine Learning (ML) is developing rapidly, especially in the branch of Reinforcement Learning. One remarkable achievement of Reinforcement Learning is the AlphaGo Zero program of DeepMind. It has learned to play the complex game of Go by itself and achieved a human-superior level [1]. This achievement has proved how powerful the method is, and it is interesting to see whether Reinforcement Learning could be a good solution to Chemical Engineering problems. Therefore, it is the aim of this project to identify which particular Chemical Engineering problems would be beneficial from Reinforcement Learning.

Many studies have been conducted on the applications of Reinforcement Learning in regulatory control and plant economics optimization [2][3][4]. In these papers, the authors focused on improving the Reinforcement Learning algorithms and demonstrating the applicability of these through numerical experiments. In this work, another approach has been used. Instead of using numerical experiments, we analyzed the properties of the problems and of Reinforcement Learning in order to make arguments about the applicability of Reinforcement Learning in each problem.

Deep Learning is another branch of ML that is growing rapidly. Many applications of Deep Learning have been developed successfully, such as image recognition, speech recognition, and language translation. In Deep Learning, artificial neural networks are used to find an appropriate mapping between the inputs and the outputs. The neural networks could approximate any nonlinear function without any prior knowledge provided [5]. Due to this ability, it has great potential for modelling chemical processes. Modelling chemical processes, especially the nonlinear ones, from the first principles is usually challenging, time-consuming, and expensive [6]. Therefore, neural networks have been considered as an alternative method to create models of chemical processes [7][8]. In this project, the neural network has been applied to model the nonlinear process in a Gas-Lift lab rig for the use of steady-state Real-Time Optimization. The process in the Gas-Lift lab rig is discussed in subsection 1.1.

As the two topics studied - the applications of Reinforcement Learning, and the application of Neural Network - are highly different, they are discussed separately in two different chapters of the project report. Therefore, the report includes 4 chapters: Introduction, Application of Reinforcement Learning in Process Control, Application of Neural Network in Economics Optimization of the Gas-Lift lab rig, and Conclusion.

## 1.1 The Gas-Lift lab rig

Gas lift is a technology to increase oil flow rates from a reservoir when the reservoir pressure is not sufficiently high. A gas-lift oil well is represented in Figure 1. The gas is injected into the tubing through the gas lift valve at the bottom of the well and mixed with the fluid from the reservoir. The density of the fluid becomes lower, and the pressure there decreases due to lower hydrostatic pressure. This helps to increase the oil flow rate from the reservoir [9]. As there are several oil wells in a reservoir and the gas capacity is limited, it raised the problem of gas distribution. We wish to distribute the gas such that the total oil flow rate - the sum of oil flow rates from all wells - is highest, while the gas capacity constraint is satisfied. The problem can be solved using Real-Time Optimization.

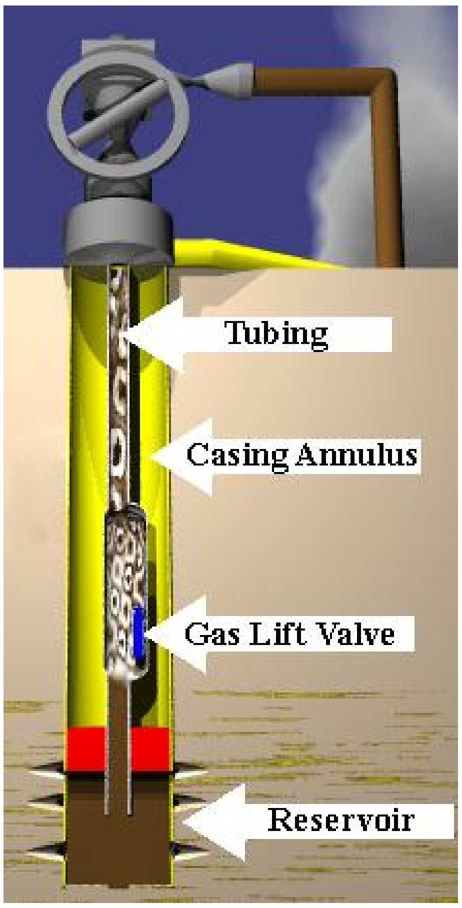

**Figure 1:** *A gas lift oil well (Source: Wikipedia)*

The Gas-Lift lab rig is a physical model of an oil well network. It has three wells, each connected to a gas injector. The gas injectors are controlled by PI controllers. Water is used in the rig to represent oil. A variable speed pump withdraws water from the storage tank and pumps it into the wells before it reaches the risers and returns to the tank. A valve is installed in each well, and it could be manipulated to adjust flow restriction. A simplified representation of the process in the Gas-Lift lab rig is shown in Figure 2. There are 13 available measurements in the rig: 3 liquid flow rates (outputs), 3 well pressure values (outputs), 3 gas flow rates (inputs), 3 valve openings (disturbances), and pump rotation (disturbance).

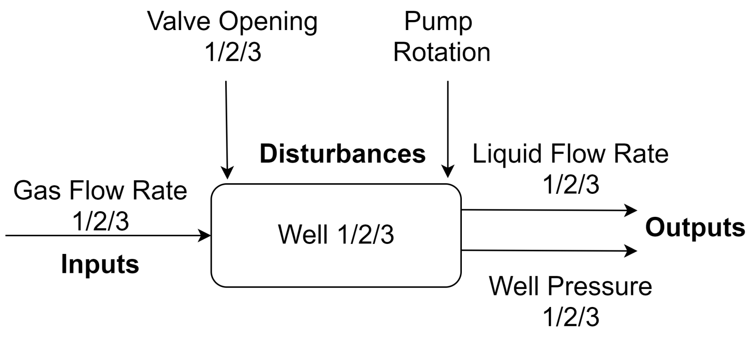

**Figure 2:** *The process of the Gas-Lift lab rig*

# 2 Application of Reinforcement Learning in Process Control

## 2.1 Reinforcement Learning - A short introduction

In Reinforcement Learning (RL), there is an agent whose purpose is to achieve a given goal in an environment. The agent does not receive instructions on how to attain its goal but must learn about that through its interactions with the environment over time.

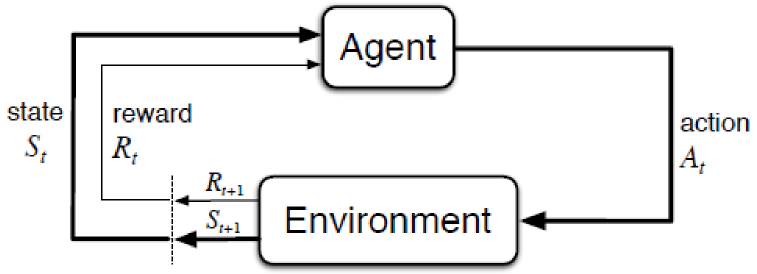

**Figure 3:** *The Markov Decision Process*

Understanding how an agent learns requires knowledge about the Markov Decision Process (MDP) . A general MDP is represented in Figure 3. At each iteration $t$, the agent receives information about the current environment's state $S_t$, based on which it selects an action $A_t$ to implement on the environment. The action causes the environment's state to change from $S_t$ to $S_{t+1}$. Information about $S_{t+1}$, together with a reward signal $R_t$, is sent back to the agent and a new iteration begins. By means of the reward, we convey the goal to the agent.
Let's take the example of playing chess to illustrate these definitions. In playing chess, the environment is the chess game. The actions are selecting which piece to use and how to use that piece. The states are the current situation on the board. Because we would like to win the game, we can define the reward is equal to 1 when we reach the "winning" states, and 0 otherwise.
The agent always tries to maximize the total reward over time, which is also called the return. This is possible through the use of value functions. The definition and formulation of action-value functions are given in [10]:
*"We define the value of taking action* a *in state* s *under a policy* $\pi$*, denoted* $q_\pi(s, a)$*, as the expected return starting from s, taking the action* a*, and thereafter following*

*policy $\pi$"*

But what is a policy, and why it is important to the definition of value functions? It is defined in [10] as:

*"A policy is a mapping from states to probabilities of selecting each possible action"*

A policy usually used in Reinforcement Learning is $\epsilon$-greedy with $\epsilon$ is a small scalar, such as 0.2. We could interpret this policy as, given a state S, there is a chance of $(1 - 0.2) \cdot 100\%$, or 80%, that the agent will choose to implement the "best" action A with the highest value function $q_\pi$(S, A), and 20% that it will choose other actions. The $\epsilon$ could never be exactly 0. It is usually the case for RL problems that we do not have the exact value functions in advance. Therefore, these value functions are usually estimated from the agent's experiences, which are not necessarily close to their correct values. Therefore, it is impossible to know whether the "best" action based on current value functions is actually the best. It is important for the agent to occasionally try other actions and collect more data about these so it could have better estimates of value functions, which is known as the dilemma of exploration-exploitation in Reinforcement Learning. Not only does $q_\pi$(s, a) depend on $s$ and $a$ the agent might take at the immediately next step, it also depends on the actions to be taken in all later steps. As a consequence, value functions must be defined in the context of a specific policy.

## 2.2 Overview of Reinforcement Learning methods

As previously discussed, RL algorithms use value-function to select actions. For problems with finite, small space of state-action, the value functions could be stored and updated in a lookup table. Algorithms that use lookup tables are called Tabular RL. Examples of Tabular RL methods are Monte Carlo, SARSA, and Q-learning [10].

For problems with infinite or large space of state-action, it is impossible to store all value functions in a table. Hence, it is essential to find functions which could represent value functions adequately in these cases. If a deep neural network is used for this purpose, the algorithm belongs to the Deep Reinforcement Learning (Deep RL) category. An example of this type is the famous Double Q-learning algorithm (DQN) by Silver et. al [11].

Another algorithm, REINFORCE, also belongs to Deep RL. However, REINFORCE uses a neural network to learn a policy directly, instead of deriving it from value functions [10]. Such methods are also called Policy Gradient methods. There also exist actor-critic methods, such as asynchronous advantage actor-critic (A3C) by Mnih et al. [12], which use deep learning to approximate both policy and value functions.

According to [13], all mentioned Deep RL methods are in the first generation of Deep RL, which are “powerful but slow”. These methods could attain human-level performance on tasks such as playing chess or Go, but they require significantly more data than humans. The second generation of Deep RL algorithms is being developed. They are aimed to utilize data more efficiently and be able to learn from fewer data points.

## 2.3 Applicability of Reinforcement Learning in Chemical Engineering problems

To identify which Chemical Engineering problems are appropriate for the use of Reinforcement Learning, the features of an application that shaped the development of RL, playing chess, are analyzed. A chess game is irreversible: it is not possible to reverse the game and reselect. Also, an action in the present decides states and actions accessible to the agent later, for example: sacrificing the queen implies removing it from the future strategies [10]. Finally, an action has delayed consequences [10]. For instance, a move is not known to be good or bad until many steps later, even until the end of the game.

The first two properties require us to have accurate evaluation of how “good” is each action in order to solve the problem, while the last two properties make it difficult to evaluate actions accurately. But the trial-and-error approach of RL is effective in resolving this challenge. As it is difficult to predict all future situations emerging from current possible actions, RL explores many different options. It also uses value functions to record the corresponding expected total rewards associated with these actions and make better decisions the next time it is in a similar situation. The value functions also resolve the third challenge: action with a higher value function/better in the short-term is expected to result in a higher total reward/better in the long-term. The trial-and-error approach of RL may require long training time, but the benefits to these kinds of problems are enormous. However, problems not having these features do not fully benefit from RL. Unfortunately, this is usually the case in chemical engineering.

### 2.3.1 Applicability of RL in steady-state Real Time Optimization (RTO)

Consider the problem of steady-state plant economics optimization. The task here is to find optimal inputs that minimize the cost function of the process. It might be challenging, but not in a fashion similar to what we discussed. If an input level is found to be not optimal, the process could be brought back to its previous state by simply setting the input back to its past value and waiting. Moreover, no matter which level is selected in the present, it will not affect the set of possible states and input levels in the future. As a result, the actions do not have delayed consequences either. Therefore, the steady-state RTO problems do not have the discussed properties, and RL is not an effective solution here.
Although Kody et al. [2] utilized RL to optimize economics, they also stated that "steady-state RTO deviates somewhat from the underlying principles that led to the creation of reinforcement learning.". As a result, they used an actor-critic RL architecture, which is quite different from the discussed RL. Sebastien and Mario [3] also used RL in economics optimization. However, in their work, RL was used to tune the economic Model Predictive Controller instead of control the plant directly.

### 2.3.2 Applicability of RL in dynamic control

Here the possibility of using RL as a controller in the regulatory control layer is investigated. The actions are selecting input levels, and the goal is to achieve given output setpoints. As requirements, an RL-based controller should be able to track the output setpoints and reject the disturbances.
The dynamic control problem shares more common features with playing chess than steady-state RTO. It is obvious that in control, actions/inputs also have delayed consequences due to the dynamcs of the plant. However, the two problems still have differences. Although the action in the present decides states in the future, the set of possible actions does not change: the current input level determines the next states of the process (together with the current state, future inputs, and disturbances), but it does not limit future accessible input levels. Due to this property, the problem is not irreversible either: as all input levels are available, there probably exists an input sequence that could bring the plant back to its previous state.
We could make the control problem more difficult and "irreversible" by requiring optimal control, but evaluating actions still does not pose a challenge here as in playing chess. Because of the delayed consequences in chess and general RL problems,

an action with a higher immediate reward could have a lower total return compared to actions with lower immediate rewards [10]. In short, being good in the short-term does not necessarily mean being good in the long-term. In contrast, MPC with a one-step prediction horizon has been used in control applications, for example, in [14].

The reason why the trial-and-error approach of RL is not effective in chemical process control has been discussed. In fact, it might be inappropriate to apply standard RL algorithms. A process may have many disturbances, both measured and unmeasured, and as stated in [15]: "Disturbance rejection is often the main objective of process control.". As the disturbances change, and they may change frequently, the dynamic of the process also changes. This is equivalent to changing the environment, which forces the agent to restart the time-consuming learning process. It is unacceptable as a controller is expected to keep the output close to its setpoint all the time.

The difficulty in re-training could be realized in the article about using RL in process control by Spielberg et al. [4]. It took their agent 1500 seconds of training time to learn the process again when it was changed, compared to 2700 seconds in its first training. This limitation must be overcome before RL could be used in process control. Currently, the second-generation RL algorithms can learn about similar but different environments in a fast and effective manner by combining Deep RL with meta-learning and episodic memory [13]. It is interesting to see if these could solve the problem of disturbance rejection in process control.

As discussed earlier, trial-and-error is essential in playing chess but not in process control, and it restricts the use of RL in the field. One possible way to adapt RL to process control is to direct the agent's trial effort at each step on a narrower range of sensible input levels, instead of the whole range of possible levels, by the use of optimization methods.

**2.3.2.1 The problem of setpoint tracking** : Although disturbance rejection is challenging for RL-based controllers, setpoint tracking is not. Even if the controllers are set to achieve only one particular setpoint during training, they can track other setpoints without conducting trial-and-error again. Assume that during training, RL tries to achieve an output setpoint $y_{sp1}$. At each time step, it encounters a state $y_t$ of the system, then selects an input level $u_t$ to implement and observes the next state $y_{t+1}$. It is important to mention that RL only requires pieces of information about $\{y_{sp}, y_t, u_t, y_{t+1}\}$ to estimate action-value functions and find an optimal policy to achieve $y_{sp}$. If the training data $\{y_{sp1}, y_t, u_t, y_{t+1}\}$ was recorded and we would like RL to achieve another setpoint $y_{sp2}$, we just need to replace $y_{sp1}$ in the training data by $y_{sp2}$ and provide the relabelled data $\{y_{sp2}, y_t, u_t, y_{t+1}\}$ to RL. RL could then find an optimal policy for $y_{sp2}$ and does not have to repeat the trial-and-error steps.

Let's take an example to illustrate this strategy. Assume that a RL-based controller was set to achieve an output of 7.5 during training. $\{y_{sp}, y_t, u_t, y_{t+1}\}$ at all time steps were recorded and stored in a table with 4 columns (the first column stored values of $y_{sp} = 7.5$, the second column stored values of $y_t$, and so on). After the training, RL had learnt to achieve the setpoint of 7.5, and it was then set to achieve a new output of 8.5. The first column of the table was relabelled with the new setpoint of 8.5 (items in the other 3 columns $\{y_t, u_t, y_{t+1}\}$ were not changed) . New action-value functions were calculated from data in the relabelled table and provided to RL. These value functions were used directly by RL without updating.

To demonstrate the effectiveness of this strategy, we conducted a numerical experiment. In this experiment, we used a strategy that is quite different but has a working mechanism identical to that of the one described. We used two controllers, 1 and 2. Initially, we trained controller 1 to track a setpoint of 7.5. However, instead of letting controller 1 to record $\{y_{sp}, y_t, u_t, y_{t+1}\}$ at all time steps, we commanded it to calculate the action-value functions for the both setpoints (7.5 and 8.5) simultaneously. After the training of controller 1 was finished, we transferred the value functions calculated for the setpoint 8.5 to controller 2 and let it to control the process without any training. The results of the experiment were represented in Figure 4. It could be seen that controller 2 has successfully achieved the new setpoint although no extra trial-and-error or updating on the new setpoint had been conducted.

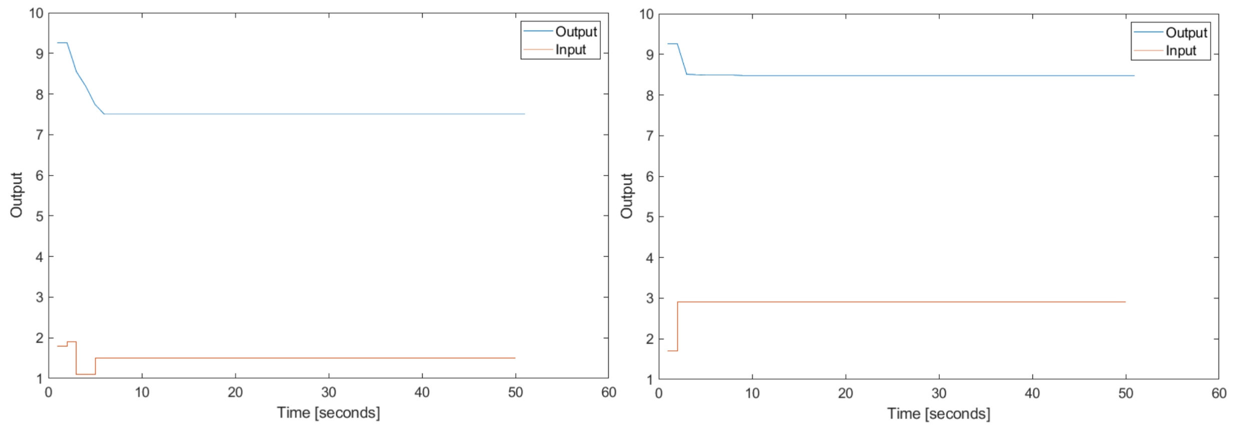

**Figure 4:** *Results of the Multi-setpoint Tracking experiments: Performance of controller 1* $y_{sp} = 7.5$ *(left), and performance of controller 2* $y_{sp} = 8.5$ *(right)*

All the code relating to this experiment can be found in the Appendix.

### 2.3.3 Other Chemical Engineering problems

We have considered the possibilities of using RL in steady-state RTO and dynamic control. In both applications, RL is used to make low-level decisions: selecting an input level to implement. It is worth investigating whether RL could be involved in making high-level decision problems, such as plantwide control design. The decisions to be made here are which variables to be controlled variables (CVs), which variables to be manipulated variables (MVs), and how should we pair the MVs with the CVs [16]. Although we could eliminate many control structures using engineering insights, possibly there are still many feasible structures left to be considered. Plantwide control design problem has the first and the second properties if we do not allow "undo and redo": if we already used a manipulated variable (MV) to control a controlled variable (CV), then we cannot pair that MV with another CV until a new round of design. The actions here also have delayed consequences which could not be observed until the completion of the design, such as economic loss in the presence of disturbances, or "snowballing" phenomenon that may occurs by our choice of the location of the throughput manipulator (TPM). These effects are evaluated only in final simulations after CVs-MVs pairing has been completed. Finally, frequent change of learning environment does not exist in this problem of plantwide control design as in dynamic control. Therefore, RL could be used here as a tool that aids us to find new design rules.

There are still low-level decision problems in chemical engineering suitable to apply RL, for example, erosion control in gas-lift oil wells. In oil wells, critical components degrade more quickly when increasing the volume flowrate, partially due to particle erosion [17]. Rapid erosion may result in expensive unscheduled maintenance of the system. Therefore, it is important to adjust the well's operation based on the erosion rate of critical components. It is not difficult to understand why this problem is potential for RL. The erosion process is naturally irreversible (the first property). Volume flowrates in the past affect the current erosion rate, which in turn determines acceptable volume flowrates in the future (the second property). Due to its interactions with erosion rate, volume flow also has delayed consequences (the third property). Yet, we must take into account that data collection here takes a significant amount of time as the erosion is an intrinsically slow process.

# 3 Application of Neural Network in Economics Optimization of the Gas-Lift lab rig

## 3.1 Introduction

Being profitable is the ultimate goal of process plants. With the increasingly stricter requirements imposed on the production, optimization of process operations becomes more and more important for plants to achieving this goal [18]. Typically, real-time optimization (RTO) is implemented to improve the economic productivity of the processes [19]. RTO usually requires nonlinear steady-state models of the process [20], preferably ones derived from first principles [21]. However, obtaining these mechanism-driven models might be challenging [22].
Artificial Neural Networks (ANN) has been used to model complex nonlinear processes since the 1990s [23]. This is due to its ability to approximate any nonlinear functions with no prior knowledge required [23][5]. ANNs have also been applied successfully as process models in RTO [7][8]. Therefore, this part of the project investigates the use of ANN to model the cost function of the Gas-Lift lab rig and how this model could be useful in RTO.

## 3.2 Neural Network: An Overview

This part of the report introduces basic concepts in Deep Learning. As Deep Learning is a vastly broad field, this introduction will focus on types of NN that are relevant to our problem, especially on the information essential to NN implementation. Modern Neural Networks could be divided into three main types: feedforward, recurrent, and convolutional [5]. The convolutional networks are usually used in image-like data processing, and we do not consider them here [5].

### 3.2.1 Feedforward Neural Networks

Feedforward network is the most basic deep learning model. It offers a solution to the function approximation problem, where we wish to approximate a correct function $y = f^*(x)$. The network approximates $f^*(x)$ by another function $y = f(x, \theta)$ and tries to find a value $\theta^*$ of the parameter vectors $\theta$ that results in the best approximation.

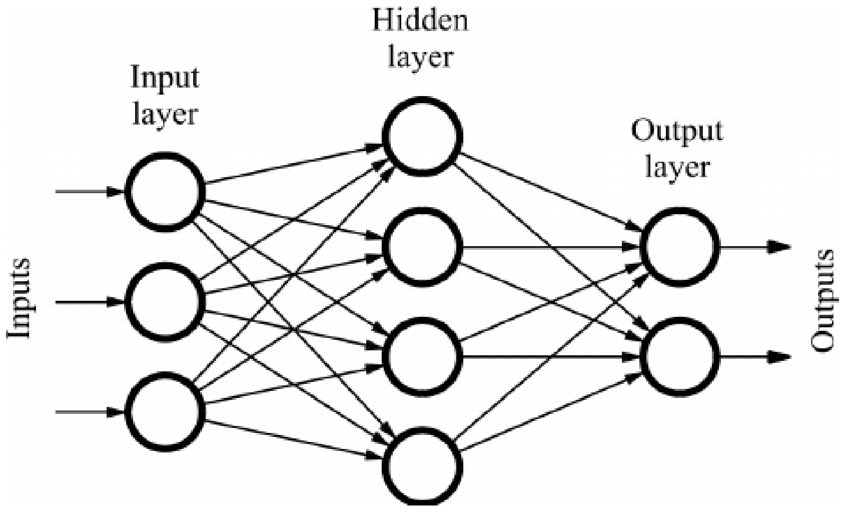

**Figure 5:** *Structure of a Feedforward Neural Network (Source: deepai.org)*

The basic unit of construction in a neural network is the neuron. Several neurons are stacked together into a layer. A network has one input layer, one output layer, and one or several hidden layers. These layers are arranged adjacent to each other, as represented in Figure 5. It could be seen that there is no connection between neurons in the same layer. A neuron only connects to neurons in the previous layer and the following one. To understand how a network works, we will investigate how a single neuron works. A single neuron is represented in Figure 6. The input of this neuron x is calculated from the outputs of neurons in the previous layer $l_1, l_2, ..., l_N$ as:

$$x = \sum_{i=1}^{N}(w_i \cdot l_i) = w_1 \cdot l_1 + w_2 \cdot l_2 + ... + w_N \cdot l_N \tag{1}$$

where $w_i$ are weights of the connections between the neurons. The neuron then calculated its output y as:

$$y = \phi(\sum_{i=1}^{N}(w_i \cdot l_i) + b) \tag{2}$$

where $\phi$ denotes an activation function, and b denotes the bias of the neuron. This output y is then fed into neurons in the next layer for further calculations. The set of all connection weights $w$ and neuron biases in the network is the parameter vector $\theta$ that has to be learnt during training.

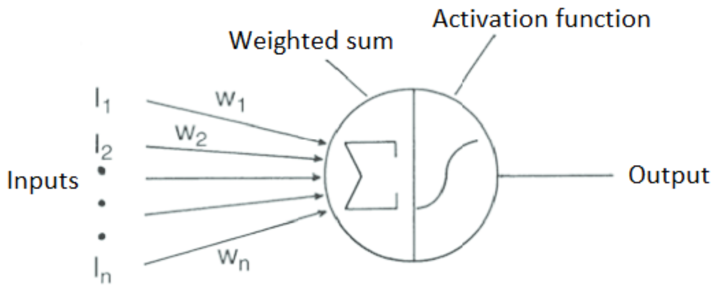

**Figure 6:** *Single neuron in a Neural Network (Source: researchgate.net)*

There are 3 activation functions frequently used, which are: sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). Any activation function in these three could be used in a feedforward network, but ReLU is becoming more and more popular.

**3.2.1.1 Optimizer:** How could the network find an optimal value of $\theta$? Let's take a single variable regression problem as example. During training, training data $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$ are shown to the network. For each $x_i$, the network predicts a corresponding $\hat{y}_i$ (forward propagation). A cost function $J(\theta)$, such as mean squared error, could be defined over n predicted and actual values as:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{3}$$

The network uses a gradient-based optimization algorithm, or optimizer, to update $\theta$ such that the cost function is minimized (backward propagation)[5]. The optimization algorithms could be divided into 2 categories: deterministic gradient methods, which use the entire training set, and stochastic methods, which only use part of the training set [5]. For large-sized training data, it may take a considerable amount of time for the deterministic algorithms to process all training examples and update the parameters. Therefore, stochastic algorithms are more suitable if we have a large dataset. An example of deterministic gradient algorithms is the Levenberg-Marquardt algorithm. Some popular stochastic optimizers are listed here: stochastic gradient descent (SGD), RMSProp, Adam [5].

The network alternates between forward and backward propagations until the completion of training. The final value of $\theta$ is denoted as $\theta_f$. The MSE training error over the training set is defined as:

$$\frac{1}{N}\sum_{i=1}^{N}(f(x_i,\theta_f)-y_i)^2 \tag{4}$$

Besides training data, we usually also have test data, which is the data the network has not encountered during training. A test error over the test set could be defined similarly to the training error. The test error is important in evaluating if a network overfits or not, as described in paragraph 3.2.1.4

**3.2.1.2 Parameters Initialization:** The aforementioned optimization algorithms find the optimal value of $\theta$ by iteration, and they need an initial point to start. Providing different initial points to an algorithm could make it to converge to different results at different rates, or even not converge [5]. Considering the importance of the initial point, strategies to initialize network parameters have been developed. The most common initialization strategy is the normalized initialization suggested by Glorot and Bengio [24]. This is the default weight initializers in Keras and MATLAB.

**3.2.1.3 Input Normalization:** LeCun [25] suggested that normalizing the input variables in the training data helps the optimization algorithms to converge faster. If the network has M input elements and N training examples, the input vectors could be denoted as $\{x_{1j}, x_{2j}, ..., x_{Mj}\}$ with j = {1,2,...,N}. The inputs could be normalized according to this formula:

$$\hat{x_{ij}} = \frac{x_{ij}-\bar{x_i}}{\sigma_{x_i}} \tag{5}$$

where $\bar{x_i} = \frac{\sum_{j=1}^{N} x_{ij}}{N}$ and $\sigma_{x_i} = \sqrt{\frac{\sum_{j=1}^{N}(x_{ij}-\bar{x_i})^2}{N-1}}$, with i = {1,2,...,M}.

**3.2.1.4 Overfitting and Regularization:** The trained model must perform well on previously unseen data, or equivalently, has the ability to generalize [5]. When the model gives good approximation for the training data but has poor generalization performance (training error is significantly better than test error), we said overfitting has occurred. Overfitting is a major challenge in machine learning, and many methods have been developed to avoid this phenomenon. These methods are called regularization, with L2 regularization as an example. A detailed discussion on regularization could be found in [5].

**3.2.1.5 Tuning a Neural Network model:** Basic components of a feedforward neural network have been discussed in previous sections. We could list some hyperparameters of a network associated with these components: number of hidden layers, number of units in each layer, the activation function, the learning rate of the optimizer, the regularization parameter, etc. By trial-and-error, we hope to find appropriate values of these hyperparameters and as a result, an adequate approximation of the original function.

### 3.2.2 Recurrent Neural Networks

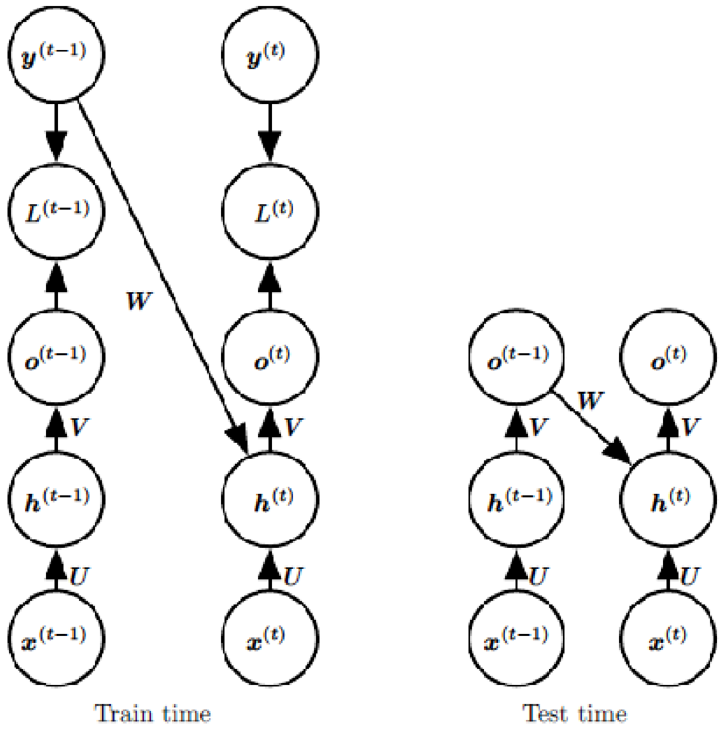

**Figure 7:** *Structure of the Recurrent Neural Network with connections from the output to the hidden layer [5]*

Recurrent neural network (RNN) is a type of neural network that is specialized in processing sequential data $\{..., x^{(t-1)}, x^{(t)}, x^{(t+1)}, ...\}$. It looks like a feedforward network with extra connection from the output layer to a hidden layer or from one hidden layer to one of its previous hidden layers. The structure of a RNN with an extra connection from the output layer to a hidden layer has been shown in Figure 7. To understand the structure of RNNs, let us consider an example of using a RNN to model a dynamic system represented by:

$$y^{(t)} = f(y^{(t-1)}, x^{(t)}, \theta) \tag{6}$$

where y is the output of the system, u is the input, and theta is the parameter vector. It could be seen that not only does the current output $y^t$ depend on the current input $x^{(t)}$, but also on the previous output $y^{(t-1)}$. In RNN, information about the previous outputs is communicated through the connections from the previous outputs $y^{(t-1)}$ (training time) or the predicted outputs $o^{(t-1)}$ (test time) to the hidden layers $h^{(t)}$ that are processing the current inputs $x^{(t)}$. One challenge of the recurrent network is to learn the long-term dependencies. As the gradients propagate across many stages of the network, they vanish or explode exponentially. Therefore, the weights of the long-term interactions are significantly smaller than those of the short-term, and it takes longer time for the network to learn the long-term interactions [5]. Advanced RNN structures have been introduced to solve this problem, including the Long Short-Term Memory model (LSTM) and networks that use the Gated Recurrent Unit.

## 3.3 Methodology

### 3.3.1 Identify the inputs, outputs, and type of the Neural Network

The process in the lab rig has been discussed in subsection 1.1. To formulate the economics optimization problem of the lab rig, we have to define the cost function J. J can be formulated as:

$$J = -\alpha_{liq} \cdot \Sigma w^{liq} + \alpha_{gas} \cdot \Sigma w^{gas} \tag{7}$$

where $\alpha_{liq}, w^{liq}, \alpha_{gas}, w^{gas}$ are liquid price, liquid flow rate, gas price, and gas flow rate, respectively. To simplify the problem, we assume the gas cost is insignificant compared to the liquid cost and neglect the gas terms. The cost function becomes:

$$J = -\alpha_{liq} \cdot \Sigma w_i^{liq} \tag{8}$$

We also need a model of the system, which represents the relations between $w_i^{liq}$ and $w_i^{gas}$. In this project, we use neural networks to represent these relations. Ideally, the outputs of the NN model should be the steady-state liquid flow rate, and the inputs should be the gas flow rate and pressure value. However, as the pressure measurements are highly noisy, in this project we used gas flow rate $w^{gas}$, valve opening value $\chi$, and pump speed $\Omega$ as inputs of the model. The optimization problem can be formulated as:

$$\begin{aligned} \min_{w_i^{gas}} \quad & J = -\alpha_{liq} \cdot \Sigma w_i^{liq} \\ \text{subject to:} \quad & w_i^{liq} = f_i(w_i^{gas}, \chi_i, \Omega) \\ & \Sigma w_i^{gas} \leq W_{max}^{gas} \end{aligned} \tag{9}$$

Due to limited time, the NN model will be trained to represent the relation between the liquid flow rate and the gas flow rate of one well only. Moreover, it is difficult to train a steady-state model using our training data since there are only around 20 steady-state liquid flow rates there. Therefore, the transition data is also utilized and the resulting model is a dynamic model. Hence, the output of the system is $w_{k+1}^{liq}$ and the inputs are:
$(w_k^{liq}, w_{k-1}^{liq}, ..., w_{k-N_o-1}^{liq}, w_k^{gas}, w_{k-1}^{gas}, ..., w_{k-N_i-1}^{gas}, \chi_k, \chi_{k-1}, ..., \chi_{k-N_i-1}, \Omega_k, \Omega_{k-1}, ...$
$, \Omega_{k-N_i-1})$
where k is the current time step, $N_o$ is called the output lag and $N_i$ is called the input lag. The model could be expressed as:

$$w_{k+1}^{liq} = f_{NN}(w_k^{liq}, ..., w_{k-N_o-1}^{liq}, w_k^{gas}, ..., w_{k-N_i-1}^{gas}, \chi_k, ..., \chi_{k-N_i-1}, \Omega_k, ..., \Omega_{k-N_i-1}) \tag{10}$$

The type of network selected is feedforward neural network. Although recurrent neural network is usually used in time series prediction, feed-forward neural network could also be applied to this type of problem. Moreover, it is more straightforward to convert a feedforward network to a mathematical function for the use of RTO. The networks have been created using 2 different frameworks: Keras (for models with Adam optimization algorithm) and MATLAB (for models with Levenberg-Marquardt algorithm).

### 3.3.2 Training data

The training data is represented in Figure 8. In the training data, the gas flow rate varies from 1 to 4 sL/min, the valve opening varies from 25% to 60%, and the pump speed varies from 35% to 46.5%. The sampling time is 1 second.

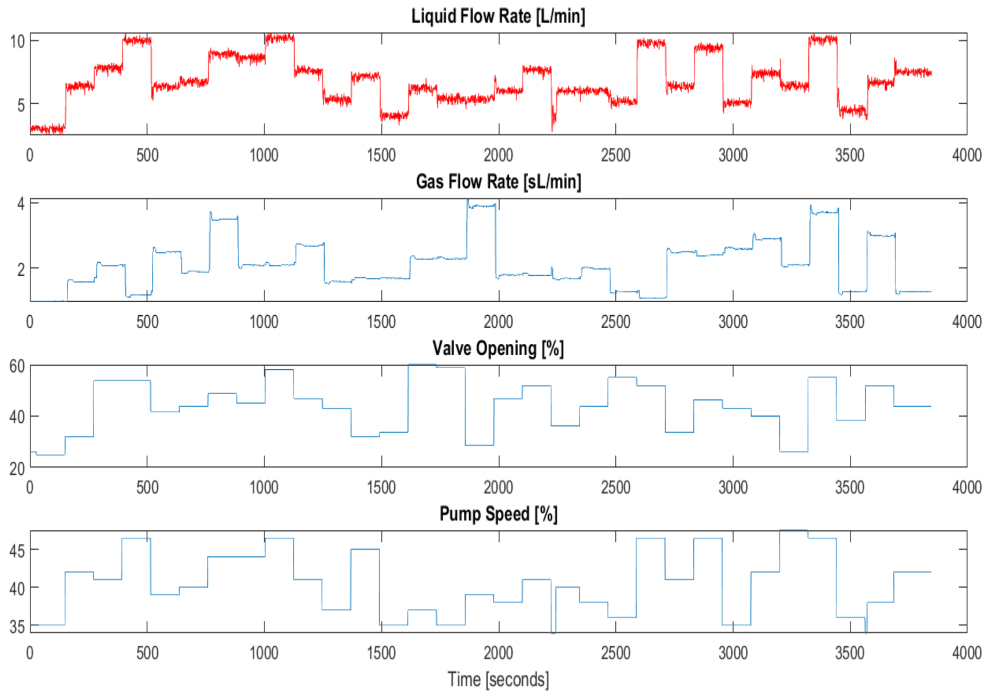

**Figure 8:** *Training Data*

The training data can not be used directly. It must be preprocessed to generate the outputs $w_{k+1}^{liq}$ and the inputs $(w_k^{liq}, ..., w_{k-N_o-1}^{liq}, w_k^{gas}, ..., w_{k-N_i-1}^{gas}, \chi_k, ..., \chi_{k-N_i-1}, \Omega_k, ...$ $, \Omega_{k-N_i-1})$ that the neural networks use for training. In this project, $N_o$ is equal to 1 and $N_i$ is equal to 2. How different values of $N_o$ and $N_i$ affect the model performance is discussed in subsubsection 3.4.3. The code used to preprocess the training data can be found in the Appendix.

### 3.3.3 Testing methods and Testing data

An important problem is how to test the performance of a model. In this project, two criteria have been used to evaluate a model. First, we know that the process in the lab rig is stable. A good dynamic model for stable systems should converge to a fixed value of liquid flow rate if the gas flow rate and disturbances remain unchanged during the prediction interval. Therefore, we need to check if a model can capture this stable property of the actual system. This convergence property of a model $f_{NN}$ is checked by the first test. The procedure of the first test is described below:

- Step 1: Set the input to be $(w_0^{liq}, w_{-1}^{liq}, ..., w_0^{gas}, w_0^{gas}, ..., \chi_0, \chi_0, ..., \Omega_0, \Omega_0, ...)$

- Step 2: Predict the liquid flow rate at the next time step by

$$\hat{w}_1^{liq} = f_{NN}(w_0^{liq}, w_{-1}^{liq}, ..., w_0^{gas}, w_0^{gas}, ..., \chi_0, \chi_0, ..., \Omega_0, \Omega_0, ...). \tag{11}$$

Record $\hat{w}_1^{liq}$

- Step 3: Add the most recent prediction to the input as the first input element and remove the least recent element of liquid flow rate such that the number of liquid flow rate elements remains unchanged. The input becomes

$$(\hat{w}_1^{liq}, w_0^{liq}, ..., w_0^{gas}, w_0^{gas}, ..., \chi_0, \chi_0, ..., \Omega_0, \Omega_0, ...) \tag{12}$$

- Step 4: Predict the liquid flow rate at the next time step by

$$\hat{w}_2^{liq} = f_{NN}(\hat{w}_1^{liq}, w_0^{liq}, ..., w_0^{gas}, w_0^{gas}, ..., \chi_0, \chi_0, ..., \Omega_0, \Omega_0, ...) \tag{13}$$

. Record $\hat{w}_2^{liquid}$

- Step 5: Repeat Steps 3-4. Observe if the prediction values converge over time.

If the model predictions converged, its performance is further evaluated by comparing its steady-state predictions and the actual flow rates under different conditions of gas flow rate and disturbances. To conduct the test, two sets of steady-state liquid flow rate data, Interpolation and Extrapolation, have been collected. In Interpolation dataset, values of the gas flow rate and the disturbances were selected from their corresponding ranges in the training data. In Extrapolation, values of the gas flow rate and the disturbances were beyond their corresponding ranges in the training data. There are 21 combinations of gas flow rate and disturbance values in Interpolation, and 14 combinations in Extrapolation. The two sets are represented in Figure 9 and Figure 10, respectively.

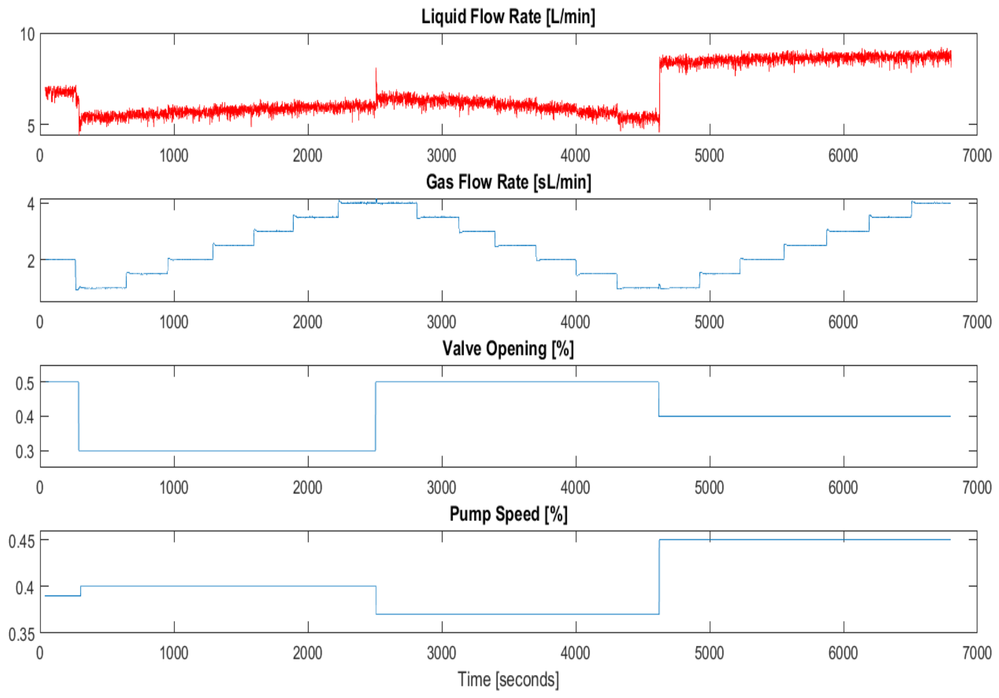

**Figure 9:** *Interpolation Testing Data*

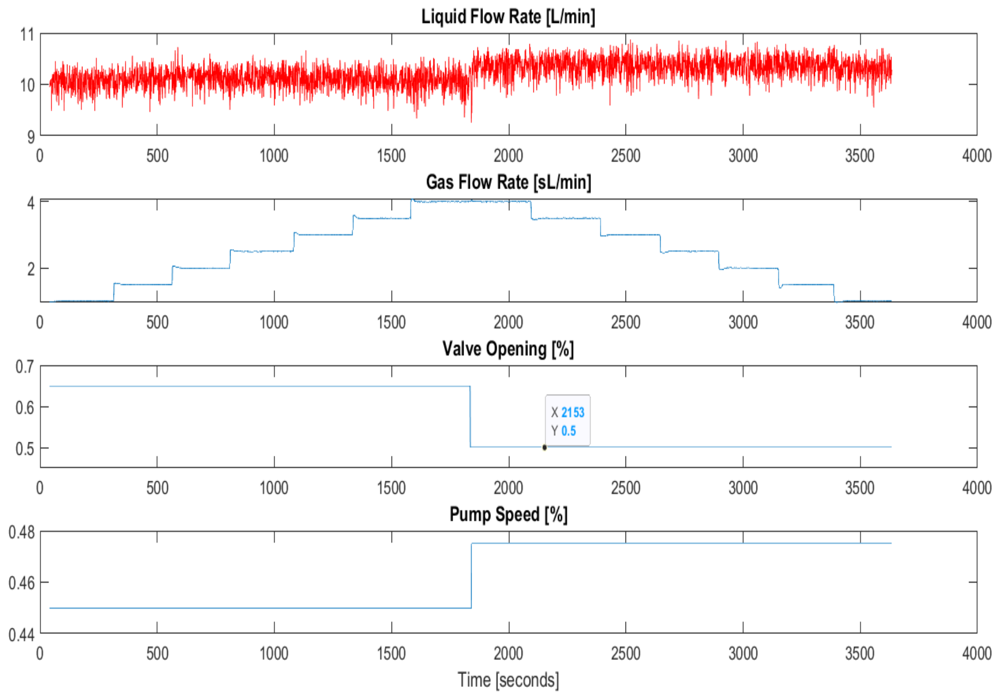

**Figure 10:** *Extrapolation Testing Data*

For each combination, the liquid flow rate is recorded in 5 minutes, and the average flow rate is taken as the actual liquid flow rate under this condition. These actual

flow rates are plotted in the same graph with the corresponding predicted flow rates to compare, and the mean square error (MSE) between them are also calculated. The MATLAB code for Test 1 and Test 2 can be found in the Appendix.
The common method to evaluate the performance of a NN model is to use a testing set. We give the model inputs from the testing set, let it predict the corresponding outputs, and compared these predictions to the actual outputs. In our case, the testing set will contain dynamic data. We have observed models that have good performance on testing set (or low testing error) but do not satisfy the convergence property. Therefore, this method is not used for evaluation purpose in this project.

## 3.4 Results and Discussion

As discussed earlier in paragraph 3.2.1.5, we have to tune many hyperparameters in order to find an appropriate model. There is currently no systematic procedure to carry out tuning. In this project, major hyperparameters have been selected to investigate their effects on the model performance, such as: activation function, number of hidden layers and number of neurons. Although many factors have been studied, two of them seem to have the most significant impact on the model performance: the type of activation function and the optimization algorithms. This section discusses the effects of these two factors on the model performance, as well as how to use the obtained model in steady-state RTO.

### 3.4.1 Effect of activation functions

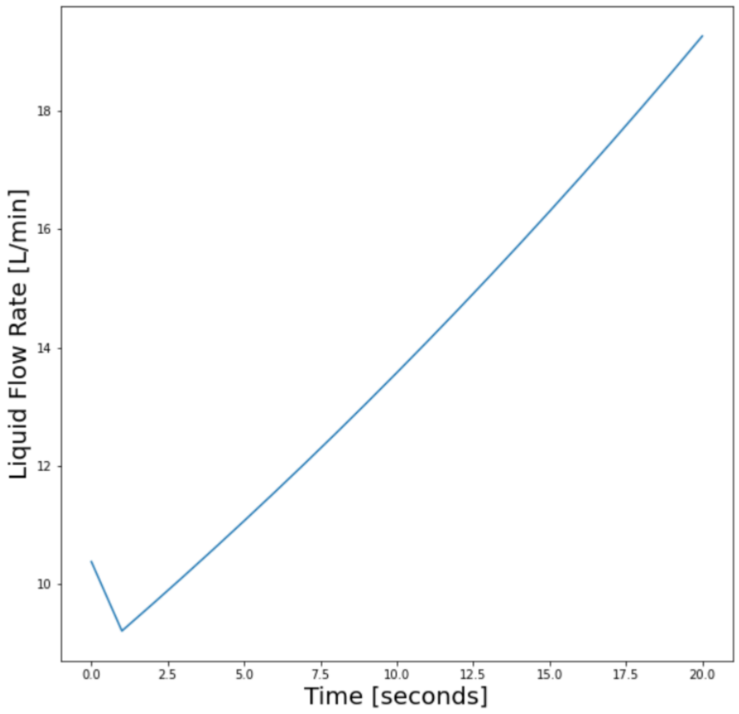

**Figure 11:** *Test 1 result of a model with ReLU activation function*

Due to various benefits it offers, the ReLU activation function is usually used in feedforward networks [26]. It was also the first type of activation function used in this project. However, models with ReLU activation function could not converge to steady-state values in Test 1. Instead, the predicted liquid flow rate always explodes over time, as represented in Figure 11. To make sure that the explosion resulted from the use of ReLU, we repeated Test 1 while keeping the activation function unchanged and varying the number of hidden layers and the number of neurons in each layer. The number of hidden layers was either 1 or 2, the number of neurons in the first hidden layer varied from 8 to 64, and the number of neurons in the second hidden layer (if it exists) varied from 4 to 32. The network was initialized using Xavier initializer and regulated by L2 regularization method. The inputs were normalized. The optimization algorithm used was Adam. We observed the output explosion in all cases. But when the activation function was changed to hyperbolic tangent, the models converged.

The reason for output explosion was figured out. Figure 12 represents the procedure of Test 1. As it could be seen from Figure 7 and Figure 12, during Test 1, the feedforward network behaves exactly as a RNN. ReLU is seldom used in RNN because it causes the output to explode, and how to use ReLU safely in RNNs is an ongoing research topic [27] [28] [29]. Therefore, the output explosion as we observed in Test 1 could be expected.

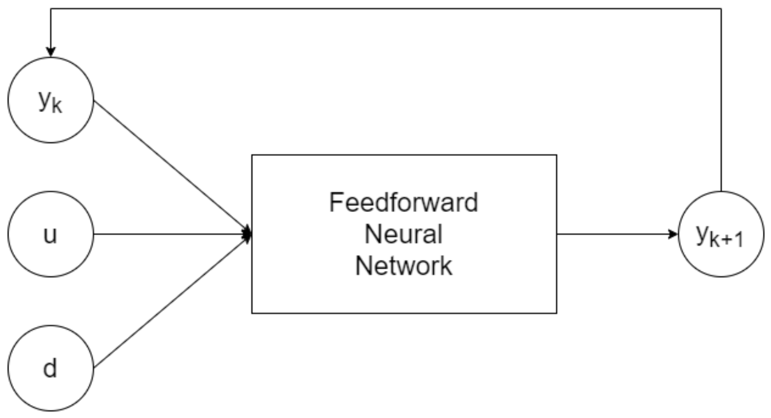

**Figure 12:** *Test 1 procedure diagram*

### 3.4.2 Effect of optimization algorithms

After models with the hyperbolic tangent activation function were discovered to have steady-state convergence property, we investigated the effects of the number of hidden layers and the number of neurons in these layers on the model performance. The models created in subsubsection 3.4.1 were trained and evaluated again, now with tanh as the activation function instead of ReLU. Unfortunately, we could not find a model with adequate performance. In the best model obtained, the difference between the predictions and the correct values could be as large as 8 L/min, as represented in Figure 13. The largest deviations were in the interpolation predictions,

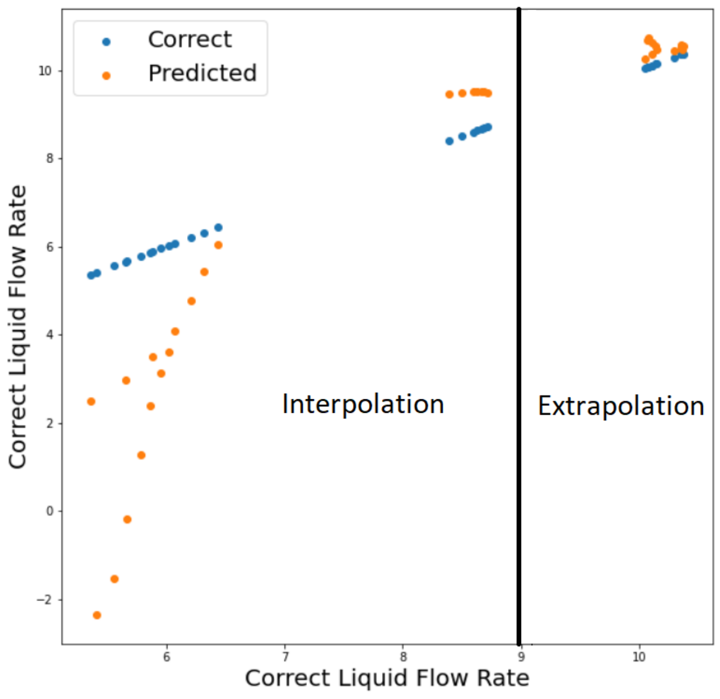

**Figure 13:** *Results with the Adam optimization algorithm*

Another optimization algorithm, the Levenberg-Marquardt algorithm (LM), has been used instead of the Adam optimizer. The new algorithm enabled the network to predict more accurately in the interpolation conditions. Figure 14 represented the results of the best model obtained, which consists of 1 hidden layer and 4 neurons in this layer. The predictions in the second cluster, where the correct outputs are about 8-9 L/min, was quite inaccurate. This was probably due to the training data, which had more data points in the output range of 5-8 L/min and less data points in the range of 8-9 L/min.

Although the extrapolation predictions deviated quite significantly from the correct values, the interpolation deviations were modest. As poor extrapolation predictions were expected for neural network models [30], the LM algorithm seems to be better than the Adam algorithm. This result aligns with the observations in [31].

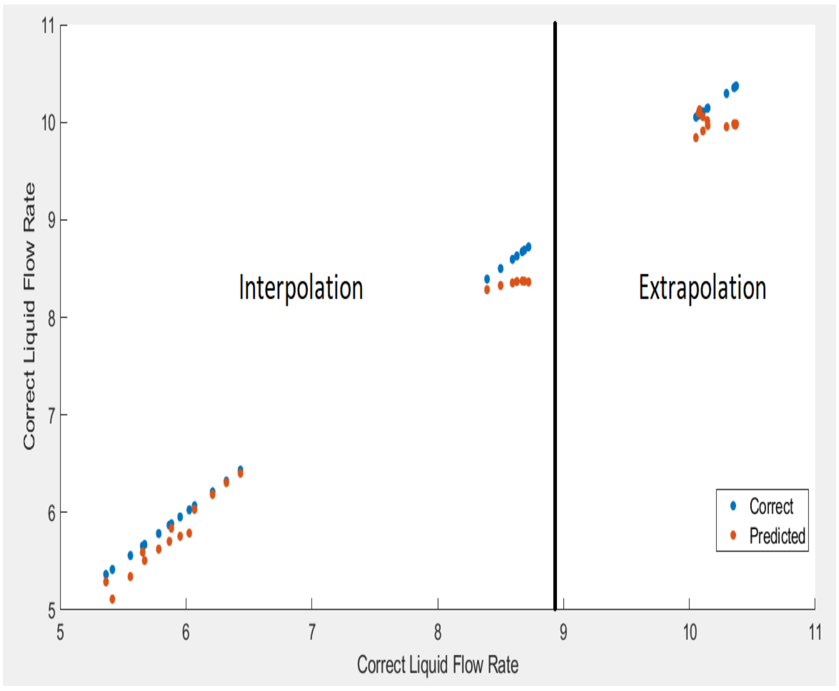

**Figure 14:** *Results with the LM algorithm (1 hidden layer, 4 neurons)*

### 3.4.3 Values of $N_i$ and $N_o$

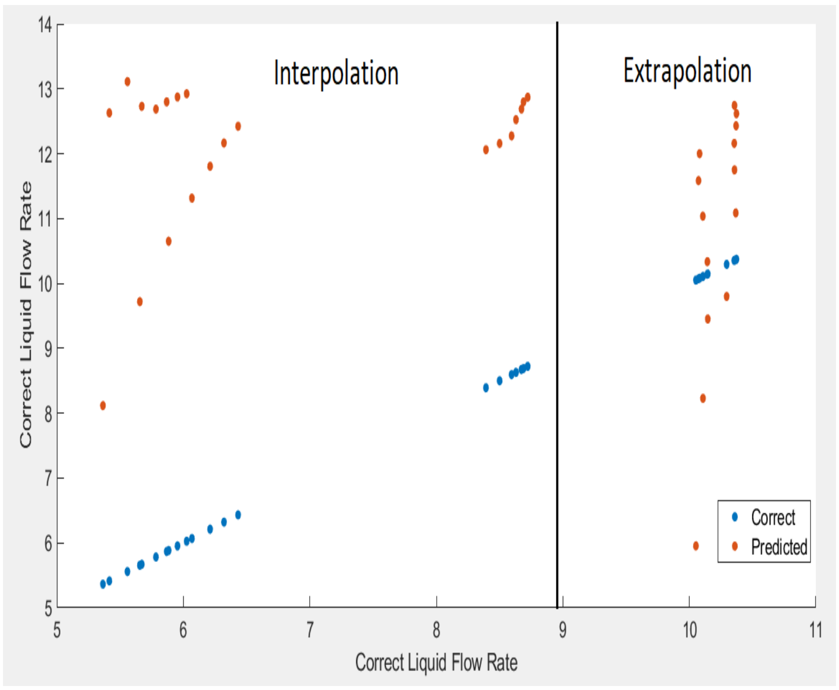

**Figure 15:** *Results with $N_i = 2$ and $N_o = 2$ (1 hidden layer, 4 neurons, LM algorithm)*

All the models discussed have the values of $N_i$ and $N_o$ being 2 and 1, respectively. These values were selected taking into account that the time constant of the system was below 2 seconds, and the time step of the models was equal to the time step of the training data - 1 second. In experiments we conducted, setting both $N_i$ and $N_o$ to 2 seems to worsen the performance, as represented in Figure 15. Due to time constraints, we did not investigate whether increasing $N_i$ and $N_o$ further could increase the model performance or not.

### 3.4.4 How to use the dynamic model in steady-state RTO

In steady-state RTO, we must have a steady-state model of the system, as shown in 14.

$$\begin{aligned} \min_{w_i^{gas}} \quad & J = -\alpha_{liq} \cdot \Sigma w_i^{liq} \\ \text{subject to:} \quad & w_i^{liq} = f_i(w_i^{gas}, d_i) \\ & \Sigma w_i^{gas} \leq W_{max}^{gas} \end{aligned} \tag{14}$$

However, the model we obtained here is a dynamic one. To use it in steady-state RTO, we have to modify the optimization problem formulation into the one represented in 15.

$$\begin{aligned} \min_{w_i^{gas}} \quad & J = -\alpha_{liq} \cdot \Sigma w_i^{liq}(10) \\ \text{subject to:} \quad & w_i^{liq}(10) = f_i(w_i^{liq}(9), w_i^{gas}, d_i) \\ & w_i^{liq}(9) = f_i(w_i^{liq}(8), w_i^{gas}, d_i) \\ & \dots \\ & w_i^{liq}(1) = f_i(w_i^{liq}(0), w_i^{gas}, d_i) \\ & \Sigma w_i^{gas} \leq W_{max}^{gas} \end{aligned} \tag{15}$$

In the new formulation, we wish to optimize the total liquid flow rate at time step 10. As the dynamic of the system is fast, we can assume that at time step 10 the liquid flow rates already reach their steady-state values. Therefore, maximizing the total liquid flow rate at 10 seconds is equivalent to maximizing the steady-state total liquid flow rate. We also replaced the steady-state model equation by the dynamic model equations. Although there are 10 dynamic model equations, we only use 1 decision variable $w_i^{gas}$ in all equations. This helps to keep the input constant throughout the control horizon and make sure that the system can be at steady state eventually.

We did not consider the use of dynamic RTO for two reasons. First, our system has fast dynamics. Hence, it is mostly in steady-state operation. Therefore, a dynamic

RTO might not be necessary. Furthermore, it requires more computational power. Also, we only evaluated the steady-state predictions of the models in this work. Therefore, the models may give inaccurate dynamic predictions, which makes them inappropriate for the use in dynamic RTO.

# 4 Conclusion

## 4.1 Applications of Reinforcement Learning in Process Control

The applicability of Reinforcement Learning in Chemical Engineering problems has been discussed. The problems considered are: steady-state RTO, regulatory control, erosion control, and plantwide control design. Application of RL in the problem of steady-state RTO is inappropriate. The use of RL in regulatory control requires that the challenge of disturbance rejection has to be resolved. The erosion control problem has several common properties with the traditional RL problems, such as the chess game, therefore, utilizing RL here might be highly beneficial.
Besides low-level decision problems such as the three problems mentioned, the application of RL in high-level decision problems, such as plantwide control design, should be investigated. In plantwide control design, the tasks are identifying the variables to control, identifying the variables to manipulate, and pairing variables in the two variable sets to achieve stable operation and self-optimizing control of the plant. Since there are possibly many feasible control structures and it is hard to fully evaluate their performance in advance of pairing completion, RL could help us to identify good structures in a case-by-case basis. Through these cases, we might also derive new design rules.

## 4.2 Applications of Neural Network in Economics Optimization of the Gas-Lift lab rig

Feedforward neural network has been applied to create a model of the Gas-Lift lab rig for the use in steady-state Real-Time Optimization. Since we did not have a sufficient amount of steady-state training data, a dynamic model has been created instead and the steady-state optimization problem has been reformulated such that the dynamic model could be used. It has been shown that the networks should not have rectified linear unit (ReLU) as their activation function, otherwise the output predictions would explode over time. The importance of optimization algorithms has also been discussed. The use of Levenberg-Marquardt algorithm results in models with higher accuracy than those from the Adam optimizer.

There are two limitations in this project: the model accuracy evaluation and the inputs of the model. Although the obtained model could give predictions that are close to the correct liquid flow rates, it should be evaluated at more operating conditions so we can have a better idea about the model performance. Afterward, efforts can be spent on making the model more accurate if necessary. Another limitation originates from the inputs used to create the model. The currently used inputs are: the liquid flow rate, the gas flow rate, and the disturbances (valve opening and pump rotation speed). In practice, information about the disturbances are rarely available. Therefore, the possibility of using gas flow rate and another available measurement - pressure at the top of the wells - as the model inputs should be studied.

## 5 References

[1] "Mastering the game of Go without human knowledge". In: *Nature* 550 (), pp. 354–359.

[2] "Real-time optimization using reinforcement learning". In: *Computers Chemical Engineering* 143 (2020).

[3] "Data-Driven Economic NMPC Using Reinforcement Learning". In: *IEEE Transactions on Automatic Control* 65 (2020), pp. 636–648. DOI: 10.1109/TAC.2019.2913768.

[4] P. Loewen S. Spielberg R. Gopaluni. "Deep reinforcement learning approaches for process control". In: *2017 6th International Symposium on Advanced Control of Industrial Processes* (2017), pp. 201–206. DOI: 10.1109/ADCONIP.2017.7983780.

[5] Y. Bengio I. Goodfellow and A. Courville. In: Deep Learning (2016). URL: http://www.deeplearningbook.org.

[6] "MPC: Current practice and challenges". In: *Control Engineering Practice* 20 (), pp. 328–342.

[7] K. Kim Chul-Jin Lee Y. Lee M. Lee J. Lee J. Na. "NARX modeling for real-time optimization of air and gas compression systems in chemical processes". In: *Computers and Chemical Engineering* 115 (2018), pp. 262–274. DOI: https://doi.org/10.1016/j.compchemeng.2018.04.026.

[8] D. Christofides Z. Zhang Z. Wu. "Real-Time Optimization and Control of Nonlinear Processes Using Machine Learning". In: *Mathematics* 7 (2019). DOI: https://doi.org/10.3390/math7100890.

[9] S. Skogestad D. Krishnamoorthy B. Foss. "Real-Time Optimization under Uncertainty Applied to a Gas Lifted Well Network". In: *MDPI* (2016). DOI: https://doi.org/10.3390/pr4040052.

[10] R.Sutton and A. Barto. In: Reinforcement Learning: An Introduction, 2nd edition (2018).

[11] D. Silver H. Hasselt A. Guez. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv:1509.06461* (2015).

[12] M. Mirza A. Graves T. Lillicrap T. Harley D. Silver K. Kavukcuoglu V. Mnih A. Badia. “Asynchronous Methods for Deep Reinforcement Learning”. In: *arXiv:1602.01783* (2016).

[13] J. Wang Z. Kurth-Nelson C. Blundell D. Hassabis M. Botvinick S. Ritter. “Reinforcement Learning, Fast and Slow”. In: *Trends in Cognitive Sciences* 23 (5 2019), pp. 408–422. DOI: `https://doi.org/10.1016/j.tics.2019.02.006`.

[14] A. Balau and C. Lazar. “One Step Ahead MPC for an Automotive Control Application”. In: *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems* (2011), pp. 61–70. DOI: `10.1109/ECBS-EERC.2011.18`.

[15] S. Skogestad and M. Morari. “Effect of Disturbance Directions on Closed-Loop Performance”. In: *Industrial Engineering Chemistry Research* 26 (1987), pp. 2029–2035.

[16] “Plantwide control: The search for the self-optimizing control structure”. In: *Journal of Process Control* 10 (5), pp. 487–507.

[17] A. Verheyleweghen and J. Jaschke. “Oil production optimization of several wells subject to choke degradation”. In: *IFAC-Papers Online* 51 (8 2018), pp. 1–6. DOI: `https://doi.org/10.1016/j.ifacol.2018.06.346`.

[18] P. Douglas M. Naysmith. “Review of Real Time Optimization in the Chemical Process Industries”. In: *Asia-Pacific Journal of Chemical Engineering* 3 (2 1995), pp. 67–87. DOI: `https://doi.org/10.1002/apj.5500030202`.

[19] X. Chen Z. Xu-Z. Shao W. Chen L. Zhu. “Sensitivity-based Mnemonic Enhancement Optimization (S-MEO) for Real-time Optimization of Chemical Process”. In: *Computer Aided Chemical Engineering* 32 (2013), pp. 853–858. DOI: `https://doi.org/10.1016/B978-0-444-63234-0.50143-3`.

[20] “Real time optimization (RTO) with model predictive control (MPC)”. In: *Computers and Chemical Engineering* 34 (12).

[21] J. Renfro C. Pantelides. “The online use of first-principles models in process operations: Review, current status and future needs”. In: *Computers and Chemical Engineering* 51 (2013), pp. 136–148. DOI: `https://doi.org/10.1016/j.compchemeng.2012.07.008`.

[22] J.C de Jesus J.C. Pinto A.D. Quelhas N. “Common vulnerabilities of RTO implementations in real chemical processes”. In: *The Canadian Journal of Chemical Engineering* 91 (4 2013), pp. 652–668. DOI: `https://doi.org/10.1002/cjce.21738`.

[23] V. Venkatasubramanian. “The promise of artificial intelligence in chemical engineering: Is it here, finally?” In: *AIChE Journal* 65 (3 2019), pp. 466–478. DOI: `https://doi.org/10.1002/aic.16489`.

[24] X. Glorot and Y. Bengio. “Understanding the difficulty of training deep feed-forward neural networks”. In: *Journal of Machine Learning Research* 9 (2010), pp. 249–256.

[25] K.R. Muller G. Montavon G.B. Orr. In: Neural Networks: Tricks of the Trade (2012).

[26] V. Nair and G. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the International Conference on Machine Learning (ICML)* (2010), pp. 807–814.

[27] G. Hinton Q. Le N. Jaitly. “A simple way to initialize recurrent networks of rectified linear units”. In: *arXiv Preprint* (2015). DOI: `arXiv:1504.00941`.

[28] R. Salakhutdinov N. Srebro B. Neyshabur Y. Wu. “Path-normalized optimization of Recurrent Neural Networks with ReLU activations”. In: *arXiv Preprint* (2016). DOI: `arXiv:1605.07154`.

[29] R. Vorontsov S. Kahou-Y. Bengio S. Chandar C. Sankar. “Towards non-saturating recurrent units for modelling long-term dependencies”. In: *arXiv Preprint* (2019). DOI: `arXiv:1902.06704`.

[30] “Extrapolation and interpolation in neural network classifiers”. In: *IEEE Control Systems Magazine* 12 (5), pp. 50–53.

[31] M. Kiran B.M. Ozyildirim. “Do optimization methods in deep learning applications matter?” In: *arXiv Preprint* (2020). DOI: `arXiv:2002.12642`.

# A Reinforcement Learning - MATLAB code

## A.1 Brief explanation of the files

The file “SARSAmoduleDouble.m” is the main simulation file. It contains information about the process that we wish to control. It also has the parameters and the update rule of the RL agent. The RL update algorithm used here is the SARSA algorithm.
The function “Q_builder” creates a table to store the initial and updated action-value functions.
The function “nextstate” calculates the next output of the system based on the current process states and the action just implemented by the agent.
The function “greedy_selector” accesses to all action-value functions corresponding to the current state and selects an action to implement according to the $\epsilon$-greedy policy. The $\epsilon$-greedy policy is already discussed in subsection 2.1.
The function “GPS” identifies the location of a state-action pair in the table created by the function “Q_builder”.

## A.2 Main file

```
clear
clc

% Process range
range_action = 1:0.1:4;  % Range of inputs
range_state = 7.07:0.01:9.26;  % Range of outputs

ysP = 7.5; usP = 1.5;  % Process nominal points
% Plant Model
Kp = 0.703; Tp = 1.62;
G1 = tf([Kp], [Tp 1]); G1d = c2d(G1,1,'zoh');
[Ap,Bp,Cp,Dp] = ssdata(G1d);

%% RL agent
Q = Q_builder(range_action,range_state);  % Initialize the Q-values
Q_stealth = Q_builder(range_action, range_state);
```

```
% As the initial Q-values are all 0, no need to specify for Q(terminal,:)
% For controller 2, load a saved Q and use directly. Do not use Q_builder

alpha = 0.05; gamma = 1;   % Parameters of the agent

yset_total = [7.5, 8.2];
yset = yset_total(1);  % Goal of the agent
%% Simulation
% Parametes
n_loop = 500;
% For controller 2, n_loop = 1
index_loop = 1;

while index_loop < (n_loop + 1)
    if index_loop < round(0.9*n_loop)
        epsilon = 0.5;
    else epsilon = 0.005;
    end
    % Pick random initial inputs-outputs
    %action_array = range_action;
    %action_array = action_array(randperm(length(action_array)));
    %action_ini = action_array(1);  % Random initial inputs
    action_ini = 4;
    x_ini = Bp*(action_ini-usP)/(1-Ap);  % Initial state
    y_ini = round(ysP + Cp*Bp*(action_ini-usP)/(1-Ap),2);

    % Initialization
    termination_now = 0; T_terminate = 50;
    epi_step = 1;
    output = [y_ini]; input = [];
    x_current = x_ini; y_current = y_ini;

    while termination_now < 1
        a = greedy_selector(Q,y_current,epsilon);
        %% Environment
        [x_next,y_next] = nextstate(x_current,a);

        % Reward function
        ReW = 10/(abs(y_next-yset)+0.005);
        ReW_stealth = 10/(abs(y_next-yset_total(2))+0.005);

        % Termination check
        if epi_step >= T_terminate %abs(y_next-yset) <= 10^(-4) &
            termination_now = 1;
        else termination_now = 0;
        end

        % For plotting
        input = [input; a];
        output = [output; y_current];
        %% Return to SARSA
        a1 = greedy_selector(Q,y_next,epsilon);  % SARSA
        position_current = GPS(Q,y_current,a);  % Look up in the table
        position_new = GPS(Q,y_next,a1);  % Look up in the table
        % SARSA Update Rule
        Q(position_current,1)=Q(position_current,1)
        +alpha*(ReW+gamma*Q(position_new,1)-Q(position_current,1));
        Q_stealth(position_current,1)=Q_stealth(position_current,1)
        +alpha*(ReW_stealth+gamma*Q_stealth(position_new,1)
        -Q_stealth(position_current,1));
        %% Return to the Environment
        x_current = x_next;
        y_current = y_next;
        epi_step = epi_step + 1;
    end
    index_loop = index_loop + 1;
end
```

```
t = 1:1:epi_step;
plot(t',output)
hold on
stairs(input)

Q_restruct = zeros(length(range_state),length(range_action));
for n = 1:size(Q,1)
    row_index = round((Q(n,2)-min(range_state))/0.01)+1;
    col_index = round((Q(n,3)-min(range_action))/0.1)+1;
    Q_restruct(row_index,col_index) = Q(n,1);
end

Q_restruct_stealth = zeros(length(range_state),length(range_action));
for n = 1:size(Q,1)
    row_index = round((Q(n,2)-min(range_state))/0.01)+1;
    col_index = round((Q(n,3)-min(range_action))/0.1)+1;
    Q_restruct_stealth(row_index,col_index) = Q_stealth(n,1);
end

figure
surf(Q_restruct)
figure
surf(Q_restruct_stealth)
```

## A.3 Accessory files

### A.3.1 Function "Q_builder"

```
function [Q] = Q_builder(range_action,range_state)
% The form of Q matrix [S1-A1 ... S10-A1;...; S1-A2 ... S10-A2;...]
NoA = length(range_action); NoS = length(range_state);

q = 0.5*ones(NoS*NoA,1);

S = zeros(NoS*NoA,1);  % Matrix of state-value
```

```
for i = 1:NoS
    for j = 0:(NoA-1)
        S(i+j*NoS,1) = range_state(i);
    end
end

A = zeros(NoS*NoA,1);  % Matrix of action-value
for m = 1:NoA
    for n = 0:(NoS-1)
        A(m+n*NoA,1) = range_action(m);
    end
end

Q = [q,S,A];

end
```

### A.3.2 Function "nextstate"

```
function [x_next,y_new] = nextstate(x_current,a)
%UNTITLED Summary of this function goes here
% Parameters of the environment model
Kp = 0.703; Tp = 1.62;
G1 = tf([Kp], [Tp 1]); G1d = c2d(G1,1,'zoh');
[Ap,Bp,Cp,Dp] = ssdata(G1d);  % Convert tf to state-space
ysP = 7.5; usP = 1.5;  % The standard point

% Noise
ar = -0.25*0; br = 0.25*0;  % Noise magnitude

% Next State
x_next = Ap*x_current + Bp*(a-usP);
y_new = ysP + Cp*x_next + (br-ar)*rand + ar;
y_new = round(y_new,2);
    if y_new > 9.26
```

```
          y_new = 9.26;
      elseif y_new < 7.07
          y_new = 7.07;
      else y_new = y_new;
      end
end
```

### A.3.3 Function "greedy_selector"

```
function [action] = greedy_selector(Q,y,epsilon)

y = round(y,2);  % Round—up for safety reason

% Pick possible actions of the state out of Q—matrix
% Q—matrix = [Q—values, States, Actions] to
% positions = [Q—values, Actions]

positions = [];
for i = 1:size(Q,1)
if abs(Q(i,2)-y) <= 10^(-4)
     positions = [positions; Q(i,1), Q(i,3)];
else positions = positions;
end
end

max_value = max(positions(:,1));  % Max Q—values
n_max = sum(positions(:,1) == max_value);  % How many max actions
% Probability of choosing max action
probab_others = round(epsilon/size(positions,1)*100);
% Probability of choosing others
probab_max = round(((1 - epsilon)/n_max + epsilon/size(positions,1))*100)

prob_matrix = [];
for j = 1:size(positions,1)
     if positions(j,1) ~= max_value
```

```
        prob_matrix = [prob_matrix, positions(j,2)
        *ones(1,probab_others)];
    else prob_matrix = [prob_matrix, positions(j,2)
        *ones(1,probab_max)];
    end
end

prob_matrix = prob_matrix(randperm(length(prob_matrix)));
% Pick action according to the given probabilities
action = prob_matrix(1);

end
```

### A.3.4 Function "GPS"

```
function index_row = GPS(Q,y,a)

y = round(y,2);  % Round-up for safety reason

for i = 1:size(Q,1)
if abs(Q(i,2)-y) <= 10^(-4) & Q(i,3) == a
    index_row = i;
end
end

end
```

## B Neural Network - Training data pre-processing

```
load trainingdata
time = trainingdata(:,1);

% Output
  % Liquid Flowrate
wliquid1 = trainingdata(:,9);
```

```
wliquid2 = trainingdata(:,11);
wliquid3 = trainingdata(:,13);
  % Top Pressure
ptop1 = trainingdata(:,18); % bar A
ptop2 = trainingdata(:,20);
ptop3 = trainingdata(:,22);

% Input: Well 1 - Well 2 - Well 3
  % Gas Flowrate
wgas1 = trainingdata(:,3);
wgas2 = trainingdata(:,5);
wgas3 = trainingdata(:,7);
  % Pump Rotation
pumppower = trainingdata(:,29);
pumpspeed = 100*(5000*pumppower-8*ones(size(time,1),1))./100;
  % Valve Opening
valvecurrent1 = trainingdata(:,23);
valve1 = 62.5*valvecurrent1-0.25*ones(size(time,1),1);
valvecurrent2 = trainingdata(:,24);
valve2 = 100*(62.5*valvecurrent2-0.25*ones(size(time,1),1));
valvecurrent3 = trainingdata(:,25);
valve3 = 62.5*valvecurrent3-0.25*ones(size(time,1),1);

figure
subplot(4,1,1)
plot(time, wliquid2,'r')
title('Liquid Flow Rate [L/min]')
subplot(4,1,2)
plot(time, wgas2)
title('Gas Flow Rate [sL/min]')
subplot(4,1,3)
plot(time, valve2)
title('Valve Opening [%]')
subplot(4,1,4)
```

```
plot(time, pumpspeed)
title('Pump Speed [%]')
xlabel('Time [seconds]')

% Preprocess trainingdata into the form [y_current2 wgas2_current valve2_
% pumpspeed2_current y_next2]
lag_input = 1;
lag_output = 1;

D = [wliquid2 wgas2 valve2 pumpspeed];
fin = size(D,1);
D = D((1+lag_input):(end-lag_output),:);

wliq2_next = wliquid2((2+lag_input):(end-lag_output+1));  % y(k+1)

if lag_input > 1
    D_inputprev = [];
    for j = 2:lag_input
        lim_up = 1+lag_input-(j-1);
        lim_lo = fin-lag_output-(j-1);
        wgas2_prev = wgas2(lim_up:lim_lo);
        valve2_prev = valve2(lim_up:lim_lo);
        pumpspeed_prev = pumpspeed(lim_up:lim_lo);
        D_inputprev = [D_inputprev wgas2_prev valve2_prev pumpspeed_prev]
    end
else D_inputprev = [];
end

if lag_output > 1
    D_outputprev = [];
    for j = 2:lag_output
        wliq2_prev = wliquid2((1+lag_input-(j-1)):(end-lag_output-(j-1))
        D_outputprev = [D_outputprev wliq2_prev];
    end
```

```
 else D_outputprev = [];
end
D = [D D_outputprev D_inputprev wliq2_next];

% Write into the CSV file
csvwrite('LagRigtrainingdataVardelay2.txt',D)
```

## C Neural Network - Test 1 and Test 2 MATLAB code

The trained neural networks should be saved as a function. Here in this code, the feedforward network that we would like to test is the function “ffwNN”

```
% Steady-State Testing Module
gas = [1 1.5 2 2.5 3 3.5 4];
valve = [0.3 0.5 0.4 0.65 0.5];
pump = [0.4 0.37 0.45 0.45 0.475];

Y_true = [5.4152, 5.5586, 5.6715, 5.7836, 5.8685, 5.9538, 6.0264,
    5.3646, 5.6562, 5.8841, 6.0681, 6.2119, 6.3212, 6.4330, 8.3923,
    8.5, 8.5945, 8.6294, 8.6728, 8.6887, 8.7221, 10.0519, 10.1075,
    10.1455, 10.1423, 10.1062, 10.0722, 10.0805, 10.2945, 10.3668,
    10.3566, 10.3545, 10.3688, 10.3719, 10.3565];

inp = [];
for i = 1:size(valve,2)
    valve_com = valve(i)*ones(1,size(gas,2));
    pump_com = pump(i)*ones(1,size(gas,2));
    inp_new = [gas; valve_com; pump_com];
    inp = [inp inp_new];
end

Y_pred = [];
for j = 1:size(inp,2)
    ini = zeros(1,7);
```

```
        ini(1,1) = 5.5;
        ini(1,2) = inp(1,j);
        ini(1,3) = inp(2,j);
        ini(1,4) = inp(3,j);
        ini(1,5) = inp(1,j);
        ini(1,6) = inp(2,j);
        ini(1,7) = inp(3,j);

        Y_plot = [];
        for i = 1:20
        % Call the trained network here
            Y_new = ffwNN(ini);
            Y_plot = [Y_plot Y_new];
            ini(1,1) = Y_new;
        end

        Y_pred = [Y_pred Y_new];

        figure
        plot(Y_plot)
    end

mse = (1/size(Y_true,2))*sqrt(sum((Y_pred-Y_true).^2))

figure
scatter(Y_true, Y_true,'filled')
hold on
scatter(Y_true, Y_pred,'filled')
xlabel('Correct Liquid Flow Rate')
ylabel('Correct Liquid Flow Rate')
```