Vinh Phuc Bui Nguyen

Application of Machine Learning in Economic Optimization

Master's thesis in Chemical Engineering
Supervisor: Johannes Jaeschke - Sigurd Skogestad
Co-supervisor: Jose Otavio Assumupcao Matias
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Vinh Phuc Bui Nguyen

Application of Machine Learning in Economic Optimization

Master's thesis in Chemical Engineering
Supervisor: Johannes Jaeschke - Sigurd Skogestad
Co-supervisor: Jose Otavio Assumupcao Matias
June 2021

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Contents

List of Figures

Acknowledgement

It has always been my dream being able to create something scientifically new. I guess I have achieved that dream in this Master's thesis. It has been a long and challenging journey, partly due to the pandemic. Fortunately, I have received all the support I needed on the way.

I would like to thank my supervisors, Professor Johannes and Professor Sigurd, for accepting to supervise my work and providing your great insights on the topic of Self-Optimizing Control. Our discussions have helped to shape my thinking.

I wish to express my sincere gratitude towards my co-supervisor, Postdoc. Jose Otavio Assumpcao Matias. All the support I have received from you would make an infinitely long list. So, I will just say it briefly: Thank you for guiding me through this tough journey, from the very beginning.

I also wish to thank my girlfriend, Dieu Thao, for always being here. I know it was not easy. I owe my friends back home, Quoc Thai and Phuong Nguyen, for the entertaining conversations. I also owe my friends here in Norway, Hoang Dam Khanh and Mikhail Pedrovich, for their daily practical support.

Abstract

In this thesis, we introduced two new applications of Machine Learning in the field of Economic Optimization. The first application addresses the problem of searching for global Self-Optimizing variables. We applied Genetic Programming (GP) to solve this problem and demonstrated how powerful is the new GP-based search method. In the second application, we used Convolutional Neural Networks (CNN) to develop a vision-based steady-state detector (SSD) for steady-state Real-Time Optimizers. It was our purpose to investigate if this vision-based SSD has higher accuracy than established statistical SSD. We found that they have comparable performances, but the CNN-based detector possesses certain advantages that the others do not have.

Part I

Introduction

How to keep the operation profitable has always been a critical question to chemical plants. Resolving this question becomes more and more challenging as increasingly higher standards on the manufacturing and on the products are imposed. As a result, interest in process optimization has surged, and many techniques have been developed to address the issue.

Among these techniques, steady-state Real-Time Optimization (SRTO) is the most widespread in the industry. SRTO calculates inputs u to improve plant performance through solving an optimization problem as following:

$$\begin{aligned} \min_{u} \quad & J(u, x, d) \\ \text{subject to:} \quad & f(u, x, d) = 0 \\ & h(u, x, d) \leq 0 \end{aligned} \tag{1}$$

where x represents states of the plant, d represents disturbances, J is the function that we wish to optimize (such as the profit), $f(u, x, d) = 0$ is the steady-state model of the plant, and the remaining constraints are other operating constraints. To guarantee the optimality of the solutions, we must have accurate estimates of disturbances d, which must be based on steady-state measurements only. If transient data is fitted to a steady-state model, the d estimates are prone to be incorrect and can significantly affect the accuracy of the solution of the problem in Equation 1. Steady-state detector (SSD) is the component of SRTO that determines whether a measurement satisfies the steady-state requirement and should be used for d estimation or not. Thus, SSD has a significant impact on the performance of SRTO.

Although there are many SSD methods, they all rely on numerical statistical tests [1] and require appropriate tuning of their parameters, which is far from trivial. In this thesis we have developed a graphical-based SSD method with easier tuning of parameters. The method uses Convolutional Neural Network (CNN) to investigate plots of measurements to evaluate if they are stationary or not. This method is shown to have comparable performance to numerical methods.

Another approach to process optimization that has recently received significant attention is Self-Optimizing Control (SOC). It can be defined through the concept of self-optimizing variables (SOC-CVs), which has been stated in [2] as: ”A set of controlled variables is called self-optimizing if, when it is kept at constant setpoints, the process is operated with an acceptable loss with respect to the chosen objective function (also when disturbances occur).”

SOC can be complimentary to SRTO [3]. As discussed, in an SRTO implementation, the economic optimization is not triggered until the process is stationary. Hence, if the disturbances affecting the process change frequently, SRTO cannot update the process inputs to the new optimal levels. As a result, the process operates suboptimally during the transient periods. Now, assume that we have integrated SOC variables into SRTO. Controlling these variables to their setpoints will automatically lead to near-optimal operation of the plant during transients, even though the SRTO is still inactive. Thus, combining SOC and SRTO helps to increase the plant profit even more than using SRTO alone.

It is clear that searching for SOC-CVs is the core problem of SOC, especially ”global” SOC-CVs - ones that have acceptable loss over a wide range of operating conditions. There are only 3 ”global” searching methods so far, which are: polynomial zero loss method, regression approach, and global approximation of controlled variables [4]. We introduce here a new search method that utilizes Genetic Programming (GP) to ”evolve” random CVs into SOC-CVs. In our case studies, GP has discovered SOC-CVs with better performances and larger operating ranges than SOC-CVs found by other methods.

This thesis consists of three parts. The first part is this introduction. In the second part, we discuss the application of Genetic Programming in self-optimizing variables search. We devote the third part to the application of CNN in SRTO.

Part II

Self-Optimizing Control

1 Self-Optimizing Control (SOC)

1.1 What is SOC and Why SOC?

We have already mentioned the definitions of SOC and self-optimizing variables in the Introduction. Here we discuss an example of SOC to illustrate the concepts more clearly.

Consider a runner who wants to finish two races. The distances in the first and second races are 100 m and 15 km, respectively. The objective is to finish the runs as soon as possible. The question here is how he/she should run to achieve his/her goals or, in terms of Process System Engineering, what to control. The answer is straightforward in the first case: keep his/her speed close to the maximum. The answer in the second case is not as clear, but keeping his/her heart rate constant is a good solution. If he/she runs at a "high" speed and let the heart rate increase continuously to its maximum, he/she risks himself/herself being too exhausted to finish the run. In contrast, if he/she runs at a "low" speed, the run takes considerably longer than the shortest time possible. Maintaining the heart rate at a fixed level helps the runner avoid both scenarios and achieve a good result. Furthermore, it does that without the need for the runner to frequently adjust the heart rate setpoint based on external factors such as wind speed and slope of the current running path. Thus, the heart rate here is a self-optimizing variable.

The idea of SOC in a chemical process is similar. We wish to identify variables such that when controlling them to a few predetermined setpoints, the process performance automatically gets closer to the optimal. This process optimization is traditionally achieved by the use of RTO, which requires solving an optimization problem repeatedly in real-time. The solving process must be sufficiently quick, or the found solutions can become obsolete otherwise. Thus, RTO methods have a high demand for computational power and can be infeasible for large, complex processes. SOC, in contrast, does not have this requirement and can play an important role in optimizing these processes. However, it is not clear and intuitive which variables are self-optimizing and how good are their performances (or how close is the process performance to the optimal performances when controlling each CV). We need methods to search for and evaluate them.

1.2 Methods to identify self-optimizing CVs

Many methods have been developed to identify self-optimizing variables. Here we only discuss methods for continuous processes. We can divide these methods into model-based and model-free approaches. In the model-based domain, there are three groups: brute force method, local methods, and global methods.

The brute force method is the earliest and also the simplest [4]. The idea is to evaluate the loss of all possible controlled variables (CVs) under all possible operating conditions. The loss of a self-optimizing variable is the difference between its performance and the optimal performance. Each operating condition is an unique combination of values of disturbances and noise. As the name suggests, the brute force method is not efficient and may be impractical for large-scale problems.

Local methods, such as the Null-space method [5], result in CVs with good performance in the proximity of the nominal operating conditions. We must evaluate them again to select a CV that performs well in a different operating region. Moreover, they do not work in operating conditions in which the set of active constraints has changed. Since we focus on searching for global SOC CVs in our method, we will not discuss the local methods in detail here. For more information, please refer to [4].

The global methods group consists of 3 approaches. These are the polynomial zero loss method [6], the regression method [7], and the global approximation of controlled variables [8]. As the name of this group implied, CVs found by these methods can work in larger operating ranges instead of only in the proximity of a certain condition. However, it should be noted that the set of active constraints must remain unchanged throughout these ranges, as in the case of local methods.

1.2.1 Polynomial zero loss method

Assume we can approximate the objective function and the steady-state model of the plant as polynomials $\bar{J}(u, x, d)$ and $\bar{g}(u, x, d)$ of u, x, and d. Here u and x represent the inputs and the states of the plant, while d represents the disturbances. We also have the outputs of the plant as polynomial functions of u, x, and d, $y = m(u, x, d)$.

At each condition of d, we can find the optimal input $u^\star$ by solving:

$$\begin{aligned} \min_{u} \quad & \bar{J}(u, x, d) \\ \text{subject to:} \quad & \bar{g}(u, x, d) = 0 \end{aligned} \tag{2}$$

In the region around the optimum, if the linear independence constraint qualifications (LICQ) and the sufficient secondary conditions for optimality hold, then the first-order necessary optimality conditions (NCO) are satisfied at the optimum:

$$\begin{aligned} \nabla \bar{J}(u^\star, x^\star, d) + \nabla \bar{g}(u^\star, x^\star, d) \cdot \lambda^\star = 0 \\ \bar{g}(u^\star, x^\star, d) = 0 \end{aligned} \tag{3}$$

The method shows that it is possible to analytically eliminate $\lambda^\star, x^\star$, and d from the NCO to obtain equations $R_k(u^\star, y^\star) = 0$. The functions $R_k(u, y)$ of the measurable variables u and y are equal to zero if and only if the NCO is satisfied. Controlling them to 0 automatically leads to optimal performance of the plant under different operating conditions. Thus, $R_k(u, y)$ are self-optimizing variables.

As the elimination process does not depend on specific values of x and d, the self-optimizing CVs can work in a wide range of operating conditions. However, if the plant is in a region where polynomial functions do not approximate its true behavior well, the CVs' performances deteriorate. We can see this in a case study later. The method also has another limitation: the computed CVs may be too complex to be of any practical use.

1.2.2 Regression method

The regression method attempts to find functions of measurements $c = h(y)$ and their setpoints c_s to approximate the NCO and minimize the loss over many operating conditions. Jaeschke and Skogestad [9] proved that the loss at an operating point when we control a CV $c = h(y)$ to its setpoint c_s is:

$$L = \frac{1}{2} || J_{uu}^{-1/2} \cdot (J_u - (h(y) - c_s)) ||_2^2 \tag{4}$$

where, $||\ ||_2$ denotes the L2 norm, J_{uu} and J_u are the reduced Hessian and the gradient of the objective function J, respectively. Therefore, the average loss at N operating points can be calculated as:

$$\theta = \frac{1}{2N}\sum_{i=1}^{N} ||{J_{uu}^{(i)}}^{-1/2} \cdot (J_u^{(i)} - (h(y^{(i)}) - c_s))||_2^2 \tag{5}$$

In the regression method, $h(y)$ are usually simple functions of the measurements, for example, $h(y) = H \cdot y$. The method tries to find H and c_s such that θ is minimal. To accomplish this, it needs numerical values of the reduced Hessian $J_{uu}^{(i)}$, the gradient $J_u^{(i)}$, and the measurement $y^{(i)}$ at each operating point $\{u^{(i)}, d^{(i)}, n^{(i)}\}$ ($n^{(i)}$ represents values of noise in output measurements). These are obtained by simulating the process under N operating conditions.

One limitation of the regression approach is the need for a CV adaptation scheme. Since the CVs $c = h(y)$ have the form of simple functions, their regions of adequate performance are relatively narrow. Therefore, when the process enters a region where the current CV does not work well, we need the adaptation scheme to switch to better CVs.

1.2.3 Global approximation of controlled variables method

As in the regression approach, this method also attempts to find a CV $c = H \cdot y$ that minimizes an average loss function over N operating points. H could be obtained by solving the following optimization problem:

$$\begin{aligned} \min_{H} \quad & \frac{1}{N}\sum_{i=1}^{N}(L_d^{(i)} + L_n^{(i)}) \\ \text{subject to:} \quad & HG^y_{nom} = J_{uu,nom} \\ & L_d^{(i)} = \frac{1}{2} y^{*(i)T} H^T J_{cc}^{(i)} H y^{*(i)} \\ & L_n^{(i)} = \frac{1}{2} trace(W^2 H^T J_{cc}^{(i)} H) \\ & J_{cc}^{(i)} = (HG^{y,(i)})^{-1} J_{uu}^{(i)} (HG^{y,(i)})^{-1} \end{aligned} \tag{6}$$

where $L_d^{(i)}$ and $L_n^{(i)}$ are the losses due to disturbances and due to noise, and $W = E(nn^T)$ for independent noise. Details and meanings of the quantities in the optimization problem can be found in [8]. But, to complete the problem formulation, we need information about the optimal output value - $y^{*(i)}$, the gain of y for u - $G^{y,(i)}$, and $J_{uu}^{(i)}$ at each disturbance condition $d^{(i)}$. Therefore, in this method, we have to run the process simulation at N disturbance conditions and record this information.

The main limitation of this method: it is not easy to find a solution to minimize the loss function due to the non-convexity of the optimization problem.

2 Genetic Programming

In this thesis, we have developed a new method to search for SOC variables based on Genetic Programming. Therefore, in this section, we will illustrate Genetic Programming and its concepts by describing its application in one of our case studies, the heat exchanger network (HEN).

2.1 HEN problem description

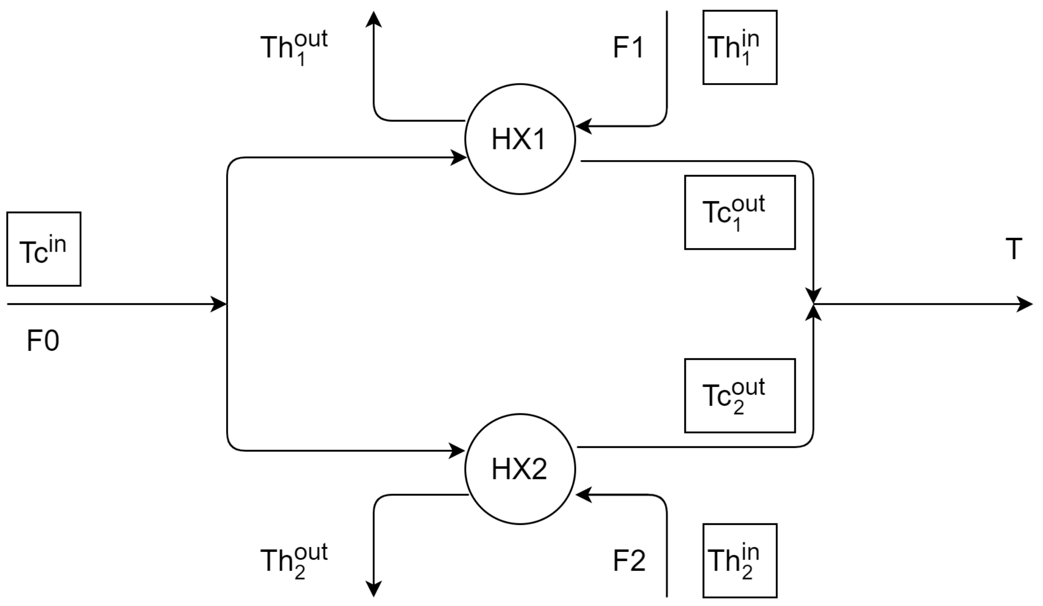


Figure 1: *Network with 2 parallel countercurrent heat exchangers*

Assume that we have a system of two countercurrent heat exchangers HX1 and HX2 as shown in Figure 1. The HXs are in parallel. Their heat transfer properties are characterized by UA_1 and UA_2. A cold stream F0 with flowrate ω_c and temperature Tc^{in} is split into two. Each of these two streams is heated up in a heat exchanger. The proportions of F0 to HX1 and HX2 are u_1 and u_2, respectively. The hot streams F1 and F2 to HX1 and HX2 have flow rates, inlet temperatures, and outlet temperatures denoted by ω_{hi}, Th_i^{in}, and $Th_i^{out}(i \in \{1, 2\})$. Temperatures of the cold streams at the outlets of the heat exchangers are Tc_1^{out} and Tc_2^{out}. The two heated streams are then mixed, resulting in a stream at temperature T. Units of the temperatures and the flow rates are °C and $\frac{kg}{s}$, respectively. Unit of UA_1 and UA_2 is $\frac{kW}{°C}$

Assume that the fluids do not change phase and the specific heat capacities c_p ($\frac{kW \cdot s}{kg \cdot °C}$) of all streams are known, identical, and do not change with temperature. The disturbances in the system are $\{Tc^{in}, \omega_c, Th_1^{in}, Th_2^{in}, \omega_{h1}, \omega_{h2}, UA_1, UA_2\}$. The manipulated variables (MVs) are u_1 and u_2. We have 5 available measurements, which are $\{Tc^{in}, Tc_1^{out}, Tc_2^{out}, Th_1^{in}, Th_2^{in}\}$.

The model equations are:

$$f1 : u_1 + u_2 - 1 = 0$$

$$f2 : u_1 \cdot Tc_1^{out} + u_2 \cdot Tc_2^{out} - 1 \cdot T = 0$$

$$f3, f4 : Th_i^{in} - Th_i^{out} - u_i \cdot (Tc_i^{out} - Tc^{in}) \cdot \frac{\omega_c}{\omega_{hi}} = 0 \quad (i \in \{1, 2\})$$

$$f5, f6 : Tc^{in} - Tc_i^{out} + \frac{UA_i \cdot \Delta T_i}{u_i \cdot \omega_c \cdot c_p} = 0 \quad (i \in \{1, 2\})$$

where, f1 is the cold side mass conservation equation. f2, f3, f4 are the energy balance equations. The control volume of f2 is the mixing point of the two cold streams, while the control volumes of f3 and f4 are HX1 and HX2, respectively. f5 and f6 model the heat transfer within HX1 and HX2, in which ΔT_i are the mean temperature differences, i.e. the heat transfer driving force. We will discuss the mean temperature differences in more details in Section 4.1.

In each operating condition, values of 8 disturbances are given. Thus, there are 7 remaining variables $\{Th_1^{out}, Th_2^{out}, Tc_1^{out}, Tc_2^{out}, T, u_1, u_2\}$ but only 6 equations. Thus, the system has one degree of freedom (DOF). Assume that we select u_2 as the DOF. If we specify the value of u_2, the system is fully determined.

Our objective is to maximize T in all operating condition. Let T_{opt} denotes the highest possible value of T in each condition. We can achieve $T = T_{opt}$ by controlling u_2 to its optimal value $u_{2,opt}$. $u_{2,opt}$ in each condition can be obtained by formulating and solving the following optimization problem:

$$\min_{u_2} \quad -1 \cdot T$$
$$\text{subject to:} \quad f1, f2, f3, f4, f5, f6$$

Alternatively, we can use u_2 to control a self-optimizing variable $c = h(y)$ to its setpoint c_s. The system is also fully determined in this case. The value of T when controlling c to c_s can be determined by solving the system of equations $\{f1, f2, f3, f4, f5, f6, f7\}$, where f7 is the equation $c = c_s$. Note that $T \leq T_{opt} \quad \forall c$ in all operating conditions.

2.2 How to use GP to search for self-optimizing variables

Here we will use GP to search for self-optimizing variables. GP is an evolution-inspired algorithm. It starts with a "population" of N "individuals"/ "programs". In our case, an individual is a CV, which can be a single measurement or a combination of different measurements. These individuals are later subjected to selection and modification by genetic operations to create the next generation of the population. The selection process is based on individuals' fitnesses, i.e. the performances of CVs, whose definition is in Section 1.1. A CV that has a greater fitness has a higher chance to be selected and become the base to create the next CV generation. CVs in the new generation are again evaluated, selected, and modified to create another generation. GP continues to form new generations until it finds a CV with an acceptable fitness or the limit of generation number $N_{generation}$ has been reached.

Algorithm 1 shows the pseudocode of GP. We will now explain concepts and details that are essential to understand the steps in this pseudocode.

Algorithm 1: Pseudocode of GP algorithm *(adapted from [10])*

```
1: Randomly create an initial population of individuals from the available
   primitives
while an acceptable solution is not found and maximum number of generations
  is not reached do
    2: Execute each individual and ascertain its fitness
    3: Select one or two individual(s) from the population with a
       probability based on fitness to participate in genetic operations
    4: Create new individuals by applying genetic operations with specified
       probabilities
end
5: Return the best-so-far individual
```

2.2.1 Step 1 - CV representation, the terminal set, the function set, and the primitive set in GP

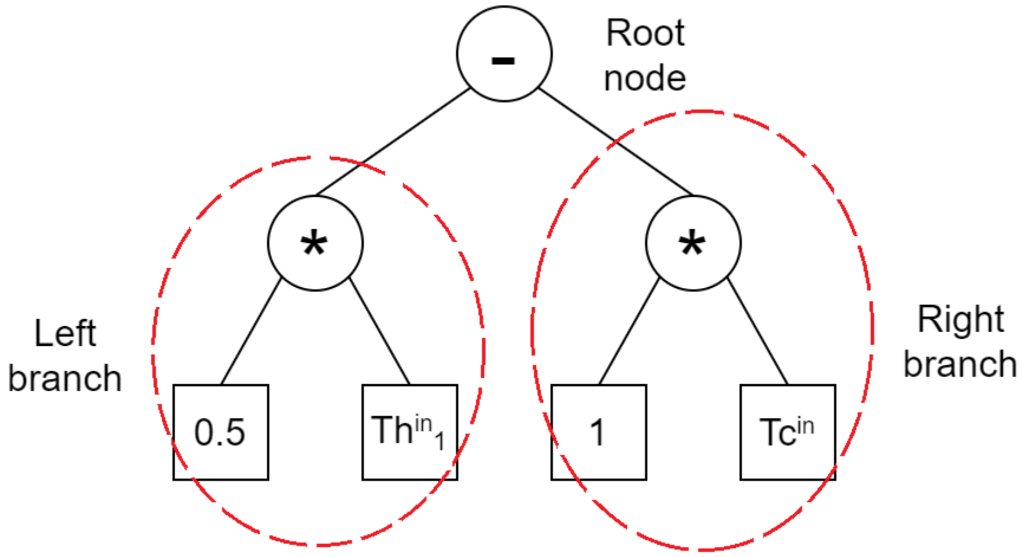


Figure 2: *The syntax tree of* $c = 0.5 * Th_1^{in} - 1 * Tc^{in}$

Consider an arbitrary CV, such as $c = 0.5 * Th_1^{in} - 1 * Tc^{in}$. Its representation in GP is a syntax tree with many nodes and a root node as depicted in Figure 2. Each node is represented as a circle or a square and contains a component of the CV. The syntax tree has two branches: left and right. It also has many "smaller" trees, each radiating from one of its nodes. For example, the left branch is a tree with 3 nodes radiating from the root node " $*$ ", and the left branch of the left branch is a tree with 1 nodes radiating from the root node "0.5" . Any "smaller" trees are called as subtrees of the original tree. Note that the tree is a subtree of itself.

Components of a CV fall into 1 of the following 3 categories: a scalar ("0.5", "1"), a measurement ("Th_1^{in}", "Tc^{in}"), or an arithmetic operation ("-", "*"). These three components are the basic ingredients to construct any CV. GP has a "function set" containing all arithmetic operations and a "terminal set" containing all measurements and scalars (actually, the set contains a random real number generator instead of multiple scalars). These two sets, "function" and "terminal", together form the "primitive set" mentioned in Step 1.

Population initialization

Another concept has been mentioned in Step 1 is the initial population. How to construct the initial population? There are two basic methods to create individuals in this population, or more precisely, their random syntax trees. These are are the full method and the grow method. Understanding these requires the following definitions: depth of a node, depth of a tree, and tree size. The depth of a node is the number of edges from itself to the root node. The depth of a tree is taken as the depth of its deepest node. Tree size the total number of nodes in the tree. For example, consider the tree in Figure 2. The depth of the node containing the scalar 0.5 is 2, the depth of the tree is 2, and the tree size is 7.

The two mentioned initialization methods have many similarities: both synthesize trees node-by-node and require users to specify a maximum depth $depth_{max}$. Moreover, they always assign randomly elements in the "function set" to the root nodes and elements in the "terminal set" to the nodes at the maximum depth. But what they do to the nodes between depth zero and the maximum depth are different. The grow method randomly picks elements from the whole "primitive" set to fill in these nodes, while the full only picks elements of the "function" set. As a result, trees generated by the full method have constant-length branches assume that the operators in the "function" set are all binary (operate on two elements), which makes their size and shape more uniform than those from the grow method.

Consider the example in Figure 3. The random tree on the left is from the full method, while the right one is from the grow method. They both have a max depth of 2. As it is possible for the grow method to select components from both "terminal" and "function" sets at depth 1 and 2, we see that the right branch of the tree on the right terminates at depth 1 while the left branch terminates at depth 3. In contrast, both branches of the "full" tree must terminate at depth 3 since only arithmetic operators could present in nodes at depth 1 and 2 and the branches cannot terminate earlier.

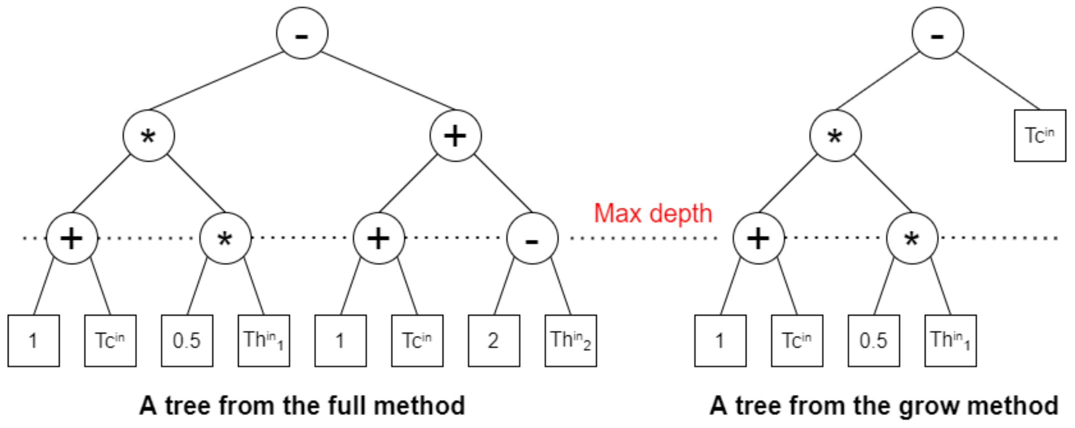


Figure 3: *Syntax trees of random CVs from the full and grow methods (max depth of 2)*

In practice, the two methods are used in combination as they help to diversify the trees' morphology. This enables a broader range of CVs to be represented and considered in GP so it has a higher chance to find a good solution.

2.2.2 Step 2 - Fitness

In biology and GP, fitness represents how well an individual adapts to its surrounding "environment". In our application, we wish to obtain a CV that results in as high as possible values of T in all disturbance conditions. So the "environment" here is the disturbance conditions. But since it is impossible to consider an infinite number of situations, our "environment" must be a set of a finite number of conditions instead. We use the symbol D to denote a disturbance condition and $\mathbb{S} = \{D_1, D_2, ..., D_n\}$ to denote a set of n disturbance conditions ($n \in \mathbb{Z}^+$).

Assume that we have a CV c and its setpoint c_s. We denote the value of T obtained when controlling c to c_s in a disturbance condition D_i by T_{D_i}. A better CV should have higher values of T_{D_i} ($\forall D_i \in \mathbb{S}$), or equivalently, a higher value of $\bar{T}$, where $\bar{T} = mean(T_{D_i})$. Therefore, in this application, we can define the fitness of a CV as $\bar{T}$. Let T_{opt,D_i} be the optimal value of T in D_i and $\bar{T}_{opt} = mean(T_{opt,D_i})$ ($\forall D_i \in \mathbb{S}$). An optimal CV should have its fitness $\bar{T}$ equal to $\bar{T}_{opt}$. Details in how to solve for T and T_{opt} in each disturbance condition D_i have been discussed in Section 2.1.

2.2.3 Step 3 - Individual selection

A question in Step 3 is how individuals are selected here. To do this, GP uses a 2-step mechanism called tournament selection. In the first step, each tournament randomly picks up a small number of individuals, $n_{candidate}$, from the population. For example, let us take $n_{candidate} = 5$. Then, the best individual among these five is picked up and fed to Step 4 in the algorithm. As the first selection step is random, individuals other than the best one could still be chosen. This helps maintain the diversity of the populations.

2.2.4 Step 4 - Genetic operations

There are 2 genetic operations in GP: mutation and crossover. Their role is to generate new CVs from existing CVs.

Mutation: Note that we just describe one type of mutation here - subtree mutation. Although there are other types, such as point mutation, subtree mutation is the most common choice in GP [10]. When a tree undergoes mutation, one of its nodes is selected randomly. Then, the subtree radiated from this node is replaced with a random tree. This tree is generated by the methods used in Step 1. As a result, we get a new random CV from an old CV through mutation.

Crossover: In contrast to mutation, we obtain two new CVs from two parental CVs. Here, we also have 2 randomly selected subtrees, each from a parental CV. They are also replaced, not with randomly generated trees as in mutation, but with each other. This means that after crossover, parts of parent 1 are transferred to parent 2, and vice versa. As crossover requires 2 individuals, 2 tournament selections have to be carried out in advance.

In practice, to determine which nodes in a CV to conduct genetic operations, the algorithm first assigns a random number $r \in [0, 1]$ to each node. Then it compares these numbers r to the probability thresholds specified by the users (P_{XO} and $P_{mutation}$ for crossover and mutation, respectively). Starting from the top of the tree, if we encounter a node whose associated number is smaller than the thresholds, that node is selected for conducting genetic operations.

2.2.5 How to identify CV setpoint c_s

This problem is unique to the application of GP in SOC. There are two ways to determine the c_s value corresponds to a CV generated by GP. We can set c_s to 0 and GP will then find CVs that give (near) optimal process performance when being controlled to 0, i.e. approximations of the gradient J_u. A more general approach is to define a nominal operating condition, and let the setpoint c_s of a CV $c = h(y)$ to be equal to $c^\star_{s,nom} = h(y^\star_{nom})$, value of c under the optimal operation at the nominal condition. The latter strategy makes it easier for GP to search and is more appropriate for difficult search problems.

2.3 Rationale behind the application of Genetic Programming in Self-Optimizing CV searching

GP has myriads of applications developed over time. One most recent example that proved GP's power could be found in [11]. Despite working on an enormous search space, GP successfully rediscovered Deep Learning algorithms from basic arithmetic operations. The established applications enable researchers to study and summarize the properties of a potential problem for GP. [10] has listed a few examples of these, including:

- The sizes and shapes of solutions (solutions' syntax trees, more precisely) are unknown
- Analytic solutions by mathematical analysis are not available
- Approximate solutions can be accepted
- It is challenging to obtain good solutions but easy to evaluate and test the performance of a candidate
- There is a significant amount of testing data

Choosing self-optimizing variables is non-intuitive, as well as determining which measurements should be combined to combine them and the complexity of these combinations. Also, given that a plant model is available, we can evaluate and test the performance of a CV by repeatedly solving the fully determined system of equations at many operating conditions, as discussed in Section 2.2.2. These properties suggest that GP is a potential method to find self-optimizing CVs.

Moreover, it is hypothesized that "biological systems by natural selection through millions of years must have developed simple self-optimizing control strategies" [12]. This hypothesis, if true, means that natural selection could result in sophisticated and effective SOC strategies. Therefore, it further motivates us to apply GP, a natural selection-inspired algorithm, to search for self-optimizing variables.

3 Case study I: Toy Example

Instead of starting with the mentioned HX example, we use a toy case study to illustrate the capabilities of GP in identifying SOC variables. This toy case study, adapted from [5], has relatively simple "global" self-optimizing: these are in the form of linear combinations of the measurements.

3.1 Problem description

There are one unconstrained degree of freedom u, and one disturbance d in our process. At the nominal condition, $d = 0$. There are also 2 available measurements y_1 and y_2, where $y_1 = 0.9 \cdot u + 0.1 \cdot d$ and $y_2 = 0.5 \cdot u - d$. Our goal is to minimize the objective function $J(u, d)$ when d changes, where:

$$J(u, d) = (u - d)^2 \tag{7}$$

3.2 Analytical solutions

Alstad and Skogestad have shown in [5] that the “global” CVs are in the form of $c = h_1 \cdot y_1 + h_2 \cdot y_2$ (where h_1, h_2 are real numbers and $h_1 = 0.5 \cdot h_2$. Controlling these variables to their setpoints $c_s = 0$ gives optimal operation under all conditions.

3.3 Settings of Genetic Programming experiment

3.3.1 Fitness evaluation

Our goal is to minimize $J(u, d)$ at all d conditions. Therefore, the lower the loss sum $\sum_{\forall d} J(u, d)$ when controlling a CV, the higher its fitness. Thus, we define fitness of a CV as:

$$\frac{1}{0.5 + \sum_{\forall d} J(u, d)} \tag{8}$$

The scalar 0.5 in the denominator helps prevent division by zero from occurring when GP finds an ideal CV with $\sum_{\forall d} J(u, d) = 0$. This ideal CV has a fitness of 2.

Knowing that the ideal CVs are in the form of linear combinations, we use only 2 operating points, $d = 0$ (nominal condition) and $d = 0.2$, to evaluate the fitness of each CV $h(y_1, y_2)$. Thus, we have $\sum_{\forall d} J(u, d) = J1 + J2$, where $J1 = J(u, d = 0)$ and $J2 = J(u, d = 0.2)$. At each point, we substitute the value of d into the equation $h(y_1, y_2) = 0$ and solve it to find u. Then, we substitute this value of u into $J(u, d)$ to obtain the loss at this point.

3.3.2 Identifying CV setpoints c_s

Here we wish to find SOC variables that give near-optimal performance when being controlled to 0. Thus, we set the setpoint c_s of all CVs found by GP to 0 by default.

3.3.3 Parameters

Table 1 represents values of GP parameters used in Case Study I. These values are determined using trial-and-error. They are actually values in our first trial. As they give good results and GP is quite robust to parameters’ values [10], we do not tune them further.

Table 1: *Values of GP parameters used in Case Study I*

Terminal set $S_{terminal}$	$\{y_1, y_2\}$
Function set $S_{function}$	$\{+, -, *\}$
Number of individuals per generation N	60
Max number of generations $N_{generation}$	250
Max tree depth in the 0th generation $depth_{max}$	5
No. of candidates in tournament selection (1st step) $n_{candidate}$	5
Cross-over probability P_{XO}	0.8
Mutation threshold $P_{mutation}$	0.2

3.4 Results and Discussion

The problem in interest is indeed easy as GP often finds the optimal solutions after a few generations (even after only one generation). However, it is interesting to see the abstract syntax tree representation of these solutions. An example is:

$$(((((y_2 \cdot y_2) + (y_1 + y_1)) + y_2) + (y_2 + y_2)) + y_2) - (y_2 \cdot y_2) \tag{9}$$

Although it seems more complex, it is actually equal to $2 \cdot y_1 + 4 \cdot y_2$. That means there are redundant parts in the syntax tree. These parts increase the complexity rapidly but have no impact on the performance of solutions. This phenomenon, usually known as "bloating", is common in GP. It poses a challenge in implementing GP as over-complex solutions, may require a huge amount of computing resources to evaluate their fitnesses while not helping the search. This challenge has to be addressed for GP to find useful solutions in more complex processes.

4 Case Study II: The Heat Exchanger Network

The second process that we apply GP to is the heat exchanger network described in Section 2.1. It has an elegant self-optimizing CV found by the polynomial zero loss method. The availability of the analytical solution and insights on the process provides us a base to evaluate GP's performance.

4.1 Heat transfer modeling - Mean Temperature Difference: LMTD, AMTD, and Chen's approximation to LMTD

Please refer to Section 2.1 for a detailed description of the process and the problem. Here we only discuss details about the mean temperature difference ΔT_i in the heat transfer equations.

ΔT in a countercurrent heat exchanger is usually taken as the logarithmic mean temperature difference. However, as the logarithmic term in LMTD may cause numerical difficulties, many approximations have been developed as alternatives. Among them, the approximation suggested by Chen is probably the best [13] [14]. This Chen's approximation to LMTD is calculated as:

$$\begin{gathered}\Delta T = (\frac{1}{2} \cdot \Delta T_1 \cdot \Delta T_2 \cdot (\Delta T_1 + \Delta T_2))^{\frac{1}{3}} \\ \text{where:} \quad \Delta T_1 = Th^{out} - Tc^{in}, \Delta T_2 = Th^{in} - Tc^{out}\end{gathered} \tag{10}$$

The arithmetic mean temperature difference (AMTD) is another popular approximation to LMTD. However, it is not as accurate as Chen's approximation [14]. The AMTD of a heat exchanger is defined as:

$$\Delta T = \frac{(Th^{in} + Th^{out}) - (Tc^{in} + Tc^{out})}{2} \tag{11}$$

AMTD is a good approximation to LMTD if the heat capacity of the hot and the cold streams are not significantly different.

In this Case Study, we will use GP to find SOC variables for both AMTD-based and Chen's approximation-based heat exchanger networks.

4.2 Analytical solutions: Jaeschke temperature and the self-optimizing CV

Jaeschke and Skogestad [15] have applied the polynomial zero loss method to find self-optimizing CVs for an identical network system. The self-optimizing CV found is based on a quantity called "Jaeschke temperature". For each branch i in the network, the Jaeschke temperature T_{Ji} is defined as:

$$T_{Ji} = \frac{(Tc_i^{out} - Tc_i^{in})^2}{Th_i^{in} - Tc_i^{in}} \tag{12}$$

Since the HEN in interest has 2 branches, there are T_{J1} and T_{J2}. The self-optimizing CV c is the difference between these two: $c = T_{J1} - T_{J2}$. The setpoint c_s of c is 0. For convinience, from now on we use the symbol ΔT_J to refer to the CV.

The basic assumption behind the derivation of ΔT_J is that we could describe the driving forces in HXs by AMTD. Thus, controlling ΔT_J to 0 would result in optimal operation in these cases.

Interestingly, ΔT_J also gives a good performance, even though not optimal, when used in an LMTD-based process. But this applies only if the process is in a region where AMTD approximates LMTD well, i.e. similar heat capacity in both streams. We will use the term "normal" region to refer to this high-performance region.

But there are 2 regions in which the use of AMTD is inappropriate and ΔT_J does not perform well, as stated in [16]. In the first region, which we call "extreme" region I, we have $\omega_{h1} >> \omega_{h2}$ and $UA_1 >> UA_2$. While in the second one, or "extreme" region II, we also have $\omega_{h1} >> \omega_{h2}$ but $UA_1 \approx UA_2$ here. Controlling ΔT_J to 0 in region I results in a large loss in T, and it is impossible for ΔT_J to be 0 in region II.

4.3 Questions to be addressed by GP

Having the analytical CV ΔT_J, its advantages and limitations as references, there are a few questions that we would like to address using GP. These questions can be organized into 4 groups. The first group focuses on the AMTD-based HEN, while the remaining concern the Chen's approximation-based HEN. The groups and their questions have been shown in Table 2.

Table 2: *Groups of questions to be addressed by GP*

	Questions
Group 1:	Could GP rediscover ΔT_J when applied to the AMTD-based HEN? If not, how large are the differences between GP solutions' performance and that of ΔT_J?
Group 2:	As ΔT_J can only give near-optimal T in the normal region, could GP find CVs with better performance and/or fewer measurements required? How do these CVs perform in extreme regions?
Group 3:	Could GP find CVs that perform well in extreme regions? If yes, do these CVs still work in the normal region?
Group 4:	Could GP discover "global" self-optimizing variables, ones that have good performance both in normal and extreme regions?

4.4 Design of Experiments

We have designed and run four experiments, each to address one group of questions mentioned above. We will now discuss the details of these experiments. A summary of these details can be found in Table 4 after Section 4.4.2.

4.4.1 "Environment"

It has been discussed in Section 2.2.2 that each GP experiment requires an "environment" - a set $\mathbb{S}$ of operating conditions. That is also the region of operating conditions that we wish CVs to have a high performance in. As in the table of experiment details, there are 3 sets of operating conditions used ($\mathbb{S}$I, $\mathbb{S}$II, and $\mathbb{S}$III). Set $\mathbb{S}$I represents normal region, while Set $\mathbb{S}$II represents the extreme regions. Set $\mathbb{S}$III is designed to cover both normal and extreme regions. The ranges of these regions have been determined through trial-and-error and are shown in Table 3. The sets are created by sampling points from these ranges using the Latin hypercube sampling method.

As discussed in Section 2.1, for each operating condition, if we specify the value of u_2, the system is fully determined and we can solve for T. Thus, T can be written as a function f of u_2 and d. As shown in [15], the curve $T = f(u_2, d)$ in each operating condition d is flat. That means, for each operating condition d, varying u_2 results in only slight changes in T. Thus, each set should cover a wide range of

Table 3: *Center points and Ranges of Sets $\mathbb{S}I$, $\mathbb{S}II$, and $\mathbb{S}III$*

	Region of $\mathbb{S}$I		**Region of $\mathbb{S}$II**		**Region of $\mathbb{S}$III**	
	Center point	Range	Center point	Range	Center point	Range
Tc^{in}	60	±20	60	±20	60	±20
ω_c	100	±20	100	±20	100	±20
Th_1^{in}	120	±20	120	±20	120	±20
Th_2^{in}	220	±20	220	±20	220	±20
ω_{h1}	30	±20	400	±20	250	±185
ω_{h2}	50	±20	100	±20	115	±25
UA_1	50	±20	600	±100	365	±275
UA_2	80	±20	600	±100	380	±260

operating conditions for the nonlinearity in $f(u_2, d)$ to be significant and the fitness evaluation of CVs to be accurate. This requirement, in turn, calls for more sampling points in each set such that they are representative of their corresponding regions. We have chosen the number of points in each set to be 50 including the nominal point. Therefore, to create a set, we sample 49 points from its range and then add the nominal point.

4.4.2 Other details

We reuse the GP parameters from Case Study I in Case Study II. However, since the problems here are relatively more complex than that in Case I, two important parameters have been increased to provide GP with more time and "diversity" in each generation to search. These are the number of individuals in population N and the maximum number of generations $N_{generation}$. Through trial-and-error, we select the values of these parameters to be 360 and 300, respectively.

Since each run of the GP code requires around 24-36 hours, each experiment is repeated only three times and the best result among them is taken. The only exception is Experiment 1, which we run 6 times to increase the probability of GP discovering ΔT_J. In each experiment, we also record the optimal fitness $\bar{T}_{opt}$ and the fitness $\bar{T}$ of ΔT_J (denoted as $\bar{T}_{Jaeschke}$). These provide references to compare the performances of CVs that GP found. How to compute $\bar{T}_{opt}$ has been discussed in Section 2.1. To compute $\bar{T}$ of ΔT_J, we treat ΔT_J as a CV and use the procedure to compute CVs' fitnesses described in Sections 2.1 and 2.2.2. .

In these experiments, we set the setpoints of the CVs found to $c^{\star}_{s,nom}$, their value in optimal operation under the nominal condition (the centre point of the sets in each experiment). This is the second strategy mentioned in Section 2.2.4.

Finally, we have modified the GP algorithm to prevent "bloating", which has been mentioned in the first case study. As an individual in the next generation is created, its tree size is checked. It is accepted to the new population only if the size is not larger than a threshold or the algorithm must repeat the steps of selection and mutation to create another CV otherwise. Setting a size limit is not a major problem in our application because we may prefer simple CVs with slightly worse performance over high-performing but overcomplex ones. We have taken the tree size of ΔT_J as a reference. As the size of ΔT_J is 21, we set the size threshold at 31, slightly higher than the reference. This setting enables GP to have more CV candidates to consider.

Table 4: *A summary of details of the experiments in Case Study II*

	Exp. for Group 1 (Exp. 1)	**Exp. for Group 2 (Exp. 2)**	**Exp. for Group 3 (Exp. 3)**	**Exp. for Group 4 (Exp. 4)**
"Environment"	Set $\mathbb{S}$I	Set $\mathbb{S}$I	Set $\mathbb{S}$II	Set $\mathbb{S}$III
Process Model	AMTD	Chen's approx.	Chen's approx.	Chen's approx.
Optimal fitness $\bar{T}_{opt}$	88.3226	88.2706	160.4240	144.7455
Fitness of ΔT_J	88.3226	88.2691	Not avail.	Not avail.
Number of experiment repeats	6	3	3	3
Number of individuals per generation N	360			
Max number of generations $N_{generation}$	300			
Max tree depth in the 0th generation $depth_{max}$	5			
No. of candidates in tournament selection (1st step) $n_{candidate}$	5			
Cross-over probability P_{XO}	0.8			
Mutation threshold $P_{mutation}$	0.2			

4.5 Results and Discussions

4.5.1 Experiment 1

Values of $\bar{T}_{opt}$ and $\bar{T}_{Jaeschke}$ confirm the best performance of ΔT_J in the AMTD-based process. GP did not find ΔT_J in the 6 runs done. However, the best CV that GP found has a fitness $\bar{T}$ of 88.3225, quite close to the optimal fitness of 88.3226 already. It has a complex form:

$$(4 \cdot Tc^{in} - Th_2^{in}) - (((Th_2^{in} - Th_1^{in}) \cdot (Tc^{in} - Tc_1^{out})) \\ - ((3 \cdot Th_1^{in} - 2 \cdot Tc_1^{out}) \cdot (Tc_1^{out} - Tc_2^{out}))) \quad (13)$$

It is no surprise that GP could not find ΔT_J. The number of CVs with the same syntax tree size as ΔT_J is enormous. As GP is a stochastic search algorithm, the probability that it finds ΔT_J among all these CVs is not high. But this may not be a problem since it can find other CVs with performances not so different from the optimum.

For extra information, GP has indeed found ΔT_J in one of our preliminary experiments, even though we have used Chen's approximation to LMTD in heat transfer equations in that experiment.

4.5.2 Experiment 2

It could be seen that the fitness of ΔT_J is smaller than the optimal (88.2691 versus 88.2706), although the difference is insignificant. In this experiment, GP found CVs with fitnesses slightly higher than that of ΔT_J. The three best CVs found, in order of decreasing fitness, are:

$$(((Tc_1^{out} - Tc^{in}) \cdot (Tc_1^{out} \cdot Th_2^{in})) - (Tc_1^{out} + ((Tc_1^{out} - Tc_2^{out}) \cdot (((((Tc_1^{out} - 88) + Th_1^{in}) \\ - (Tc_1^{out} - Th_1^{in})) \cdot (Tc^{in} - (2 \cdot Th_1^{in}))) - Tc_2^{out})))) \quad (14)$$

and

$$(((Tc_1^{out} - Tc^{in}) \cdot (Tc^{in} \cdot Th_2^{in})) - (Tc^{in} + ((Tc_1^{out} - Tc_2^{out}) \cdot (((((Tc^{in} - 88) + Th_1^{in}) \\ - (Tc_1^{out} - Th_1^{in})) \cdot (Tc_1^{out} - (2 \cdot Th_1^{in}))) - Tc_2^{out})))) \quad (15)$$

and

$$(((Tc_1^{out} - Tc^{in}) \cdot (Tc^{in} \cdot Th_2^{in})) - (Th_1^{in} + ((Tc_1^{out} - Tc_2^{out}) \cdot (((((Tc^{in} - 88) + Th_1^{in}) - (Tc_1^{out} - Th_1^{in})) \cdot (Tc_1^{out} - (2 \cdot Th_1^{in}))) - Tc_2^{out})))) \quad (16)$$

Their fitnesses are 88.2695, 88.2693, and 88.2693, respectively. Among them, we choose only the best CV to investigate its performance further on Set $\mathbb{S}$II. But it does not perform well on Set $\mathbb{S}$II. Similar to ΔT_J, controlling the CV to its setpoint gives a loss as large as 3-5 °C in some operating conditions or is impossible in the others.

There are 5 measurements in the CV mentioned. We also ran experiments with only 3 measurements in the terminal set to see if GP can find SOC CVs with fewer measurements. The best 3-measurements CV found has a fitness of 88.24, lower than that of ΔT_J. It seems challenging to find CVs that are as good as ΔT_J with less than 5 measurements. Therefore, we always include 5 measurements in Experiments 3 and 4.

4.5.3 Experiment 3

ΔT_J does not perform well in conditions in Set $\mathbb{S}$II. Among these 50 conditions, there are 19 in which it is impossible to control ΔT_J to 0. In the 31 cases remaining, this is possible but results in large losses (range from 2-5 °C in each case). The fitness of ΔT_J in this set of 31 points is 155.5979

GP found CVs that have good performances here. The three best CVs found, in order of decreasing fitness, are:

$$(((Tc_1^{out} - Th_2^{in}) * (2 * Th_1^{in})) + (((37 * Tc_1^{out}) * (Th_2^{in} - Tc_2^{out})) + (Th_2^{in} * ((((Tc_1^{out} - Th_2^{in}) - Th_2^{in}) - Th_2^{in}) - ((Tc_2^{out} - Tc_1^{out}) - Tc_1^{out}))))) \quad (17)$$

and

$$(((87 * Tc_1^{out}) + (Tc_1^{out} - Th_1^{in})) + (((37 * Tc_1^{out}) * (Th_2^{in} - Tc_2^{out})) + (Th_2^{in} * ((((Tc_1^{out} - Th_2^{in}) - Th_2^{in}) - Th_2^{in}) - ((Tc_2^{out} - Tc_1^{out}) - Tc_1^{out}))))) \quad (18)$$

and

$$
\begin{aligned}
&(((87 * Tc_1^{out}) + (Th_2^{in} - Th_1^{in})) + (((37 * Tc_1^{out}) * (Th_2^{in} - Tc_2^{out})) \\
&+ (Th_2^{in} * ((((Tc_1^{out} - Th_2^{in}) - Th_2^{in}) - Th_2^{in}) - ((Tc_2^{out} - Tc_1^{out}) - Tc_1^{out}))))) \quad (19)
\end{aligned}
$$

Their fitnesses in 50 points of Set $\mathbb{S}$II are 160.3415, 160.3389, and 160.3388, respectively. These are close to the total optimal in this 50-points region, 160.4240. We select the best CV to investigate further. This CV can be driven to its setpoint in all cases of Set $\mathbb{S}$II. In the region of 31 points where controlling ΔT_J to 0 is feasible, its fitness is 159.3127. This is significantly better than the fitness of ΔT_J and approaches the optimal in this region (159.3985). However, it has a poor performance in conditions in Set $\mathbb{S}$I. The loss in each case varies from 1 to 4 °C.

4.5.4 Experiment 4

Results from Experiments 2 and 3 suggest that we must include both normal and extreme regions in GP's "environment" to search for "global" CVs. It is what we have done in Experiment 4. With Set $\mathbb{S}$III as the "environment", GP found CVs with good performance in both normal and extreme operating conditions. The three best CVs found, in order of decreasing fitness, are:

$$
\begin{aligned}
&(((18 * Th_1^{in}) - (7 * Th_1^{in})) + ((((Tc^{in} - Th_1^{in}) * (Th_2^{in} - Tc_1^{out})) - (Th_1^{in} * Th_2^{in})) \\
&+ ((Th_1^{in} + (48 - Tc_1^{out})) * ((7 * Tc_2^{out}) - Tc_1^{out})))) \quad (20)
\end{aligned}
$$

and

$$
\begin{aligned}
&(((24 * (Th_2^{in} - Tc_1^{out})) + Tc^{in}) + ((((Tc^{in} - Th_1^{in}) * (Th_2^{in} - Tc_1^{out})) \\
&- (Th_1^{in} * Th_2^{in})) + ((Th_1^{in} + (48 - Tc_1^{out})) * ((7 * Tc_2^{out}) - Tc_1^{out})))) \quad (21)
\end{aligned}
$$

and

$$
\begin{aligned}
&(((17 * (Th_2^{in} + Tc^{in})) + ((((Tc^{in} - Th_1^{in}) * (Th_2^{in} - Tc_1^{out})) \\
&- (Th_1^{in} * Th_2^{in})) + ((Th_1^{in} + (48 - Tc_1^{out})) * ((7 * Tc_2^{out}) - Tc_1^{out})))) \quad (22)
\end{aligned}
$$

Their fitnesses are 144.7291, 144.7290, and 144.7289, respectively. These are close to the optimal fitness of 144.7455. We do not compare with the fitness of ΔT_J

in Set $\mathbb{S}$III since controlling ΔT_J to 0 is infeasible in some conditions.

We select the best CV to check how it performs in Set $\mathbb{S}$I and Set Set $\mathbb{S}$II. In Set $\mathbb{S}$I, its fitness is 87.9837, slightly lower than the optimal fitness and the fitness of ΔT_J (88.2706 and 88.2691, respectively. In Set $\mathbb{S}$II, its fitness is 160.3709, while the optimal fitness is 160.4240 (including the points with infeasible control of ΔT_J). These data suggest that the CV found has near-optimal performances in both normal and extreme operating conditions.

4.6 Summary

The results of the conducted experiments are summarized in Table 5. The first three columns in the table represent the properties of the best CV found by GP in each experiment. The answers to the questions in Table 2 are shown in Table 6.

Through the conducted experiments, we have shown that GP is a potential method for finding self-optimizing CVs. It is capable of searching for nonlinear CVs that have good performance over a wide range of operating conditions. Moreover, it does not require solving difficult optimization problems as in other data-driven SOC variables searching methods. It also has the advantages of being highly automated and requiring minimum human involvement, especially in specifying the form and structure of the solutions.

Table 5: *A summary of results in Case Study II*

	Search sets	**Can work in sets**	**Fitness in the working sets**	**Fitness of ΔT_J in the same sets**	**The optimal fitness in the same sets**
Exp. 1	Set $\mathbb{S}$I	Set $\mathbb{S}$I	88.3225	88.3226	88.3226
Exp. 2	Set $\mathbb{S}$I	Set $\mathbb{S}$I	88.2695	88.2691	88.2706
Exp. 3	Set $\mathbb{S}$II	Set $\mathbb{S}$II	160.3415	Not avail.	160.4240
Exp. 4	Set $\mathbb{S}$III	Set $\mathbb{S}$I	87.9837	88.2691	88.2706
		Set $\mathbb{S}$II	160.3709	Not avail.	160.4240
		Set $\mathbb{S}$III	144.7291	Not avail.	144.7455

Table 6: *The answers to the questions in Table 2*

	Questions
Group 1:	GP did not rediscover ΔT_J when applied to the AMTD-based HEN. The differences between GP solutions' performance and that of ΔT_J were small (83.3225 versus 83.3226).
Group 2:	GP found CVs with better performances than that of ΔT_J, but not with fewer measurements. These CVs did not perform well in extreme regions.
Group 3:	GP found CVs that perform well in extreme regions. These CVs did not work in the normal region.
Group 4:	GP discovered "global" self-optimizing variables, ones that have good performance both in normal and extreme regions.

4.7 Further Study

When applied to the HEN process, GP found CVs that have performances comparable to the analytical solution ΔT_J. It also found CVs that can work in both the normal and the extreme regions, in contrast to ΔT_J. However, noise in the process has not been considered. It is interesting to see how GP performs when adding noise to the process model. We have not investigated how changes in GP affect its performance either. For example, does switching to another GP algorithm or increasing the tree size limit improve the search?

There are questions regarding the method to be addressed. Most importantly, can it find CVs that can handle active set changes automatically (if these exist) and eliminate the need for CV switching? This would increase the industrial acceptance of SOC and enable its applications in real systems [4]. It would also help to advance the field of dynamic SOC. Then, can we reduce the amount of time it takes for GP to find SOC variables? It took around 24 hours for each run in Case Study II in this work, and to find good solutions may require 4-5 runs. It was a considerable amount of time. Can we combine GP with other methods to improve its searching efficiency? For example, using insights or results from the polynomial zero loss method to aid GP in its search by modifying the terminal/function set. Finally, can we extract any pattern from the high-performing CVs that GP found and learn from these patterns to design better self-optimizing variables?

Part III

Steady-state detection in SRTO using Convolutional Neural Networks

5 Steady-state Real-Time Optimization (SRTO)

5.1 Components of SRTO

As mentioned in the Introduction, to find inputs that improve the plant performance, SRTO must repeatedly solve an economic optimization problem. Thus, it needs a solver to fulfill this task. It also has two other crucial components: a parameter estimator and a steady-state detector.

The parameter estimator is important to guarantee that the RTO model used in the economic optimization layer is in synchronization with the plant. In the parameter estimation step, selected model parameters are adapted such that the model captures the current system disturbances. Since these disturbances are rarely measured, averaged process measurements are used for representing their effect in the system.

What types of measurements can be used by the estimator depends on the model employed in the RTO method. For example, while dynamic RTO can utilize both steady-state and transient measurements, steady-state RTO can only use steady-state ones. Thus, it is important to determine accurately whether the plant is at steady-state or not and SRTO relies on SSD for this task.

Figure 4 shows how the three main components of SRTO coordinate. Assume that the process has just reached a steady-state and the true current disturbance values are d. SSD detects this event through measurements y and activates the parameter estimator. The estimator then updates the optimization problem with new disturbance values $d_{estimated}$. The economic optimization solves for the new optimal input setpoints u_{sp} and send these to lower control layers for implementation ending the SRTO cycle. The next cycle would starts when the process becomes stationary again.

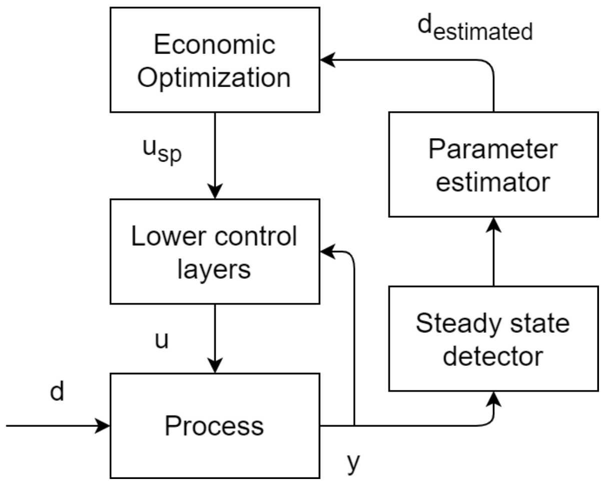


Figure 4: *SRTO diagram*

SSD is a critical component of SRTO as it determines when to trigger the system. If it has a high false positive rate (declaring steady-state while the process is not yet stable), the estimator will use transient data to update the steady-state model, which results in inaccurate estimates of disturbances. Solutions found with inaccurate disturbance values can become sub-optimal, or even worsen the plant performance [17]. In contrast, if SSD has a high false negative rate (declaring transient while the process is already stable), RTO activation only happens long after the plant has been at steady-state. This increases the intervals during which the process is operating in sub-optimal regions. Despite often overlooked in the SRTO literature, the accuracy of the steady state detector has a significant impact on the SRTO performance. Thus, more attention should be paid to investigating SSD methods in realistic situations.

5.2 Steady-state detection methods

According to [1], all existing SSD methods belong to one of these four groups:
• **Group A**: These methods employ linear regression over a data window, following by a t-test on the regression slope. The t-test is used for inferring if the slope does deviate significantly from 0 or not. The plant is considered to be at steady state if the slope is not statistically different from 0.
• **Group B**: In these methods, the means of the two most recent data windows are calculated. A t-test is then used to assert if these two means are different or not. The plant is considered to be at steady state if the two mean values are not statistically different.

• **Group C**: In these methods, the standard deviation is calculated over the most recent data window. The plant is considered to be at steady state if the standard deviation is statistically smaller than a threshold.

• **Group D**: These methods employ two different approaches to calculate the variance of the same dataset, following by an F-test on the ratio of the two variance values. The t-test is used for inferring if the two variances are significantly different. The plant is considered to be at steady state if the two variances are not statistically different.

In this thesis, we have selected two existing SSD methods as the base to evaluate the performance of our new method. The first method, which we refer to by the name SSD1, is proposed by Cao and Rhinehart in [18] and belongs to group D. The second method, which we call SSD2, is from Kelly and Hedengren [17] and belongs to group A. To the best of our knowledge, there is no systematic comparison of the performances of methods in different groups. Moreover, among the methods that we found in the literature, SSD1 and SSD2 have the most detailed method descriptions. They are also easy to understand and implement. Thus, we selected them to use in this thesis. We will now introduce the algorithms in these two methods.

5.2.1 Method SSD1

To implement SSD1, at each time step t, we need to compute the filtered measurement $y_{f,t}$ by applying the exponential moving average filter with filter factor λ_1 ($0 \leq \lambda_1 \leq 1$) to the plant measurements y_t as following:

$$y_{f,t} = \lambda_1 \cdot y_t + (1 - \lambda_1) \cdot y_{f,t-1} \tag{23}$$

We also need to compute the filtered variance $s^2_{1,t}$ of $(y_t - y_{f,t-1})$, the difference between the current measurement y_t and the filtered measurement at the previous step $y_{f,t-1}$, according to the following formula:

$$s^2_{1,t} = \lambda_2 \cdot (y_t - y_{f,t})^2 + (1 - \lambda_2) \cdot s^2_{1,t-1} \tag{24}$$

where λ_2 ($0 \leq \lambda_2 \leq 1$) is the filter factor. Similarly, we compute the filtered variance $s^2_{2,t}$ of $(y_t - y_{t-1})$, the difference between the two most recent consecutive raw measurements, by the following formula:

$$s^2_{2,t} = \lambda_3 \cdot (y_t - y_{t-1})^2 + (1 - \lambda_3) \cdot s^2_{2,t-1} \tag{25}$$

where λ_3 $(0 \leq \lambda_3 \leq 1)$ is the filter factor. Then, the R-statistic is used for comparing these two filtered variances, $s^2_{1,t}$ and $s^2_{2,t}$. Intuitively, if the process is not at steady-state, $s^2_{1,t}$ is much larger than $s^2_{2,t}$ [19]. Thus, we compute the variance ratio $R_t = \frac{(2-\lambda_1)\cdot s^2_{1,t}}{s^2_{2,t}}$ and compare it with R_{crit} [1]. R_{crit} is a tuning parameter, whose value is in the same order of magnitude as 1 . If $R_t > R_{crit}$ then we conclude that the plant is unlikely to be at steady-state.

The authors suggested to set the values of the filter factors $\lambda_1, \lambda_2, \lambda_3, R_{crit}$ as $0.2, 0.1, 0.1,$ and 2, respectively. However, setting $R_{crit} = 2$ is just a starting point and the parameter can require further tuning.

For a multivariable system, we can apply the algorithm to each variable and consider that the system is stationary if some of the variables, e.g. all of them, are at steady-state.

5.2.2 Method SSD2

In SSD2, we assume that the plant measurements at time t can be written as:

$$y_t = m \cdot t + \mu + a_t \tag{26}$$

where the term $m \cdot t$ represents the deterministic drift component, μ represents the mean of the hypothetical stationary process, and a_t represents the white noise component with a zero mean and a standard deviation of σ_a.
Similarly, we have the plant measurements at time $t-1$ as:

$$y_{t-1} = m \cdot (t-1) + \mu + a_{t-1} \tag{27}$$

From these equations, we have:

$$y_t - y_{t-1} = m + a_t - a_{t-1} \tag{28}$$

As the term $a_t - a_{t-1}$ has an expected value of 0, the slope value m is computed as the arithmetic the differences $y_t - y_{t-1}$ over a data window containing n measurements. We can then estimate μ and σ_a over the same data window as:

$$\mu = \frac{1}{n} \cdot (\sum_{t=1}^{n} y_t - m \cdot \sum_{t=1}^{n} t) \tag{29}$$

[1]The term $(2-\lambda_1)$ appears in the numerator of R_t as a consequence of applying filter on the process measurements. Please refer to [18] for more details.

$$\sigma_a = \sqrt{\frac{1}{n-2} \cdot \sum_{t=1}^{n} (y_t - m \cdot t - \mu)^2} \tag{30}$$

One could perform a statistical test on m to check if it is significantly different from zero. However, Kelly and Hedengren showed that performing a t-test on the absolute difference between the measurement and μ (the mean of the hypothetical stationary process) yields more accurate results. For each point y_t in the data window, we check whether $|y_t - \mu| \leq t_{crit} \cdot \sigma_a$, where t_{crit} is the Student's t critical value at a significance level α and a degrees-of-freedom n. If the absolute value of the difference between the measurement and the mean of the hypothetical stationary process μ is smaller than the specified threshold, we assign the value 1 to the corresponding flag p_t. Otherwise, p_t is assigned 0. Finally, we calculate the probability of the plant being at steady-state $P = \frac{\sum_{t=1}^{n} p_t}{n} \cdot 100\%$ and compare it to the cut-off value to give the final conclusion.

It should be noted that the cut-off value and the window length are application-specific and have to be determined manually. The authors recommend to use a window length that is three to five times larger than $\frac{\tau}{\tau_s}$, where τ and τ_s are the time constant of the plant and the sampling interval, respectively. The procedure to apply the SSD2 method to multivariable systems is similar to that of the SSD1 method.

6 Convolutional Neural Network (CNN) and k-Means Clustering Algorithm

6.1 Convolutional Neural Network

Convolutional Neural Network is a kind of neural networks specialized in processing image-like data. They are renowned for their performance in computer vision tasks such as image classification and object detection.

In image classification, we want the computer to classify images in preselected categories. For example, to recognize dog photos and cat photos in a collection of both animals' photos. Object detection is quite similar to image classification. The computer also has to recognize objects in images. But an image may contain many objects from different categories instead of only one. The machine has to recognize all these objects and locate them by drawing a box surrounding each object in the image.

CNNs are built using multiple Fully onnected layers, similarly to other types of neural networks. But they also have two types of layers that are unique to them: the convolutional layers and the pooling layers [20].

6.1.1 Convolutional Layer

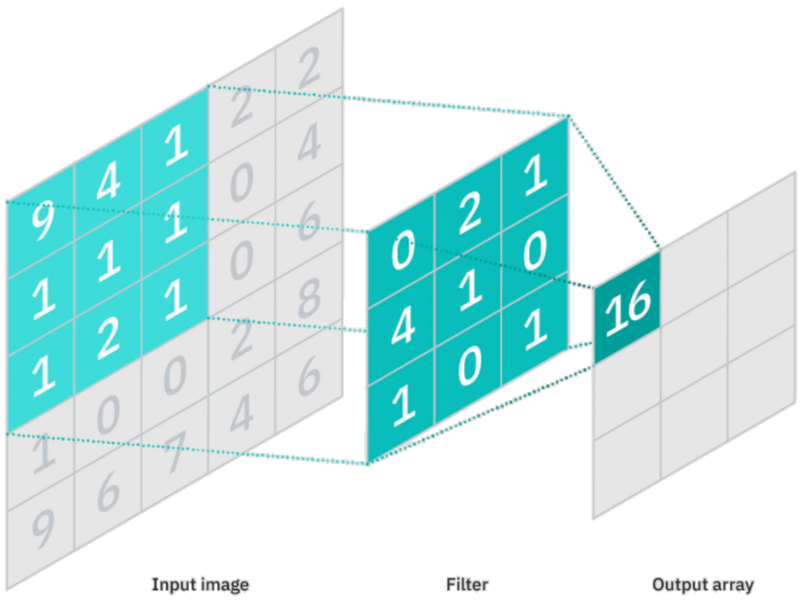


Figure 5: *How a Convolutional Layer processes its input* [2]

Convolutional layers, or filters, are 2-D arrays of scalars. Figure 5 shows an example convolutional layer and how it works when applied to an image. The input image in Figure 5 is represented by a 2-D array of scalars [3], . The filter is also a 2-D array of scalars. The scalars in the arrays of the example image and filter are shown in Figure 5. It is usually the case that an image has larger size than a filter. In the example, the image is a 5x5 array while the filter is a 3x3 array. Therefore, the filter is applied first in the top-left 3x3 region of the image. The output is obtained by taking the dot product between the two 3x3 arrays highlighted in green in Figure 5, which is:

$$(9 \cdot 0 + 4 \cdot 2 + 1 \cdot 1 + 1 \cdot 4 + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 2 \cdot 0 + 1 \cdot 1) = 16$$

[2]Adapted from: https://www.ibm.com/cloud/learn/convolutional-neural-networks

[3]If an image is represented by only one 2-D array of scalars, this indicates that the original image is in black-and-white. Each pixel in an black-and-white image is represented as a scalar, thus the image is represented as a 2-D array. Each scalar in the array represents how dark or bright its corresponding pixel is. Colored images are also represented as 2-D arrays, but each image has more than one array. For example, a RGB image is represented by three arrays, each represents red or green or blue intensities of pixels of the image.

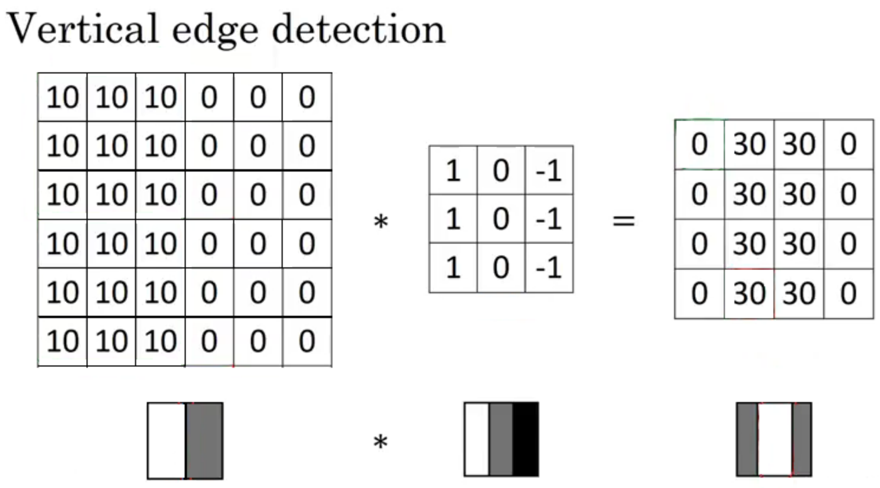


Figure 6: *Vertical edge detection by Convolutional Layer* [4]

We then move the filter one column to the right and apply it again to the new region. The process is repeated until we have applied the filter to the whole image.

The convolutional layers help CNNs to detect features in an image such as straight lines, curves, corners or more abstract features, which is crucial for object identification. Figure 6 represents how a filter detects vertical edges in an image. From left to right, we have matrix representations of the input, the filter, and the output, respectively. Below each array is its corresponding "image". We can interpret the input from the output as following: there are changes in the color of pixels when we move horizontally in the input (0 to 30 to 0) but no change when moving vertically (0 to 0, 30 to 30), thus there are vertical edges in the input. It is as if the filter has detected and retained in its output information about the vertical edge in the input.

Values of the scalars in the filter array determines which feature of an image a filter can detect. Thus, designing a filter is equivalent to determining its array scalars. Traditionally, filters are designed manually to target specific image features, such as the one in Figure 6. However, array scalars of the filters in CNN are not designed by human experts but are learned by the CNN itself during training. Therefore, it is not straightforward to interpret which features are detected by filters of CNNs.

[4]Adapted from: https://kharshit.github.io/blog/2018/12/14/filters-in-convolutional-neural-networks

6.1.2 Pooling Layer

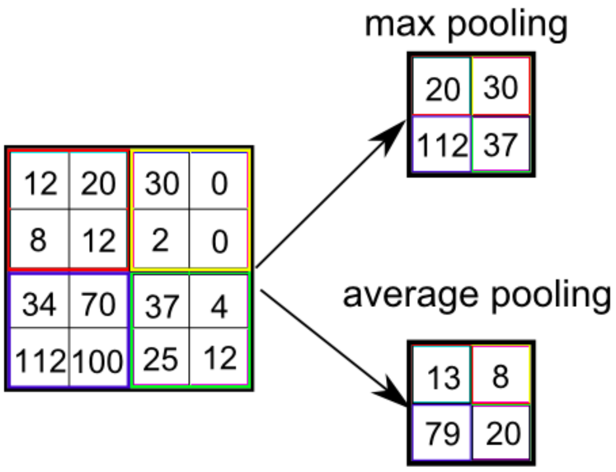


Figure 7: *How Max Pooling and Average Pooling layers process their input* [5]

The function of the Pooling Layers is to reduce the dimensional (thus, the complexity) of the output from its previous layer. Although Convolutional Layers also reduce the size of their inputs, they do that significantly slower than Pooling Layers.

Figure 7 shows the outputs obtained when applying 2x2 Max Pooling/Average Pooling Layer to a 4x4 input. It can be seen that the size of Pooling Layers' outputs is just 2x2, smaller than that of the input. Similar to Convolutional Layers, Pooling Layers also sweep through all regions of the input. However, they are different since they just output the maximum or the average of all scalars in each region instead of the dot product.

6.1.3 VGGNet and its architecture

There are multiple variants of CNN, such as AlexNet, VGGNet, GooLeNet, and ResNet, to name a few. Although they all have the same three building blocks mentioned (the convolutional, the pooling, and the fully connected layers), the network architecture of each variant is different from each other and is specific to that variant only. Since we have used VGGNet in this work, we would like to introduce its architecture in more detail.

[5] Adapted from: https://laptrinhx.com/convolutional-neural-network-cnn-3713577170/

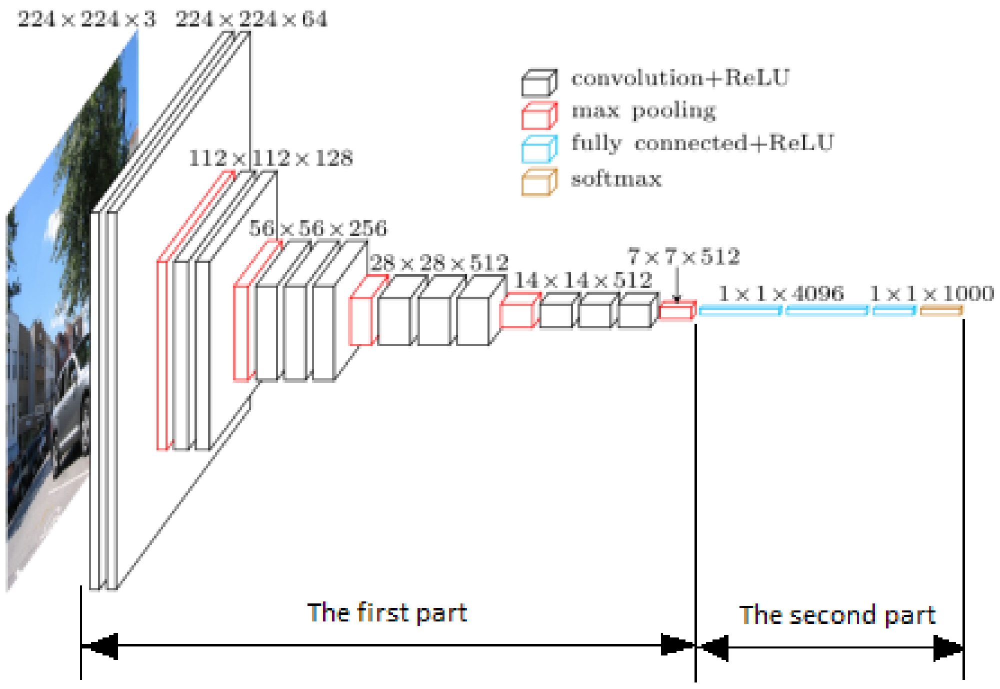


Figure 8: *The architecture of VGGNet* [6]

Figure 8 shows VGGNet's architecture, which could be divided into two parts. In the first part, there are only Convolutional and Max Pooling Layers. The final layer of this first part is a Max Pooling layer, whose outputs are in the form of vectors. These vectors are usually called "feature vectors". The second part consists of Fully Connected Layers and a softmax layer. We have not discussed the softmax layer, but it helps the CNN determine how likely an image belongs to each class or category. Similar to the fully connected layer, the softmax layer is not specific to CNNs and can be found also in other types of neural networks.

It is clear from Figure 8 that VGGNet has many layers and a large number of parameters to be learned, which is also the case with other Convolutional Networks. Therefore, training a CNN requires enormous amounts of data and resources (computational power and time, for example). As a result, it is a common practice in the field of Computer Vision to reuse and modify an established CNN architecture with proven performance. To train an existing CNN for a new application, we usually "freeze" the Convolutional and Pooling layers. That means we keep the configuration and parameters of these layers unchanged during training with training data of the new application. We only change the configuration of Fully-Connected layers (shown in blue in Figure 8) and let the neural network to relearn parameters of these layers

[6]Adapted from: https://www.cs.toronto.edu/ frossard/post/vgg16/

during training, if necessary.

6.2 k-Means Clustering Algorithm

In this thesis, we combine CNN with the k-Means clustering algorithm to use in steady-state detection. Thus, we will now introduce the clustering algorithm.

The k-Means clustering algorithm helps to partition an arbitrary number of data points into k clusters. Each data point is in the form of a vector and it belongs to one cluster only after the partition. The cluster assignment is based on the similarity between points. Usually, the similarity between two arbitrary points is quantified by the Euclidean distance between them. Each cluster has a corresponding cluster centroid, which is the center of the cluster. Algorithm 2 represents the pseudocode of the k-Means algorithm.

Algorithm 2: Pseudocode of the k-Means clustering algorithm *(from [21])*

```
Require:  The number of clusters to find $k$
          A dataset $D$ containing $n$ training instances $\{d_1, ..., d_n\}$
          A distance measure, $Dist$, to compare instances to cluster centroids

1: Select $k$ random cluster centroids, $c_1$ to $c_k$, each defined by values for each
   descriptive feature, $c_i = (c_{i1}, ..., c_{im})$
while being in the first two iterations or there are still cluster reassignments in
  the previous iteration do
    2: Calculate the distance of each instance, $d_i$, to each cluster centroid, $c_1$ to
       $c_k$, using $Dist$
    3: Assign each instance $d_i$ to belong to the cluster $C_i$, to whose cluster
       centroid $c_i$ it is closest
    4: Update each cluster centroid $c_i$ to the average of the descriptive feature
       values of the instances that belong to cluster $C_i$
end
```

A major drawback of the k-Means algorithm is that k has to be specified by the user in advance. But it does not hinder the use of the algorithm in this thesis since knowledge about the number of image categories is usually available in our application, as we will show later.

6.3 Combining VGGNet and k-Means algorithm for image classification

To understand how we can combine VGGNet and the k-Means algorithm for image classification, we must review the architecture of the VGGNet first. As discussed in Section 6.1.3, there are 2 parts in the VGGNet. The first part is the part that consists of Convolutional and Pooling layers only. Its function is to generate feature vectors from input images. The second part consists of Fully Connected and softmax layers. This part is basically a neural network whose function is to determine the category of images through investigating their corresponding feature vectors. It is also the part that needs to be retrained whenever we apply VGGNet to a new application. However, we can replace this part with the k-Means clustering algorithm to form a new CNN, which we refer to as the k-CNN.

The k-CNN is suitable for image clustering tasks. For example, when we show it a mixture of 50 cat images and 50 dog images and tell it to sort these into two, it will automatically sort the images into 1 group consisting of dog images and 1 group consisting of cat images based on the similarities between their feature vectors. The new CNN knows that there are two different types of images in the mixture and which images belong to each type, but it does not know the name of each type.

Although image clustering and image classification are not identical, we can easily modify the procedure to enable the k-CNN to carry out image classification. We also let it sort a few samples of dog and cat images into two groups. Then we tell the k-CNN the name of each group. To determine the category of a new image later, it just needs to determine which sample group is the feature vector of that image closest to and assign the corresponding group name to be the image's category.

We have already mentioned in Section 6.1.3 that parameters in the first part of VGGNet are fixed and do not have to be updated for new applications. Thus, we do not have to retrain this part of VGGNet. The k-Means algorithm, by its nature, does not require training either. As a result, the k-CNN is a plug-and-play algorithm and does not require any training in both image clustering and image classification tasks (although it needs a few examples for reference purposes in the latter case).

7 Methodology - Results and Discussions

The basic idea behind our method of using CNN in SSD is to show plots of plant measurements to the CNN and let it identify whether the plant is at steady-state or not. This mimics the process of steady state detection through plots in humans. From now on, we will call our method the CNN method.

We have designed 5 experiments to investigate the performance of CNN as a SSD method. The plant we studied in Experiments 1 and 2 is a first-order plus delay process. The transfer function of this process is:

$$G_p(s) = \frac{0.703 \cdot e^{-10s}}{1.62s + 1} \tag{31}$$

The nominal operating point of the process is $(u_{nom}, y_{nom}) = (1.5, 7.5)$. The input range is $u \in [1.0, 4.0]$, corresponding to an output range of $y \in [7.14, 9.26]$.

In Experiments 3, 4, and 5, we study how CNN method performs in typical operating scenarios of a chemical plant and typical types of noises, including white noise, colored noise, and noise follows a heavy-tailed distribution. The typical operating scenarios are step response, short ramp response, long ramp response, and steady-state intervals. We have created an output profile, the one represented in Figure 14, that includes all these scenarios and used this profile in our experiments. We also use SSD1 and SSD2 methods discussed in Section 5.2 to identify whether the plant is at steady-state and to compare the performance of the new method with.

Unless stated otherwise, the process noise in the five experiments follows a normal distribution with a mean of 0 and a variance of 0.01. Since each experiment is developed based on the results of the previous one, we will describe the methodology of Experiment 1 first, following by the results and result discussion of Experiment 1, then move on to the methodology, results, and result discussion of Experiment 2, and so on until Experiment 5.

7.1 Experiment 1 - Methodology

In the first experiment, we would like to know whether CNN can "see" the differences between steady-state data plots and transient data plots. Therefore, we generated 2 sets of data plots, the steady-state set (SS) and the transient set (TS). Each set contains 50 plots.

In each plot, the x-axis and the y-axis represent time and output measurement, respectively. The scales of these axes are extremely important for steady-state detection. For example, if the scale of the y-axis is too small, such that it is comparable to the magnitude of noise in the measurement, both SS and TS plots will appear to represent non-stationary measurements to us (and the CNNs). Similarly, if the scale of the x-axis is much larger than the number of data points in the plot, the plot is "compressed". Thus, we cannot observe clearly how the measurements changed over time. As a result, we have to select appropriate scales for both axes. We found that for $G_p(s)$, with the input range of [1.0, 4.0] and the noise level that we selected, the y-axis should range from 5.0 to 10.0. We also determined that the number of data points T in each plot to be in the range of 15 to 50. We let T change from plot to plot to see if the CNN can recognize the similarity between plots in the same set even though they have different number of data points. We found that the x-axis range of [0, 50] is appropriate for $T \in [15, 50]$.

To create a plot in Set TS, we pick up a random input level u_0 in the range [1.0, 4.0], compute the corresponding steady-state output level $y_{SS}(u = u_0)$, and let $y_0 = y_{SS}(u = u_0)$. At each time step $0 < t < T$, we randomly update u_t using a continuous uniform distribution in the range [1.0, 4.0]. Next, we compute the new measurement y_t using the transfer function $G_p(s)$. When we already have T measurements, we add noise to them and plot the noisy measurements. Creating a SS plot is similar to creating a TS plot, except that we let $u_t = u_0$ $(\forall t \leq T)$. Details about the two image sets and their plots are summarized in the Table 7. Figure 9 represents examples of plots from Set SS and Set TS.

Table 7: *Details about Set SS, Set TS, and their plots*

Number of plots in each set	50
x-axis limit in each plot (time axis)	0-50
y-axis limit in each plot (output axis	5-10
Number of datapoints in each plot (T)	Random integer in [15; 50]
Input level u_0 at time $t = 0$	Select randomly in [1.0; 4.0]
Output level y_0 at time $t = 0$	$y_0 = y_{SS(u=u_0)}$
Input level u_t at each time step t $(0 < t \leq T)$	$u_t = u_0$ **(for Set SS)** Select randomly in [1.0; 4.0] **(for Set TS)**

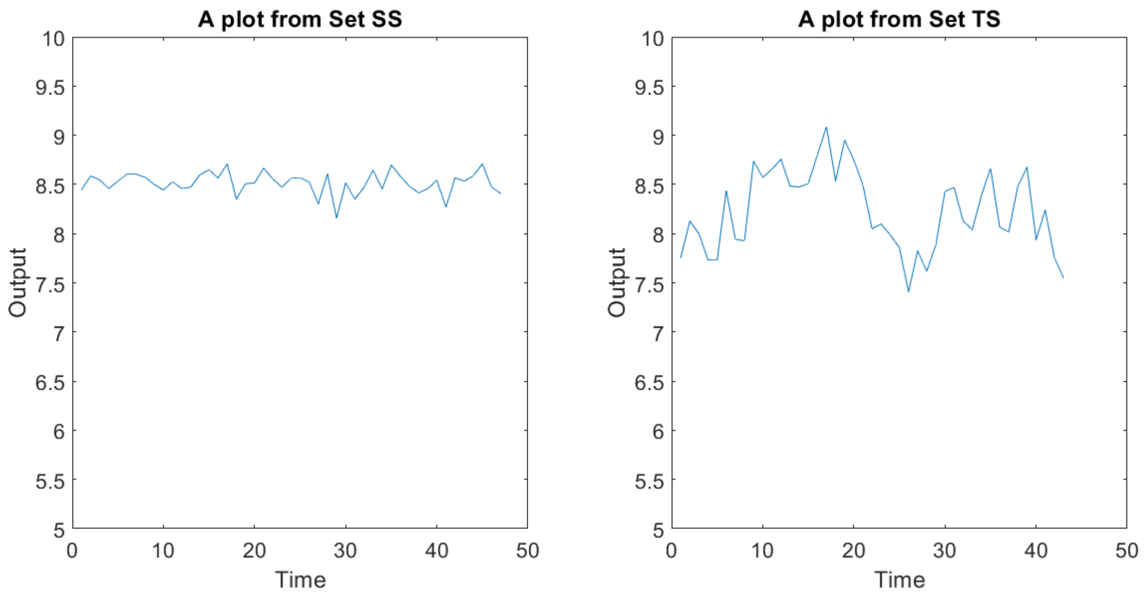


Figure 9: *Examples of plots from Set SS and Set TS in Experiment 1*

The procedure of Experiment 1 has been shown in Figure 10. First, we present the first part of VGGNet with plots from Set SS and Set TS. VGGNet generates one feature vector for each plot. Therefore, we have a set of 100 feature vectors. We hypothesize that plots in the two sets look different through the lens of the CNN, and thus the feature vectors it generates will have the following property: feature vectors from Set SS will, in the vector space, lie closer to each other than vectors from Set TS. Thus, we predict that when subjected to partitioning by the k-Means algorithm with $k = 2$, vectors from both sets will form 2 separate groups, each of which mainly consists of vectors from one plot set. For example, group 1 mainly consists of vectors from Set SS, while group 2 mainly consists of vectors from Set TS and vice versa. Therefore, to verify our hypothesis, in the next step, we partition 100 generated feature vectors into two groups using the k-Means algorithm and investigate how plots in Set SS and Set TS are distributed to the two groups. If the actual distribution agrees with our prediction, we can accept our hypothesis about the steady state detection capability of the CNN method.

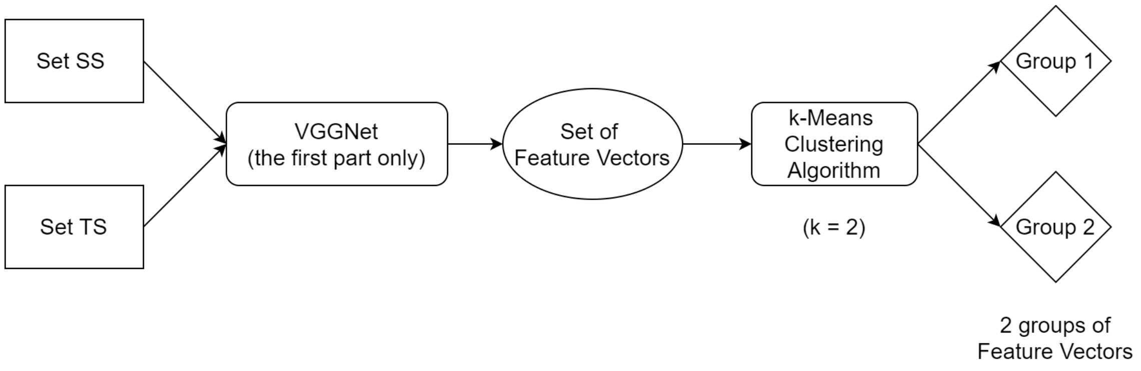
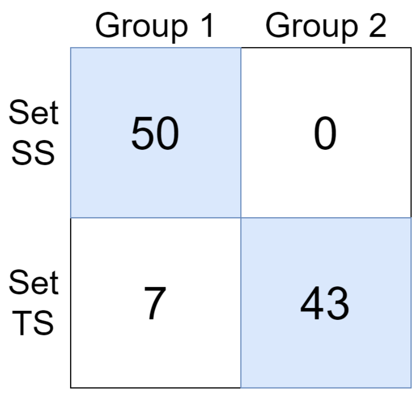


Figure 10: *Procedure of Experiment 1*

7.2 Experiment 1 - Results



	Group 1	Group 2
Set SS	50	0
Set TS	7	43

Figure 11: *Results of Experiment 1*

Results of Experiment 1 are shown in the confusion matrix in Figure 11. Each cell shows the number of plots that are in the group represented by the cell's column and comes from the set represented by the cell's row. For example, the top-left cell tells us that there are 50 plots in Group 1 that come from Set SS, and the bottom-right cell displays that there are 43 plots in Group 2 that come from Set TS. These results suggest that, from the point of view of the CNN, plots of Set TS look different from plots of Set SS. It implies that the CNN method can distinguish between steady-state data and transient data plots and we should investigate its performance further.

7.3 Experiment 2 - Methodology

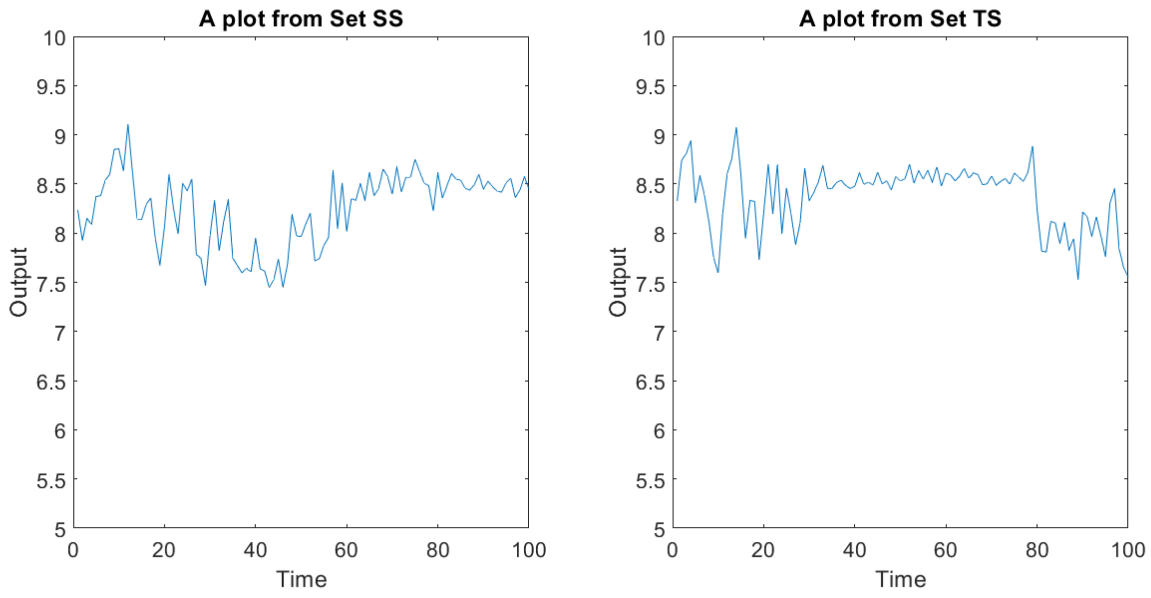


Figure 12: *Examples of plots from Set SS and Set TS in Experiment 2*

The procedure of Experiment 2 is similar to that of Experiment 1. However, there are 2 differences in the plots of Experiment 2. First, there are more data points in each plot (100 instead of 15-50 like before). Therefore, the x-axis limit in each plot has also been changed to [0; 100]. Secondly, plots in Experiment 2 have more intervals than those in Experiment 1. For example, consider the SS plots from Experiment 1 and 2 in Figure 9 and Figure 12, respectively. We can see that the output in the SS plot in Figure 9 fluctuates slightly from $t = 0$ to $t = 50$. Thus, there is only one interval, the steady-state interval, in this plot. In contrast, the SS plot in Figure 12 has two intervals. The first transient interval is from $t = 0$ to $t = 70$, in which the output changes significantly, and the second steady-state interval is from $t = 70$ to $t = 100$, in which the output just fluctuates slightly. Plots from both Set SS and Set TS of Experiment 2 have multiple intervals. We can classify a plot by looking at its final interval. If the final interval consists of slightly fluctuated measurements, the plot is from Set SS. Otherwise, the plot is from Set TS. Due to the presence of multiple intervals, plots in Experiment 2 are closer to plant measurement plots in real situations. Figure 12 represents examples of plots from Set SS and Set

TS used in Experiment 2.

7.4 Experiment 2 - Results

	Group 1	Group 2
Set SS	26	24
Set TS	20	30

Figure 13: *Results of Experiment 2*

Results of Experiment 2 are shown in the confusion matrix in Figure 13. Plots of Set SS are distributed evenly to Group 1 and Group 2 (26 versus 24). It is also the case for plots of Set TS. These results indicate that the CNN method cannot distinguish between images in Set SS and Set TS in this experiment.

This is quite unexpected since the method performs well in Experiment 1 and the two experiments are highly similar. The only difference is that while plots in Experiment 1 have only one interval (steady-state or transient), these in Experiment 2 have multiple. The operation of Pooling Layers may be one possible reason for CNN's poor performance here. As discussed previously, Pooling Layers help to reduce the size and the complexity of input images. During this reduction process, Pooling Layers may fuse information about different intervals of a plot, effectively making plots in Set SS indistinguishable from ones in Set TS.

Results in Experiment 2 show that the method cannot detect whether a plant is at steady-state or not if there are too many data points and intervals. Of course, they cannot detect accurately either if there are too few data points. Therefore, we have to tune the number of data points in plots shown to the CNN appropriately. It is also the reason we use sliding windows in later experiments to generate sub-plots with fewer data points from an original plot with many data points.

Detecting directly from a plot with many data points may be possible with object-detection algorithms, such as R-CNN [22], which can automatically split input images into smaller parts and detect objects within each part. Since such algorithms are complex and this thesis concerned only about introducing CNN as a SSD method and not exploring its full capabilities, we selected the simpler approach of sliding windows and did not investigate this direction.

7.5 Experiment 3 - Methodology

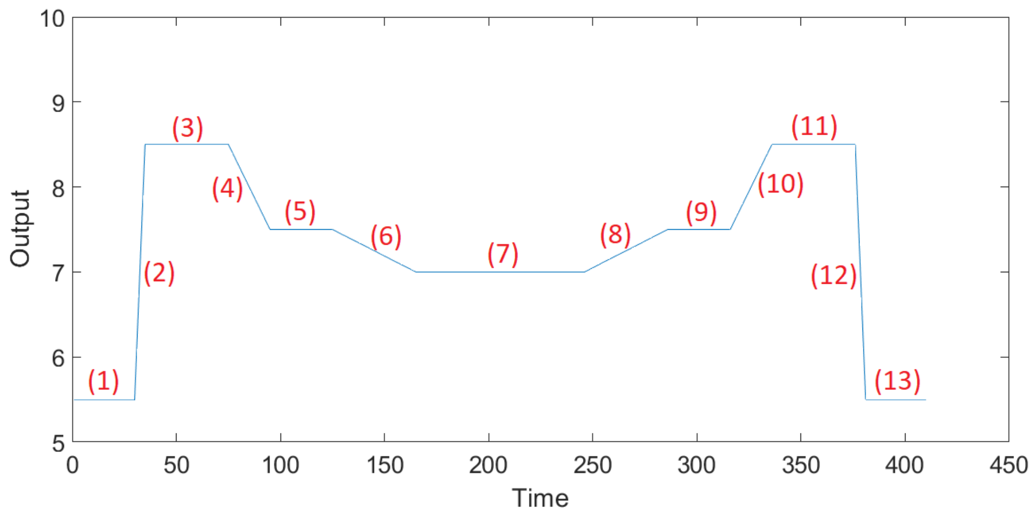


Figure 14: *Plot of plant measurement in Experiment 3 without noise.*

In Experiment 3, we wish to check the performance of the CNN method in situations that usually happen in plant control, which are step response, short ramp response, long ramp response, short steady-state interval, long steady-state interval. We include these scenarios in the 13-intervals measurement plot of Experiment 3, Figure 14, in which:

- Step response intervals: segments 2, 12
- Short ramp response intervals: segments 4, 10
- Long ramp response intervals: segments 6, 8
- Short steady-state intervals: segments 1, 5, 9, 13
- Long steady-state intervals: segments 3, 7, 11

However, this plot in Figure 14 is not presented directly to the CNN. According to the experimental procedure in Figure 15, Gaussian noise is added to the measurements first, resulting in a noisy plot represented by Figure 17. A sliding window with a window length of 20 and a step length of 1 slides through the noisy plot to generate sub-plots that are presented to the CNN.

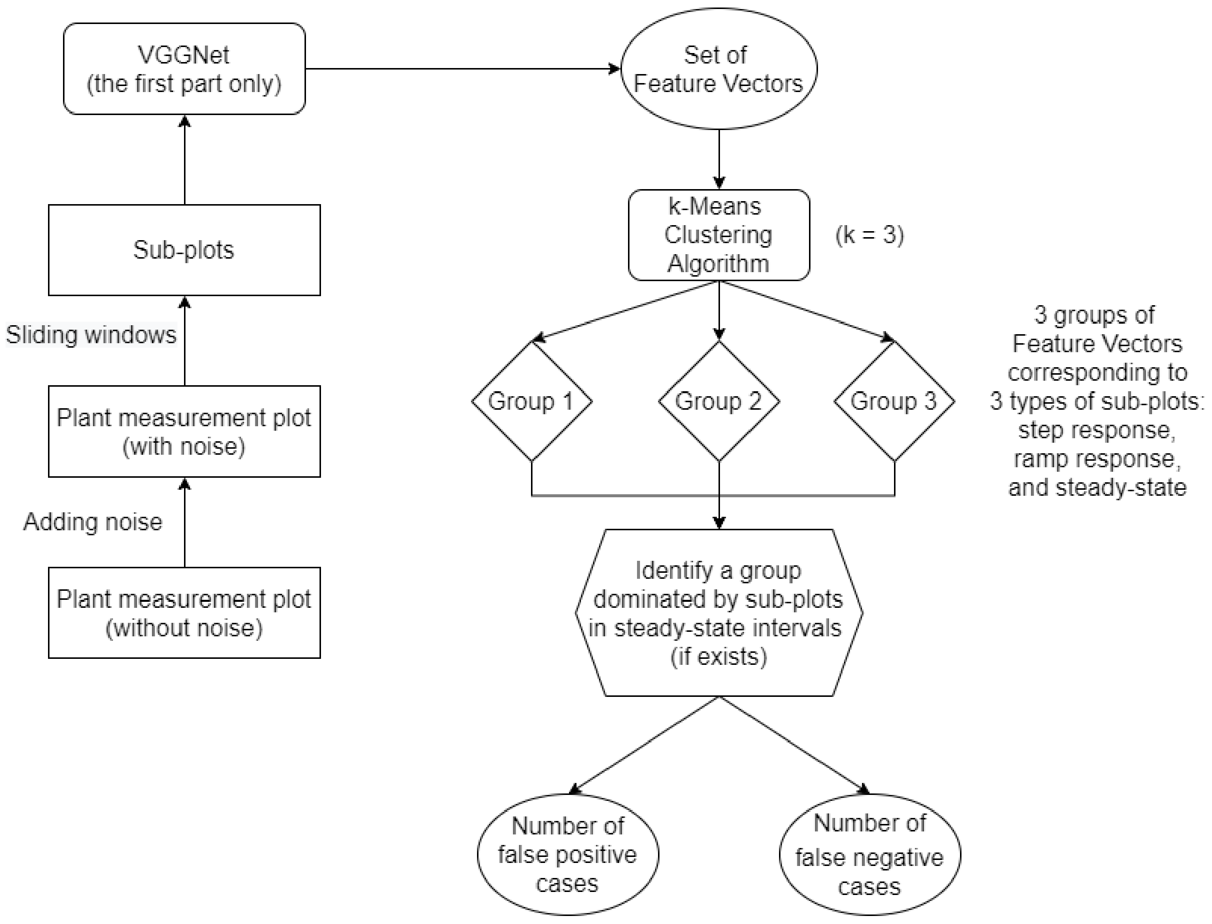


Figure 15: *Procedure of Experiment 3*

Figure 16 shows an example of generating subplots from a plot using a sliding window with a window length of 6 and a step length of 1. It should be noted that we take a plot with no noise as an example to illustrate how sliding windows work in Figure 16. That is for visualization and explaining purposes only. In the experiment, the sliding window slides through the noisy plot in Figure 17. Initially, the window was in the position of the red square and covered a time interval from 25 to 30. 6 measurements during this interval y_t $(t \in [25; 30])$ are taken and used for generating the first sub-plot. However, each measurement must be processed before the transfer as following:

$$y_t = y_t - (\frac{\sum_{t=25}^{30} y_t}{6} - \frac{5+10}{2})$$

The purpose of this signal processing is to center the measurements in the subplot. Not only is the output modified but also the time. It can be seen that the time axis in the subplot does not start from the start of the interval covered by the red window, which is $t = 25$. It starts from $t = 1$ instead. These modifications help to eliminate the possibility that CNN incorrectly focuses on details in the axes instead of the measurement signals. After the subplot from the red window is generated, the window moves one step forward. It is now represented by the green square. Another subplot is generated from the green window before it moves forward again. The process repeats until the window reaches the final data point in the original plot.

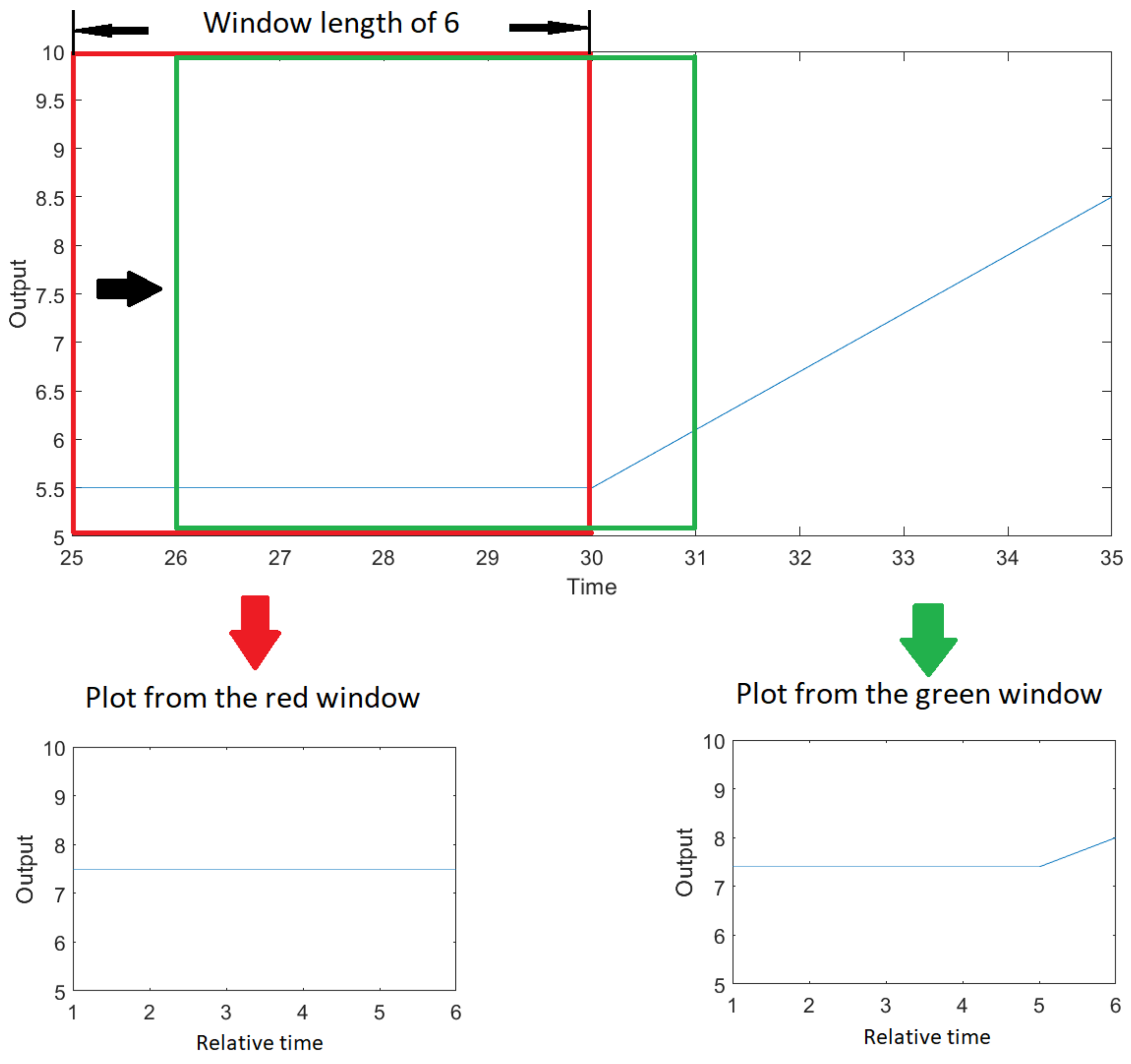


Figure 16: *An example of subplots from a plot using sliding windows*

Since there are 410 data-points in Figure 17 and the sliding window has a length of 20 and a step of 1, there are 391 sub-plots generated. Details about these sub-plots are represented in Table 8.

Table 8: *Details about sub-plots generated by the sliding window in Experiment 3*

x-axis limit in each plot (time axis)	0-100
y-axis limit in each plot (output axis)	5-10
Number of datapoints in each plot (T)	20 (Equal to the length of the sliding window)

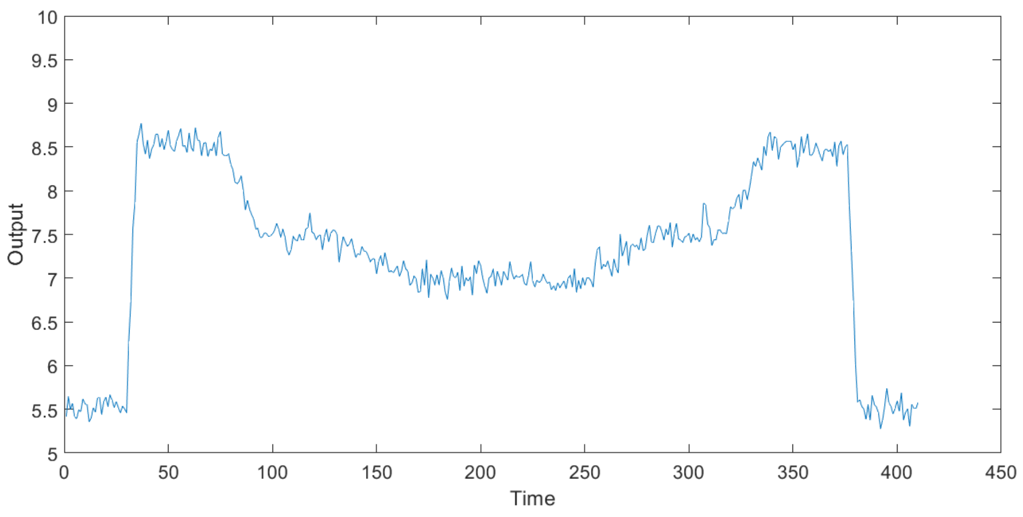
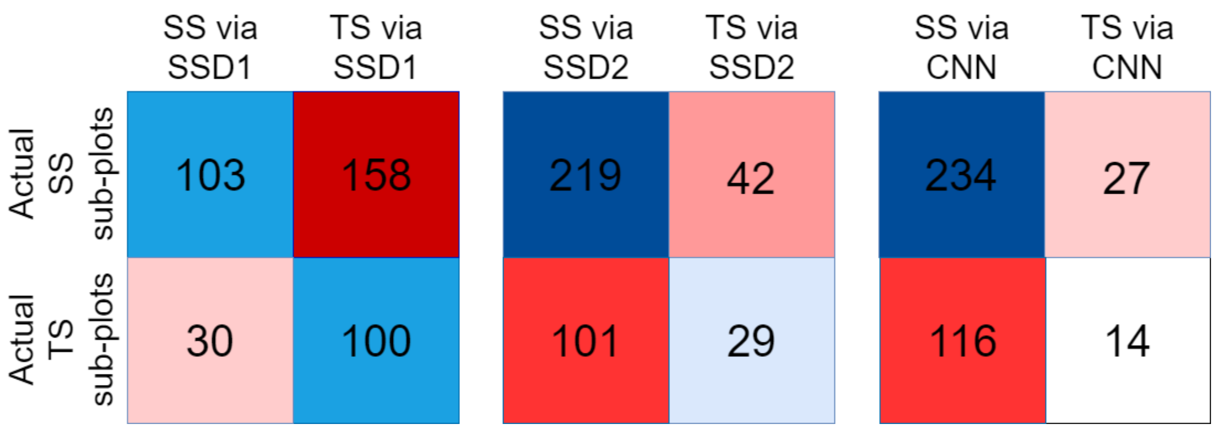


Figure 17: *Plot of plant measurement in Experiment 3 with white noise.*

The feature vectors from the sub-plots are partitioned into 3 groups (Groups 1, 2, and 3) using the k-Means algorithm. We select $k = 3$ as we (conservatively) consider there are three types of sub-plots. These types are sub-plots in step response intervals, ramp response intervals, and steady-state intervals, respectively. Then we identify a group that contains mostly sub-plots from steady-state intervals (if no such group exists, it means the CNN method has a poor performance). We consider this group as the steady-state group, and other groups as transient. Since we generated the data synthetically and know which data points are static or transient, we check how many sub-plots in the steady-state/transient groups are actually in the steady-state/transient intervals to determine the number of false positive/false negative cases. We use these numbers to evaluate the performance of the CNN method relative to SSD1 and SSD2 methods. We also investigate the distribution pattern of the errors made by each method to obtain a better understanding about their behaviors, such as in which conditions that they are more likely to make mistakes in the plant condition detection

7.6 Experiment 3 - Results



	SS via SSD1	TS via SSD1	SS via SSD2	TS via SSD2	SS via CNN	TS via CNN
Actual SS sub-plots	103	158	219	42	234	27
Actual TS sub-plots	30	100	101	29	116	14

Figure 18: *Results of Experiment 3*

Results of Experiment 3 are represented in the confusion matrices in Figure 18. From left to right, the matrices show the best performance achieved by SSD1, SSD2, and CNN methods, respectively. Each column in a matrix represents the category that the corresponding SSD method assigned to sub-plots in that column, while each row represents the actual category of sub-plots in that row. For example, from the bottom-left cell of the first matrix, we know that the SSD1 method, which relies on filtered measurements and variances comparison for steady state detection, mistakenly classified 30 TS sub-plots as SS. The top-right cell of the same matrix tells us that there were 158 SS sub-plots classified as TS by SSD1. We can also calculate the total number of errors that SSD1 made by taking the sum of numbers in the two red cells, which is equal to $30 + 158 = 188$. The total number of sub-plots that SSD1 classified correctly can be computed by taking the sum of numbers in the two blue cells - the top-left and bottom-right ones - which is equal to $103 + 100 = 203$. It is easy to recognize that the darker a cell gets, the greater the number inside is.

SSD1 actually has the worst performance among the three methods. The other two methods have similar performances. SSD2, which determines whether the process is stationary or not by comparing actual measurements to a hypothetical stationary mean, has 101 false positive, 42 false negative, and 143 total errors. These numbers in the case of the CNN method are 116, 27, and 143, respectively.

To evaluate the methods further, we have visualized the errors that each method makes in the without-noise time-output plots as in Figure 19. These error instances are represented as orange dots in the plots. At first glance, the distribution of errors of SSD1 is different from that of SSD2 while being similar to CNN. The errors in SSD2 are scattered all over the plot, while they are more concentrated in the other two methods. However, the error distributions of SSD1 and CNN are also different from each other. While the CNN's errors are present in almost all intervals of the plots, they usually appear only at the beginning of each interval (except for intervals 6, 8, and 10). In contrast, the SSD1's errors are present only in a few intervals (e.g., intervals 3, 5, and 8), but the errors appear frequently and consistently throughout these intervals. It is as if the SSD1 method misclassified the whole of each of these intervals.

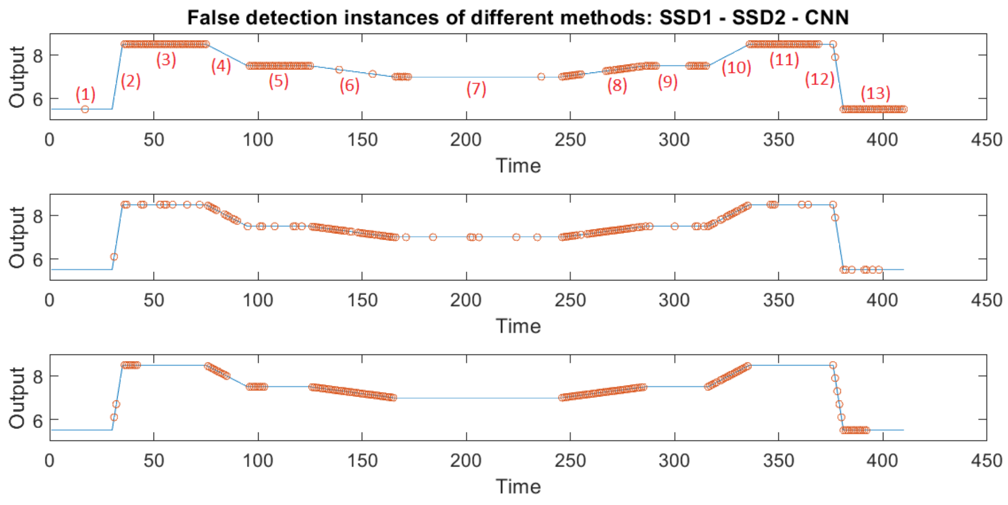


Figure 19: *False detection instances of different methods in Experiment 3 (from top to bottom: SSD1 - SSD2 - CNN)*

The error distribution of SSD1 suggests that it responds slowly to the operating profile changes. It incorrectly classifies intervals 3 and 5 as transient, seemingly because their previous intervals, 2 and 4, are responses with a large magnitude of measurement change. In contrast, it determines that interval 7 is at steady-state. In prior to 7 is a long-ramp response interval, which has a magnitude of measurement change highly similar to that of steady-state ones.

The errors at the beginning of each interval in the CNN method are expected because it is possible to acknowledge an interval change only after it has happened for a while. These also show that the CNN method is highly consistent: it has a low probability to trigger RTO during a transient interval. This consistency of the CNN is an advantage over the SSD2 method although, in general, both steady-state detection methods have the same accuracy.

The error distributions have also shown that all three methods incorrectly classify long-ramp intervals as steady-state intervals. This is expected because in long ramps, the system state is changing slowly, and thus the system can almost be seen as being in a quasi-steady state. However, this type of error probably does not affect RTO performance significantly since it is usually reasonable to treat a slowly changing system as being at steady-state.

7.7 Experiment 4 - Methodology

The procedure of Experiment 4 is similar to that of Experiment 3, except that the process noise now follows a heavy-tailed distribution instead of the normal distribution. The purpose of Experiment 4 is to investigate how the CNN method performs when the probability of having outliers in measurements is higher.

We select the Stable distribution, also called the α-stable distribution, as our heavy-tailed distribution. Noises that follows the Stable distribution are called α-stable noises. The Stable distribution does not have an explicit probability density function. Thus, we do not include the probability function here. For more details about the Stable distribution, please refer to [23]. Values of the parameters of the distribution are shown in Table 9.

Figure 20 is the plot of the measurements against time after noise following the described Stable distribution has been added.

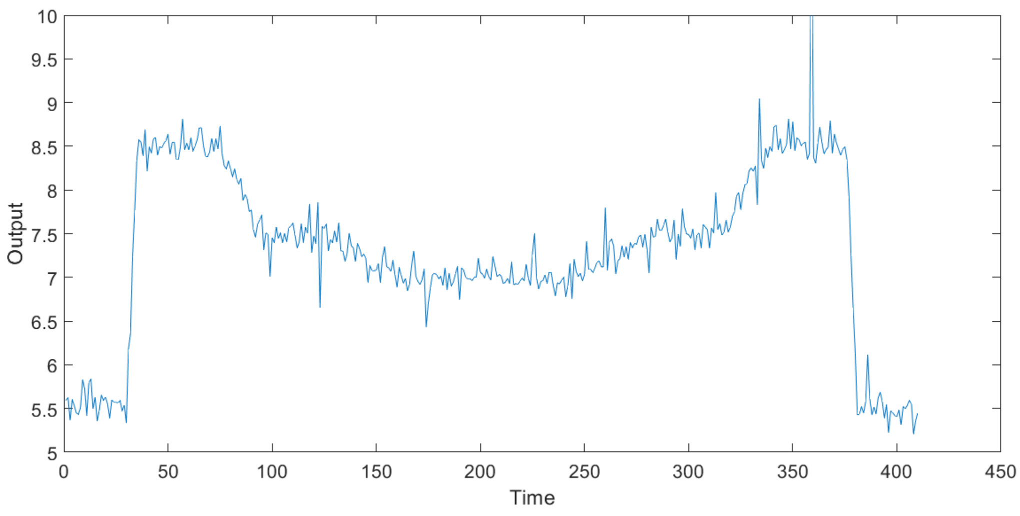
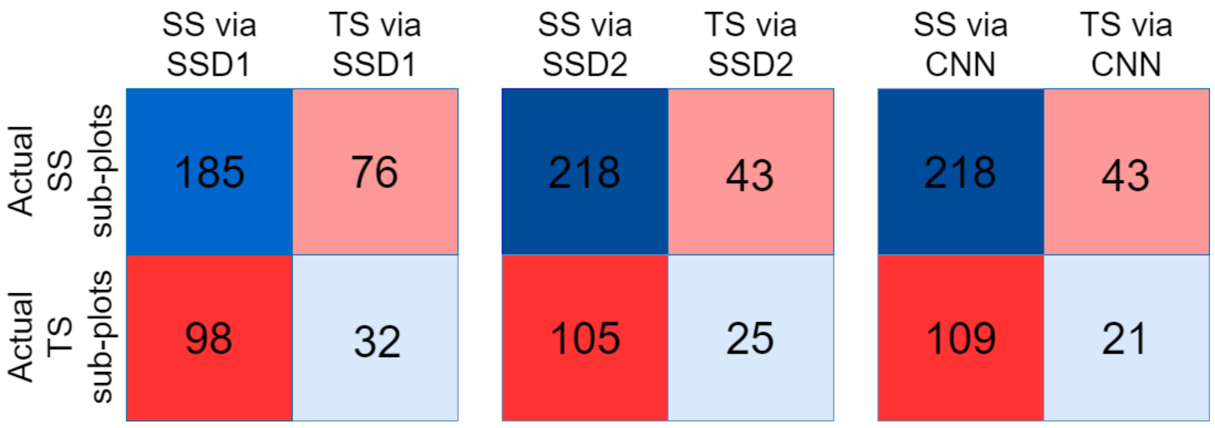


Figure 20: *Plot of plant measurement in Experiment 4 with α-stable noise.*

Table 9: *Values of the parameters of the Stable distribution used in Experiment 4*

α	1.5
β, δ	0
γ	0.0725

7.8 Experiment 4 - Results



	SS via SSD1	TS via SSD1	SS via SSD2	TS via SSD2	SS via CNN	TS via CNN
Actual SS sub-plots	185	76	218	43	218	43
Actual TS sub-plots	98	32	105	25	109	21

Figure 21: *Results of Experiment 4*

The behaviors of the three methods in Experiment 4 are similar to those in Experiment 3. The error statistics and error distribution of the three methods are represented in Figure 21 and Figure 22, respectively.

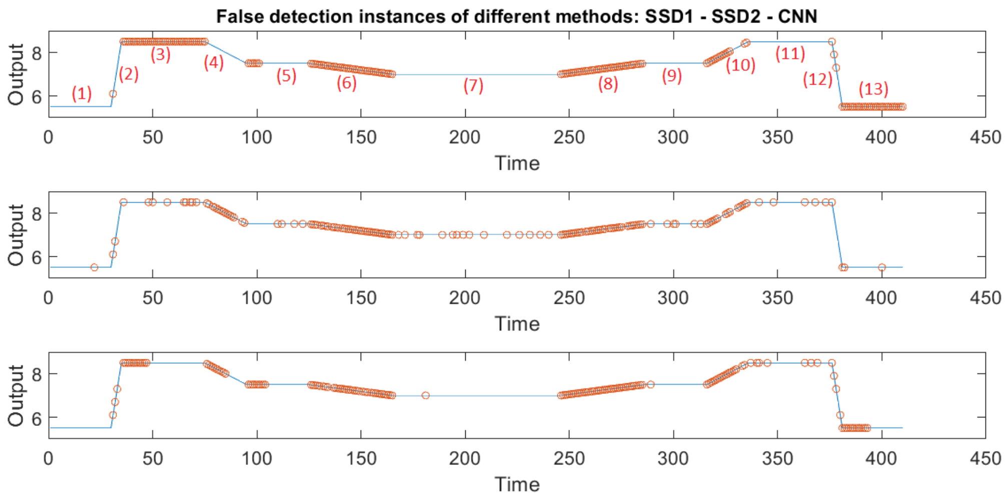


Figure 22: *False detection instances of different methods in Experiment 4 (from top to bottom: SSD1 - SSD2 - CNN)*

SSD1 is still the method with the worst performance, with 174 errors. CNN performs slightly worse than SSD2 did, but the difference is small (152 versus

148 errors). The change of noise distribution from normal distribution to Stable distribution seemingly does not affect the performance of SSD2 and CNN, although SSD1 performs slightly better in the latter case. The distributions of errors of the three methods remain unchanged. The CNN method retains its advantage of high consistency over the SSD2 method.

7.9 Experiment 5 - Methodology

The procedure of Experiment 5 is similar to that of Experiment 3, except that we have colored noise in the process instead of the white noise. Through this experiment, we wish to evaluate the performance of the CNN method in systems with fast measurement, i.e. high sampling frequency, in which Colored noise may exist [24].

To create the colored noise, we generate a sequence S of T random numbers $S = \{r_1, r_2, ..., r_T\}$ from the normal distribution $\mathcal{N}(0, 0.1)$. Then we create a filtered sequence S_f with T elements to represent the colored noise realizations. This filtered sequence is also the sequence of colored noise to be added to the original, without-noise plot in Figure 14 to result in the plot represented in Figure 23. The elements $r_{f\,(t)}$ $(t \in [1, T])$ of the filtered sequence are calculated from the elements of the original sequence r_t $(t \in [1, T])$ as following:

$$r_{f\,(t)} = 0.1 \cdot r_{f\,(t-1)} + 0.9 \cdot r_t \quad (with \quad r_{f\,(0)} = r_0)$$

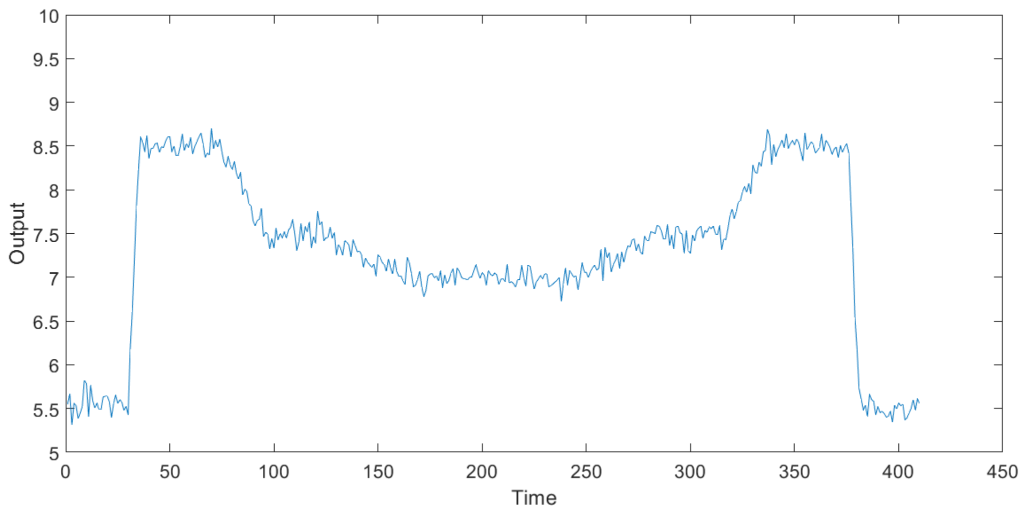
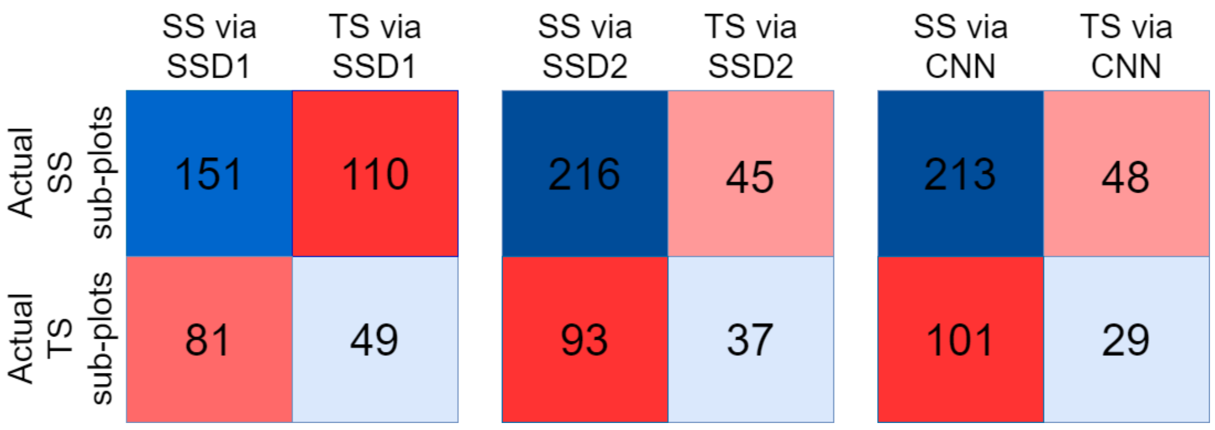


Figure 23: *Plot of plant measurement in Experiment 5 with colored noise.*

7.10 Experiment 5 - Results

Results from Experiment 5, as shown in Figure 24 and Figure 25, are not different from those from Experiments 3 and 4. The change of noise distribution again does not seem to affect the performance of the CNN method.



	SS via SSD1	TS via SSD1	SS via SSD2	TS via SSD2	SS via CNN	TS via CNN
Actual SS sub-plots	151	110	216	45	213	48
Actual TS sub-plots	81	49	93	37	101	29

Figure 24: *Results of Experiment 5*

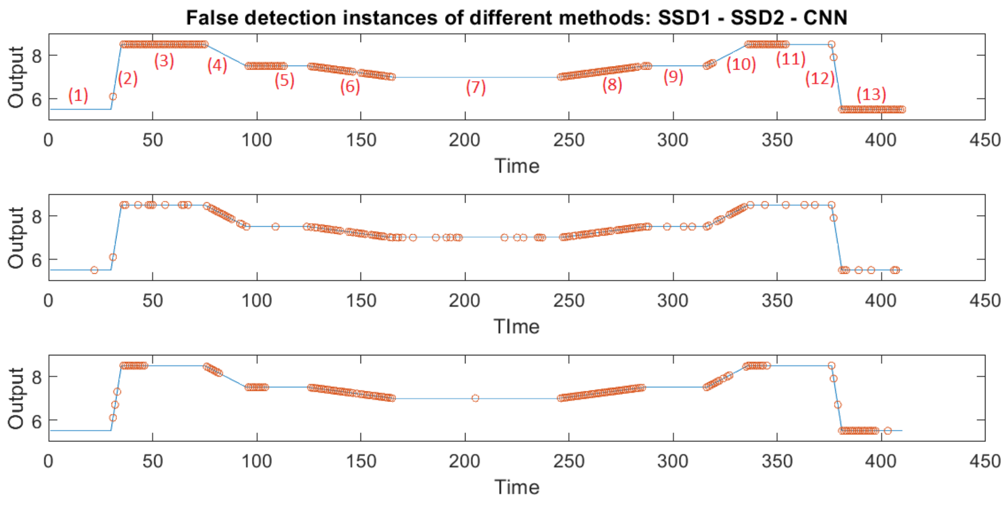


Figure 25: *False detection instances of different methods in Experiment 5 (from top to bottom: SSD1 - SSD2 - CNN)*

8 Limitations and Further Study

We have studied CNN as a potential steady-state detection method. We have shown that the CNN method has performance comparable to that of other numerical SSD methods and seems to be robust to different types of noise distributions. It also has an advantage over the SSD2 method: the high consistency in its detection, which may positively affect the performance of SRTO systems. This hypothesis should be validated and studied quantitatively in a complete system of SRTO in the future.

One important implementation aspect we have not considered is the computational time required for the CNN method. In this study, we implement the method on GPUs (graphic processing units), which may not be readily available in computer systems of chemical plants. Thus, we have no base to estimate the required computational time when running on non-GPU systems. However, if the sampling interval used in the SSD is on a scale of minutes, the computational time is unlikely to cause any problems. It is also interesting to see how the method performs in terms of accuracy and computational time when other popular Convolutional Neural Networks are used instead of the VGGNet.

We did not investigate in details how the method's parameters, which are the scale of the y-axis and the sliding window length, impact its performance. The impacts of these parameters on the performance of the CNN method should be understood better and have a systematic tuning approach. We should also evaluate the method's performance further on data from real-life systems.

Finally, despite the way that CNN works is very intuitive and its performance is comparable to that of other SSD methods, we acknowledge that its implementation is not as easy. Moreover, coding a CNN is also more difficult than the other methods and may need a much more specialized person to do it.

Part IV

Conclusion

In this thesis, we have developed two new applications of Machine Learning in steady-state real-time optimization.

We have shown in the first application how powerful Genetic Programming is as a SOC variables search method. It can find SOC variables that give small economic losses over a large operating region. Interestingly, as GP can generate many SOC variables with similar performances for a process, we may be able to study more about SOC from these variables.

In the second application, we have demonstrated that Convolutional Neural Networks can be a potential alternative to statistical SSD methods. This new CNN-based method has an intuitive working mechanism and comparable performance to other SSD methods. Moreover, it has shown a higher consistency in its detections, which may be advantageous to the performance of SRTO.

Despite the results so far being promising, the thesis presents only proof-of-concept studies, where both applications are applied to bechmark problems. There are many aspects of both methods that need to be further explored and understood. For example, can GP GP find dynamic SOC variables? Or how does the CNN method perform on real-life data? Future work should concentrate on addressing these questions.

9 References

[1] A. Mhamdi et al. Thermal Desalination Processes - Volume I, On-line Optimization of MSF Desalination Plants. EOLSS Publishers Co. Ltd, 2000. ISBN: 978-1-84826-425-0.

[2] S. Skogestad. “Plantwide control: The search for the self-optimizing control structure”. In: *Journal of Process Control* 10 (2000), pp. 487–507.

[3] J. E. A. Graciano et al. “Integrating self-optimizing control and real-time optimization using zone control MPC”. In: *Journal of Process Control* 34 (2015), pp. 35–48.

[4] J. Jaeschke, Y. Cao, and V. Kariwala. “Self-optimizing control - A survey”. In: *Annual Reviews in Control* (2017). DOI: `http://dx.doi.org/10.1016/j.arcontrol.2017.03.001`.

[5] V. Alstad and S. Skogestad. “Null Space Method for Selecting Optimal Measurement Combinations as Controlled Variables”. In: *Ind. Eng. Chem. Res.* 46 (3 2007). DOI: `https://doi.org/10.1021/ie060285+`.

[6] J. Jaeschke and S. Skogestad. “Optimal controlled variables for polynomial systems”. In: *Journal of Process Control* 22 (1 2012).

[7] L. Ye, Y. Cao, and Z. Song. “Approximating necessary conditions of optimality as controlled variables”. In: *Ind. Eng. Chem. Res.* 52 (2 2013).

[8] L. Ye, Y. Cao, and X. Yuan. “Global approximation of self-optimizing controlled variables with average loss minimization”. In: *Ind. Eng. Chem. Res.* 54 (2015).

[9] J. Jaeschke and S. Skogestad. “Optimal operation by controlling the gradient to zero”. In: *Proceedings IFAC orld congress, Milano, Italy* (2011).

[10] R. Poli, W. Langdon, and N. McPhee. A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd, 2008. ISBN: 978-1409200734.

[11] E. Real et al. “AutoML-Zero: Evolving Machine Learning Algorithms From Scratch”. In: *arXiv:2003.03384* (2020).

[12] S. Skogestad. “Near-optimal operation by self-optimizing control: from process control to marathon running and business systems”. In: *Computers and Chemical Engineering* 29 (2004).

[13] J. Chen. "Logarithmic mean: Chen's approximation or explicit solution?" In: *Computers & Chemical Engineering* 120 (2019).

[14] F. Petterson. "Heat exchanger network design using geometric mean temperature difference". In: *Computers & Chemical Engineering* 32 (2008).

[15] J. Jaeschke and S. Skogestad. "Optimal operation of heat exchanger networks with stream split: Only temperature measurements are required". In: *Computers & Chemical Engineering* 70 (2014).

[16] S. Aaltvedt. *Optimal Operation of Parallel Heat Exchanger Networks.* Master's thesis at the Norwegian University of Science and Technology (NTNU), 2013.

[17] "A steady-state detection (SSD) algorithm to detect non-stationary drifts in processes". In: *Journal of Process Control* 23 (2013).

[18] "An efficient method for on-line identification of steady state". In: *Journal of Process Control* 5 (1995).

[19] R. R. Rhinehart. "Automated steady and transient state identification in noisy processes". In: *2013 American Control Conference* (2013). DOI: `10.1109/ACC.2013.6580530`.

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

[21] J. D. Kelleher. Fundamentals of Machine Learning for Predictive Data Analytics (1st edition). The MIT Press, 2015. ISBN: 978-0262029445.

[22] R. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: (2013). DOI: `arXiv:1311.2524`.

[23] URL: `https://www.mathworks.com/help/stats/stable-distribution.html`.

[24] A. Bryson and D. Johansen. "Linear filtering for time-varying systems using measurements containing colored noise". In: *IEEE Transactions on Automatic Control* 10 (1 1965).