8-2021

Surrogate Optimization Model for an Integrated Regenerative Methanol Transcritical Cycle

Yili Zhang
Utah State University

SURROGATE OPTIMIZATION MODEL FOR AN INTEGRATED REGENERATIVE METHANOL TRANSCRITICAL CYCLE

by

Yili Zhang

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

of

MASTER OF SCIENCE

in

Mechanical Engineering

Approved:

Hailei Wang, Ph.D. Major Professor	Geordie Richards, Ph.D. Committee Member
Barton Smith, Ph.D. Committee Member	Jia Zhao, Ph.D. Committee Member

D. Richard Cutler, Ph.D.
Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2021

ABSTRACT

Surrogate Optimization Model for an Integrated
Regenerative Methanol Transcritical Cycle
by
Yili Zhang, Master of Science
Utah State University, 2021

Major Professor: Hailei Wang, Ph.D.
Department: Mechanical and Aerospace Engineering

To explore the potential levelized cost of energy (LCOE) improvements on a small modular reactor (SMR)-based power cycle, an integrated regenerative methanol transcritical cycle with up to seven free design parameters was designed and simulated with the Python & Coolprop software. The physics-based model is capable of simulating the cycle functionalities by taking design parameters, and then producing corresponding performances, such as: LCOE, first-law efficiency etc. This procedure takes a few minutes given a set of viable design parameters. In order to compare the performances between different types of cycles, the corresponding optimized results are needed. However, the global optimization of the physics-based model is very time-consuming, given the high degree of freedoms and the complexity of the system.

This thesis provides a research review on a machine learning-based surrogate model in replacing the physics-based model, so that the global optimization can take place faster. Candidate surrogate models based on different algorithms are built and analyzed by their validation, test accuracy, r2 score and relative errors values of the model LCOE, first-law efficiency and penalty (violation level of the system). The last chosen model is further optimized in terms of its structure and node hyper-parameters. With the structurally and parametric optimized surrogate model being incorporated, different global optimizers are

used and analyzed. As the result, the optimized design parameters from the surrogate-optimizer model are fed into the physics-based model, and their corresponding results are compared with the baseline optimized results of the physics-based model.

In conclusion, the study reveals that the Multilayer Perceptron (MLP) networks with two hidden layers gives the best prediction performance, and therefore they are chosen as the surrogate model. In addition, four global optimizers, namely the basinhopping, the differential evolution, the dual annealing and the fmin, are working well along with the chosen surrogate model. They integrated surrogate-optimizer model is capable of finding the optimized LCOE as well as the corresponding design parameters. In comparison with the baseline optimized LCOE, the relative error is less than 3.5%, and this searching procedure completed within 30 minutes.

(185 pages)

PUBLIC ABSTRACT

Surrogate Optimization Model for an Integrated
Regenerative Methanol Transcritical Cycle
Yili Zhang

In order to reduce the cost ($) per megawatts hour ($MWh$) of electrical energy generated by a nuclear power cycle with a novel small modular reactor (SMR), a new SMR-based nuclear power cycle with Methanol as working fluid was designed. It was built virtually with the Python & Coolprop software based on all components' physical properties, and it is therefore called the physics-based model. The physics-based model would require seven user-defined values as input for the seven free design parameters, respectively. The physics-based model outcomes include LCOE (the cost per megawatts hour of electrical energy generated by the cycle), the first-law efficiency of the power cycle (the ratio between the net power out of the power cycle and the net thermal energy into the power cycle), and the penalty (severity of system violation with the given design parameters values). In order to compare between different power cycles, the corresponding optimized LCOE are needed. However, in order to find the design parameters that optimize the system LCOE, it can take up to three days with the physics-based model, because the physics-based model is highly complex and it takes thousands of iterations averagely to the optimize the power system.

In confronting the time complexity issue in optimization, the study in this paper explores the viability of replacing the physics-based model with a machine learning-based surrogate model. During the optimization procedure, the machine learning-based surrogate model is expected to accelerate process of finding the corresponding outcomes, and thus to save time. Candidate surrogate models are built and analyzed in terms of their prediction accuracy. The last chosen model is further optimized in terms of its structure and hyper-parameters. With the structurally and parametric optimized surrogate model being

incorporated, different global optimizers are used and analyzed. As the result, the optimized design parameters from the surrogate-optimizer model are fed into the physics-based model, and their corresponding results are compared with the baseline optimized results of the physics-based model.

In conclusion, the study reveals that the Multilayer Perceptron (MLP) networks with two hidden layers gives the best prediction performance, and therefore they are chosen as the surrogate model. In addition, four global optimizers, namely the basinhopping, the differential evolution, the dual annealing and the fmin, are working well along with the chosen surrogate model. They integrated surrogate-optimizer model is capable of finding the optimized LCOE as well as the corresponding design parameters. In comparison with the baseline optimized LCOE, the relative error is less than 3.5%, and this searching procedure completed within 30 minutes.

DEDICATION

To my mother, Junling Pang. Thanks for your unconditional love and support in both my life and career. I love you forever.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Hailei Wang. Without the recognition, instructions and financial support from him since the undergraduate years, I would not have had chance to accumulate sufficient research experiences, which lead me to this project. Both my knowledge and academic professionalism are strengthened from his patient and professional trainings.

I would also like to thank Dr. Geordie Richards and Jacob Bryan. I am grateful for the valuable contribution and suggestions provided by them in the cooperation. Without the contribution from them, the project would not have been so fruitful. I want to particularly thank rest of my thesis committee, Dr. Barton Smith for agreeing to be my committee member during his sabbatical, Dr. Geordie Richards for the valuable teachings and advice given to me, and Dr. Zhao for providing me valuable views from the perspective of other fields.

Besides my advisor and committee members, I would like to thank NuScale Power and the manager Derick Botha for the valuable advice and financial support provided for the NuScale Project Phase II, which formed the foundation of my thesis project.

Last but not the least, I would also like to thank Nuclear Regulatory Commission (NRC) for its financial support through the Award No. 31310019M0014.

Yili Zhang

CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

LCOE	levelized cost of energy
SMR	small modular reactor
MLP	multilayer perceptron
MWh	megawatts hour
GA	genetic algorithms
ML	Machine Learning
ANN	artificial neural network
ORC	organic Rankine cycle
LSTM	long short term memory
MSE	mean squared error
DNN	deep neural network
PCA	principle component analysis
CPU	central processing unit
SHGO	simplicial homology global optimisation
TGO	topographical global optimisation
CSA	classical simulated annealing
FSA	fast simulated annealing
BLPP	bilevel programming problems
DPP	data pre-processing
SMOTE	synthetic minority over-sampling technique
RUS	random undersampling
MLF	multi-layer feed-forward
ADAM	adaptive moment estimation
BCE	binary cross entropy
RF	random forest
MAP	maximum a posterior

KNN	k nearest neighbor
LR	logistic regression
SVM	support vector machine
ResNets	residual networks
CNN	convolutional neural network
FN	false negative
FP	false positive
TP	true positive
TN	true negative
NB	naive bayes
RAE	relative absolute error
BH	basin-hoppin
w/	with
LE	linear error
DA	dual annealing

CHAPTER 1

INTRODUCTION

1.1 Background & Overview

Optimization is important for thermodynamic and power system designs. Common objective functions of power cycle optimizations include: net power out, first and second law efficiencies and Levelized Cost of Energy (LCOE) [14–16]. The key on system optimization is to find the design parameters of the system that either minimize or maximize the objective functions. However, given the non-linearity of the equations of state for thermodynamic modeling and temperature-dependent thermophysical properties of the working fluids, large system optimization can be time-consuming and slow to converge.

From the optimizations of our simulation model, it has been observed that global/semi-global optimization algorithms, such as Genetic Algorithms (GA) and Patternsearch [17], can realize high accuracy within reasonable amount of time when the number of the design parameters for a system is small (≤ 4). However, when the number is large, global/semi-global optimization algorithms normally fail to converge.

Traditionally, the parametric study has been commonly used as an optimization method for power systems. Specifically, each design parameter is explored within its design range while keeping the other parameters unchanged at their baseline values. As the results, the parametric study can provide insights into every individual design parameter's impacts on the system's performance index; however, combination of the best free design parameters for a system is not necessarily the one that optimizes the system.

During the past decade, with the prevalence of the Machine Learning (ML) technique, more and more people has been incorporating ML into their domain field in solving some previously unsolvable problems. Similarly, in optimizing thermodynamic and power systems, using surrogate models based on ML algorithms in supplementing the experimental

or simulation models became more and more popular, as appropriate surrogate models can substantially reduce the computational power required while realizing high searching accuracy [18].

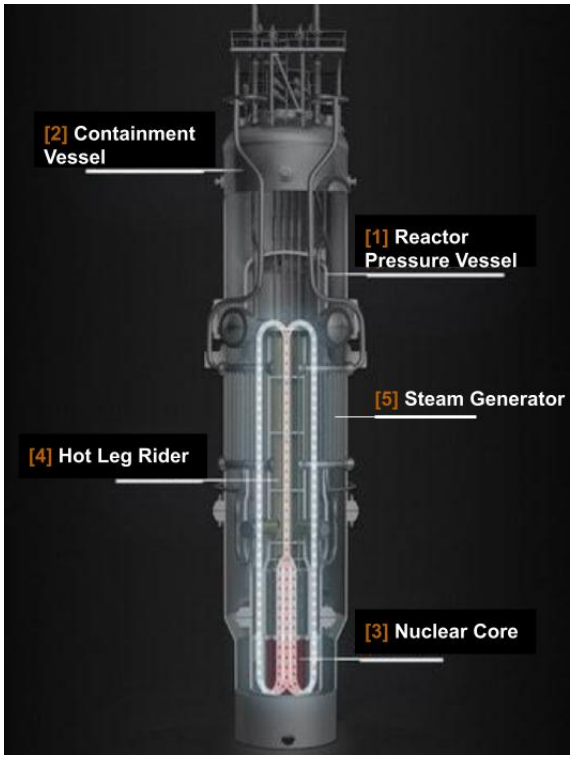


Fig. 1.1: NuScale Power Module (NPM) System. [1].

This study explores using a ML-based surrogate model to replace the physics-based simulation model of an Integrated Regenerative Methanol Transcritical Cycle with a small modular reactor (SMR) using natural circulation of the coolant. The primary cycle is shown in Fig. 1.1, while the integration of the primary cycle and the working cycle is shown in Fig. 1.2, respectively. Once the surrogate model's performances match closely with the cycle simulation model developed using Python and REFPROP, the surrogate model will be coupled with an optimizer to find the design parameters that minimizes the Levelized Cost of Energy (LCOE) and the penalty of the Cycle.

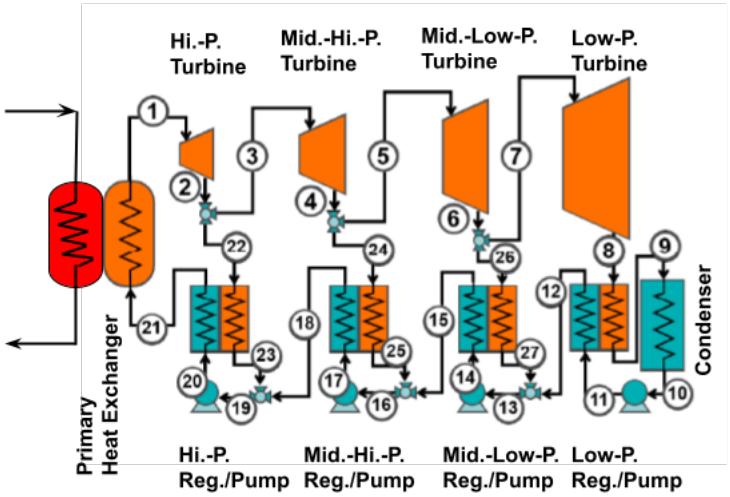


Fig. 1.2: Diagram of Integrated Regenerative Methanol Transcritical Cycle with the SMR.

1.2 Thesis Problem

The integrated regenerative methanol transcritical cycle consists of the primary cycle and the secondary cycle. The primary cycle, as shown in Fig. 1.1, consists of a small reactor core housed with other primary system components in an integral reactor pressure vessel and surrounded by a steel containment vessel, which is immersed in a large pool of water. When it functions, the primary reactor coolant is circulated upward through the reactor core and the heated water is transported upward through the hot riser tube. The coolant flow is turned downward at the pressurizer plate and flows over the shell side of the steam generator, where it is cooled by conduction of heat to the secondary coolant (happens in the Primary Heat Exchanger) and continues to flow downward until its direction is again reversed at the lower reactor vessel head and turned upward back into the core. The coolant circulation is maintained entirely by natural buoyancy forces of the lower density heated water exiting the reactor core and the higher density cooled water exiting the steam generator [19]. The secondary cycle, as shown in Fig. 1.2 is a regenerative methanol transcritical turbine–generator system. It consists of four expansion turbines, four regenerators, one secondary heat exchanger, one condenser, four pumps and three splitting valves. When it functions, as indicated in Fig. 1.3, the feedback methanol working fluid is pumped into the high-pressure regenerator and the secondary heat exchanger, where it is

heated from the sub-cooled state to the transcritical state; then it is circulated to the entire turbine–generator system to generate power.

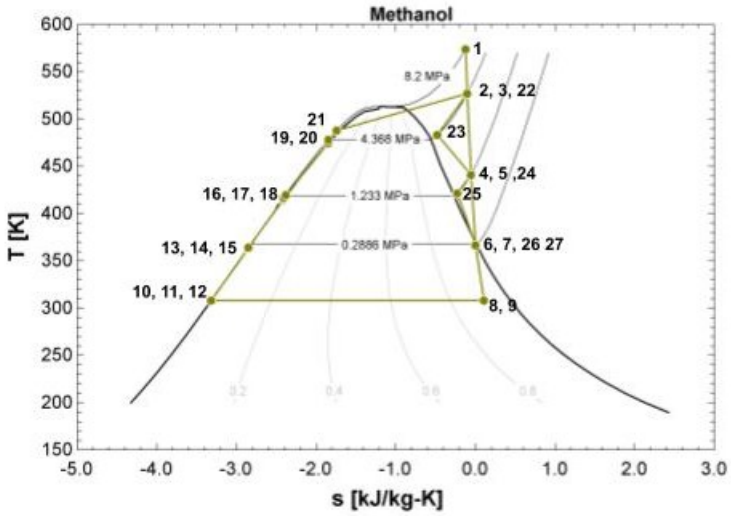


Fig. 1.3: T-S Diagram of the Integrated Regenerative Methanol Transcritical Cycle.

Table 1.1: Regenerative Transcritical Methanol Cycle Baseline Design Points.

Symbol	**Name**	**Value**	**Unit**
P_{max}	Power Cycle Maximum Pressure	8.2	MPa
Pr_1	High-pressure Ratio	0.5327	-
Pr_2	Middle-pressure Ratio	0.2823	-
Pr_3	Low-pressure Ratio	0.2340	-
f_1	High-pressure Mass Fraction	0.2659	-
f_2	Middle-pressure Mass Fraction	0.1583	-
f_3	Low-pressure Mass Fraction	0.1301	-

For testing the performance of the Integrated Regenerative Methanol Transcritical Cycle, a physics-based simulation model is built. In the model, seven design parameters are required and the model's baseline values of the design parameters are summarized in Tab. 1.1. By referring to Fig. 1.2, it can be seen that the power cycle's maximum pressure

Table 1.2: Regenerative Transcritical Methanol Cycle Baseline Results.

Symbol	**Name**	**Value**	**Unit**
eta_I	First Law Efficiency	32.76	%
LCOE	Levelized Cost of Energy	89	$\frac{\$}{MWh}$
penalty	penalty	0	-

(P_{max}) is the pressure at position 1 in Fig. 1.2. In addition, the high-pressure ratio (Pr_1) is calculated from the ratio between the pressure at position 2 (turbine outlet pressure) and the pressure at position 1 (turbine inlet pressure). Similarly, the middle-pressure ratio (Pr_2) is calculated from the ratio between the pressure at position 4 and the pressure at position 3, and the low-pressure ratio (Pr_3) is calculated from the ratio between the pressure at position 6 and the pressure at position 5, and the high-pressure mass fraction (f_1) is calculated from the fraction between the mass flow rate at position 22 (split mass flow rate) and the mass flow rate at position 2 (turbine outlet mass flow rate). Similarly, the middle-pressure mass fraction (f_2) is calculated from the fraction between the mass flow rate at position 24 and the mass flow rate at position 4, and the low-pressure mass fraction (f_3) is calculated from the fraction between the mass flow rate at position 26 and the mass flow rate at position 6. With the baseline design parameters as inputs, the physics-based model's simulating outputs are summarized in Tab. 1.2, which includes the first law efficiency, the levelized cost of energy (LCOE) and the penalty, where the first law efficiency is calculated from:

$$\eta_I = \frac{W_{net,out}}{Q_{in}} \tag{1.1}$$

and the LCOE is calculated from the ratio between the sum of costs ($) of the apparatus over lifetime and the sum of electrical energy (MWh) the apparatus produced over lifetime. Moreover, the model introduces the parameter of penalty in order to indicate the severity of violations beyond the design constraints of the system given set of design parameters. For a set of design parameters without inducing any violations, the system penalty becomes zero. For those design parameters that induce small penalty but gives good performances

in general, some minor changes of the design parameters values can help reduce it to zero. While it is helpful in assessing the design parameters, penalty is especially useful when performing design optimization as it can be used as a criteria to steer the searching algorithm away from high penalty regions. In the current physics-based model of the integrated cycles, a total of 17 penalties summarized in four categories are defined as below: [20]:

1. When the pinch temperature in the primary heat exchanger or any regenerator is less than 5 Kelvin (K); Particularly, the pinch point is the location in heat exchanger (or regenerator) where the temperature difference between hot and cold fluid is minimum at that location [21].

2. When the outlet temperature of the cold fluid in the regenerators is less than its inlet temperature;

3. When the quality of the working fluid at pumps' inlet is greater than 0;

4. When the vapor quality at turbines' outlet becomes less than 0.87

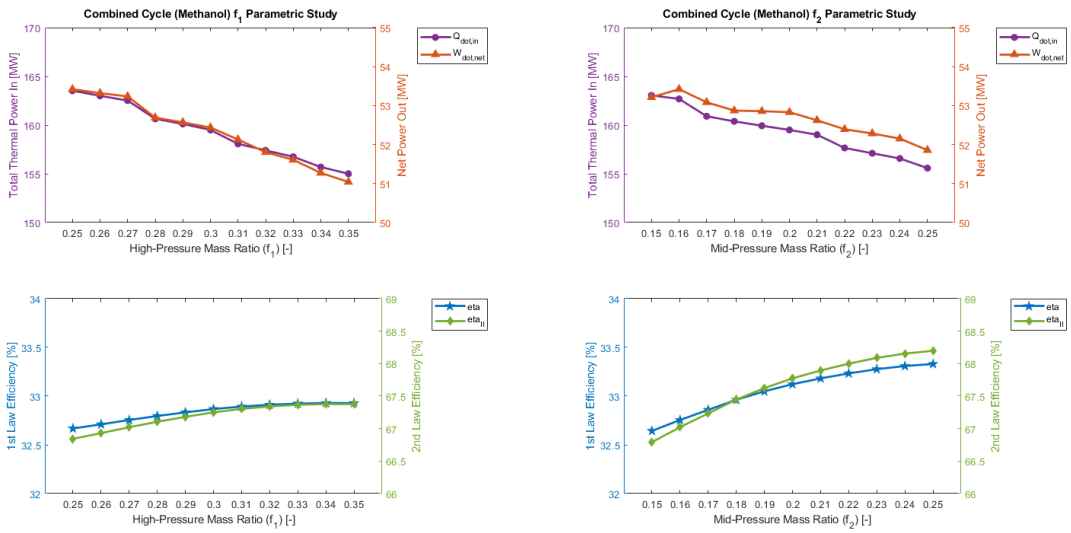


Fig. 1.4: High/Middle Pressure Mass Ratio of Regenerative Methanol Transcritical Cycle.

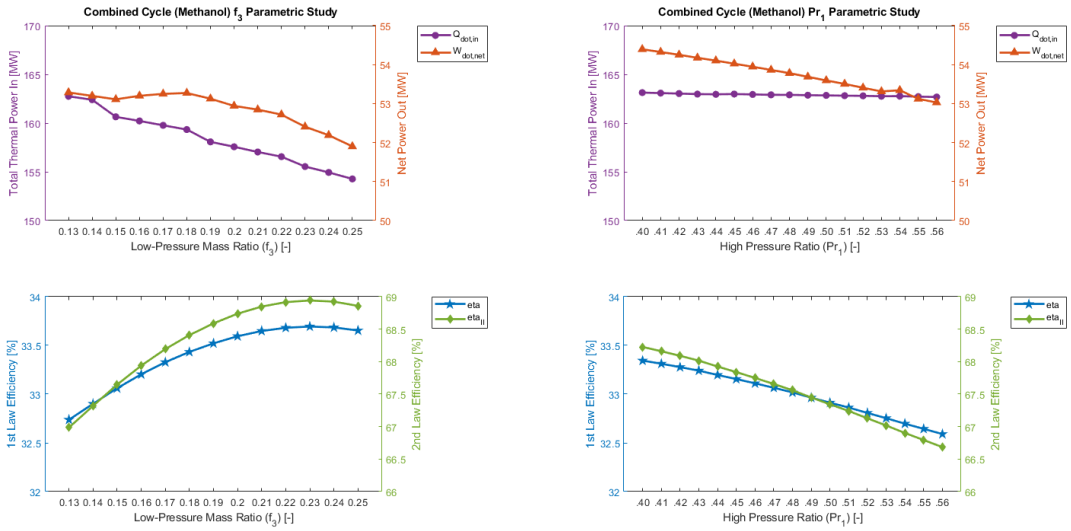


Fig. 1.5: Low Pressure Mass Ratio/High Pressure Ratio of Regenerative Methanol Transcritical Cycle.

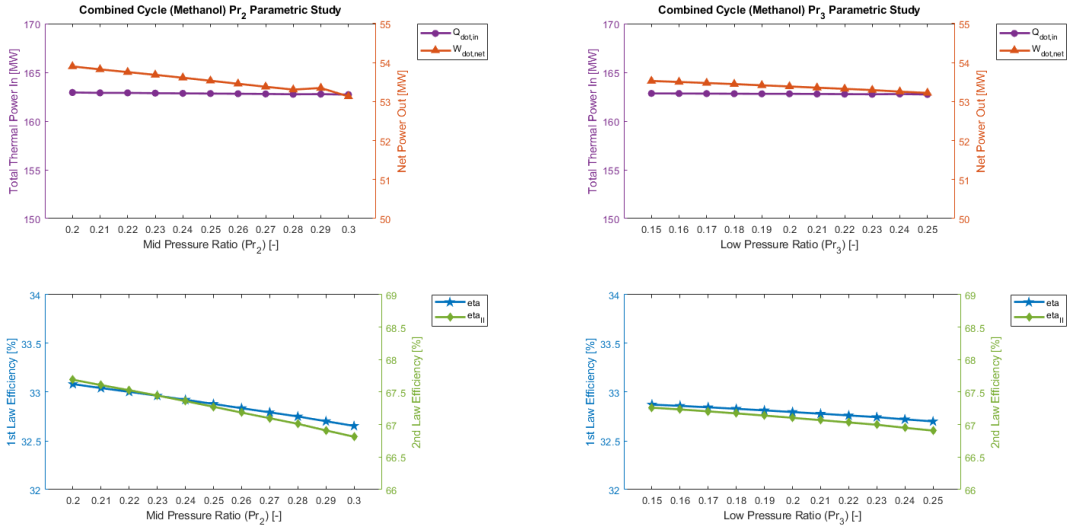


Fig. 1.6: Middle/Low Pressure Ratio of Regenerative Methanol Transcritical Cycle.

The initial approach to finding the Integrated Regenerative Methanol Transcritical Cycle's optimal design parameters is by conducting the parametric studies on each of the design parameters based on the physics-based simulation model. Specifically, when conducting the parametric study on a certain design parameter, the other dependent design parameters are kept constant at their baseline values, and a series of values to the design parameter are

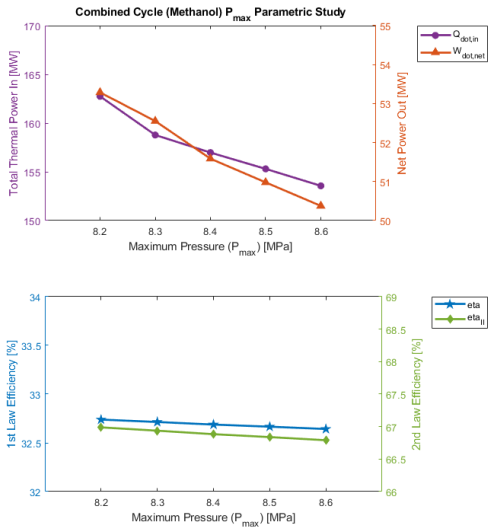


Fig. 1.7: Maximum Temperature/Pressure Parametric Study.

put into the physics-based simulation model one by one to get the corresponding results. By repeating this procedure for each of the design parameters, the trending of the results' variation in terms of each of the design parameter's variation would be obtained. The parametric study results are shown in Fig. 1.4, Fig. 1.5, Fig. 1.6 and Fig. 1.7. It can be observed that in general the net heat into and the net power out from the power cycle decrease with the increase of the seven design parameters. The first and second law efficiencies increase with the increase of the three mass fractions parameters, while decreasing with the increase of the three pressure ratio parameters and the maximum pressure parameters. Especially, peaks appears in the first and second law efficiencies parametric studies with the change of high-pressure mass fraction and the low-pressure mass fraction.

Unfortunately, the physics-based simulation model of the Integrated Regenerative Methanol Transcritical Cycle is a large program consisting of many sub-components, such as heat exchanger and turbines; as the results, it takes significant amount of computational time for the physics-based simulation model to converge, and it takes from a couple of hours to days for the system to complete a round of optimization. To solve the time complexity issue of the optimization with physics-based model, a Machine Learning (ML)-based surrogate model is built to replace the physics-based simulation model in the optimization process. In

Table 1.3: Seven Design Parameters.

Symbol	**Name**	**Range**	**Unit**
P_{max}	Power Cycle Maximum Pressure	[8.22e6, 9.22e6]	Pa
Pr_1	High-pressure Ratio	[0, 1]	-
Pr_2	Middle-pressure Ratio	[0, 1]	-
Pr_3	Low-pressure Ratio	[0, 1]	-
f_1	High-pressure Mass Fraction	[0, 1]	-
f_2	Middle-pressure Mass Fraction	[0, 1]	-
f_3	Low-pressure Mass Fraction	[0, 1]	-

Table 1.4: Surrogate Model Outcome Parameters.

Symbol	**Name**	**Range Bef. DPP**	**Range Aft. DPP**	**Unit**
eta_I	First Law Efficiency	[0.00, 33.90]	[20, 40]	%
LCOE	Levelized Cost of Energy	[78.11, 1672.41]	-	$\frac{\$}{MWh}$
penalty	penalty	[0.00, 300.18]	-	-
f	target	[78.31, 1806.77]	-	-

building the ML-based surrogate model, some preliminary dataset are collected first from the physics-based simulation model. In the dataset, each sample consists of seven design parameters and their corresponding outputs. The ranges of the seven design parameters and the outputs are summarized in Tab. 1.3 and Tab. 1.4, respectively. Particularly, the target, f, is the sum of $LCOE$ and $penalty$ values, and it is used as the minimization target in the optimization process. Secondly, structure of the ML-based surrogate model will be established, and the dataset collected from the physics-based model will be used to train the preliminary surrogate model, so that the weight parameters in the model are correctly defined. The trained surrogate model carries optimized weight parameters so that it is capable of accurately predicting the outputs of a new set of design parameters. Lastly, the ML-based surrogate model and the last-chosen optimizer will be used together as an surrogate-optimizer model, with which the system's optimal design parameters can be eventually found.

1.3 Literature Review

A large number of scientific and engineering fields are confronted with the need for computer simulations to study complex, real world phenomena or solve challenging design problems, but the state-of-the art computer codes for simulating real physical systems are often characterized by vast number of input parameters. Performing high accuracy optimization on such systems is not always feasible because of the need to perform hundreds of thousands or even millions of forward model evaluations in order to obtain convergent statistics [18, 22]. One of the ways to reduce this computational burden is to build a surrogate model of the computationally costly simulation code [23]. Surrogate models, such as the use of neural networks, kernel methods, and other surrogate modeling techniques, are compact and cheap to evaluate, and have proven very useful for tasks such as optimization, design space exploration, prototyping, and sensitivity analysis [18]. For instances, in 2011, Rashidi et al. presented a parametric study for and optimization of a transcritical power cycle, where three distinct multi-layer perceptron artificial neural networks (ANNs) are built as surrogate models with the inlet turbine pressure, inlet turbine temperature and fraction of the maximum power as inputs, and with the thermal efficiency, exergy efficiency and specific network as objective functions, respectively [24]. The procedure comprises three steps. Step 1 is to find thermal efficiency, exergy efficiency, and specific network for different values of inlet turbine pressure, inlet turbine temperature, and fraction of the maximum power using the robust numerical code, engineering equation solver. In step 2, three distinct multi-layer perceptron ANNs based on the data obtained from step 1 are trained. In step 3, three distinct GAs are used to optimize the thermal efficiency, exergy efficiency, and specific network. The results were compared with a previously reported case and were found to be in good agreement. In the same year, Zhao et al. proposed an optimization scheme combing the ANN models and the generic algorithm to optimize the design and operating parameters of the Atkinson cycle engine [25]. Firstly, computation-efficient nonlinear models for the baseline engine were built based on the Artificial Neural Network (ANN) technique. The network has six inputs that are engine speed (N), spark angle (SA), intake

valve closure (IVC), exhaust valve opening (EVO), geometrical compression ratio (GCR) and air-to-fuel ratio (AFR), and each network has one output, which is brake specific fuel consumption (BSFC), torque, knock intensity (KI), and exhaust temperature, respectively. The ANN models were trained and tested using the data computed by a precisely calibrated GT-Power engine simulation model; and experimental results obtained from the actual engine tests have validated the excellent prediction accuracy of the ANN models. Secondly, based on the ANN models, the optimization of geometrical compression ratio (GCR) and operating parameters was performed using a genetic algorithm (GA), which is more suitable for the optimization of highly nonlinear systems [26]. As the results, due to higher peak pressure and larger air-to-fuel ratio (AFR) found from the proposed scheme, the brake thermal efficiency for the Atkinson cycle engine improves 3.95%, and the corresponding fuel economy improves 11.76%. In 2020, Zhang et al. designed a novel dynamic surrogate model based optimization (DSMO) for centralized thermoelectric generation (TEG) system affected by heterogeneous temperature difference (HeTD) to achieve maximum power point tracking (MPPT) [27]. Since heterogeneous temperature difference HeTD usually results in multiple local maximum power points (LMPPs), dynamic surrogate model based optimization (DSMO) needs to rapidly approximate the global maximum power point (GMPP) instead of being trapped at a low quality local maximum power point (LMPP). To avoid a blind search, a radial basis functionn (RBF) network is adopted to construct the dynamic surrogate model of input/output feature according to the real-time data of centralized thermoelectric generation (TEG) system. The proposed surrogate model aims to discover the mapping relationship between the power output and the duty cycle. Since it is a single input single output mapping, a radial basis function (RBF) meshwork [28] is adopted because its excellent nonlinear mapping ability and fast convergence. Furthermore, a greedy search is adopted to accelerate the convergence based on dynamic surrogate model. As the results, the proposed dynamic surrogate model based optimization (DSMO) is proved to be able to rapidly converge to an optimum with a small power fluctuation. In 2018, Ali et al. presented a surrogate-assisted modeling and optimization of the single mixed refrigerant

process of natural-gas liquefaction [29]. To address the computational-burden issue and obtain the results in a reasonable time for the complex single mixed refrigerant process, an approximate surrogate model was developed using a radial basis function combined with a thin-plate spline (TPS) approach. In this study, the radial basis function (RBF) with piecewise smooth RBF kernel was adopted. The six decision variables of the surrogate RBF model are four refrigerants mass flow rates, the condenser pressure and the temperature of vapor/mixed refrigerant (MR) after expansion; while the objective outputs are the compression energy and the penalty of the system. As the results, the optimal performance obtained using the surrogate-assisted modeling methodology was reasonably close to that from the rigorous model-based approach, and the surrogate-assisted modeling can reduce the computational burden in optimization, which was two orders of magnitude lower than for any other reported approach. Besides the power and generator systems mentioned above, there is also great interest in constructing such models in many other fields, serving for the purpose of minimizing the computational cost and maximizing model accuracy [18]. To name just a few, surrogate models are used in grey-box or black-box modeling of a wide variety of systems including electromagnetic modeling of complex structures, geological distributions of minerals, interaction of airflows with airfoils, chemical processes and etc. [30]. For instance, ANN has been implemented efficiently to interpolate the aerodynamic pressure loads for one way unmanned aerial vehicle fluid structure interaction [31]. The result shows good agreement with the actual pressure profile on aircraft compared against two-dimensional curve fitting with higher order polynomials [32]. In addition, in 2018, Kim et al. developed a surrogate model for storm surge prediction using an artificial neural network with the measured tidal level in Korea peninsula [33]. In their scheme, the 59 historical storms during 1978 to 2014 years are used in this modelling. Tidal data recorded for 15 years was applied. The neural network between seven input parameters (i.e., latitude, longitude, moving speed, heading direction, central pressure, radius of strong wind speed, maximum wind speed) and the storm surge is trained by Levenberg-Marquardt backpropagation algorithm. As the results, the developed surrogate model satisfies high-accuracy and high-speed for

predicting he storm surge based on an artificial intelligence method and a grid-free system.

Solving the complex optimization problems in limited time is an indispensable issue in the field of engineering optimization [34]. In general, optimization can be divided into the linear optimization and nonlinear optimization. Within nonlinear optimization, two types of optimization methods can be distinguished: local (around a baseline) and global (over the entire input variable domain of variation) [23]. In addition, there are also optimization methods between local and global, such as Patternsearch [17]. In solving complex, real-world optimization problems, the global optimization method is the most commonly used, such as the Genetic Algorithm [35].

In terms of optimizing thermodynamic systems using surrogate models, the combination of Artificial Neural Network (ANN) and Genetic Algorithm (GA) are commonly methods. In 2011, Suresh et al. studied dealing with the coupled ANN and GA based (neuro-genetic) optimization of a high ash coal-fired supercritical power plant in Indian climatic condition to determine the maximum possible plant efficiency [36]. In the study, the power plant simulation data obtained from 'Cycle-Tempo' ('Cycle-Tempo' is a well-structured package for the steady state thermodynamic modeling and analysis of systems for the production of electricity, heat and refrigeration [37]) is used to train the ANN to predict the energy input through fuel (coal). The optimum set of various operating parameters that result in the minimum energy input to the power plant is then determined by using the trained ANN model as a fitness function with the GA. The maximum plant efficiency is then finally obtained from the power plant simulation in 'Cycle-Tempo' using the set of optimum parameters. As the results, the neuro-genetic optimization methodology significantly reduces the computational effort without compromising the accuracy of the results along with the major advantage of on-line optimization. In 2012, Hajabdollahi et al. modeled and optimized a steam turbine power plant thermo-economically using Artificial Neural Network (ANN) and the fast and elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) [38]. In the study, the ANNs surrogate model was trained with the turbine inlet temperature, boiler pressure, turbines extraction pressures, turbines, pumps

isentropic efficiency, reheat pressure as well as condenser pressure as inputs, and with the efficiency and total cost rate as outputs, repsectively. Then, the Non-dominated Sorting Genetic Algorithm (NSGA-II) is applied to maximize the thermal efficiency and minimize the total cost rate (sum of investment cost, fuel cost, and maintenance cost) simultaneously. As the results, 3.76% increase in efficiency and 3.84% decrease in total cost rate were obtained simultaneously, compared with the actual data of the running power plant. In 2014, Jamali et al. proposed a combined cycle based on the Brayton power cycle and the ejector expansion refrigeration cycle, and optimzied the surface area of the heat exchangers of the system to meet the load requirement with the combined artificial neural network (ANN) and multi-objective genetic algorithm (GA) methods [39]. In the study, the surrogate ANN model was trained with eleven input parameters including some key exergy, pressures, temperatures, heat exchanger diameters and motor agular velocity; and with exergy efficiency and total length of heat exchangers as outputs. The two outputs from ANN are objective functions in multi-objective generic algorithm (GA) optimization. As the results, the solutions approximate the Pareto frontier [40], which means the ANN-GA combined optimization methods results valid outcomes in this study.

As the ANN-GA optimization approach gaining more and more interests, the complexity of the surrogate model applications has also been increasing. For instances, in 2010, the thermodynamic parameters of a supercritical CO_2 power cycle is optimized using exergy efficiency as the objective function. The genetic algorithm (GA) was used under a given waste heat condition. Only four components are included in the cycle: heat recovery vapor generator, turbine, condenser and pump. An artificial neural network (ANN) model with the multi-layer feed-forward network type and back-propagation training is used to achieve design optimization rapidly [14]. Specifically, one hidden layer is used, as it has been proved that one hidden layer is enough to approximate any continuous function as long as it has a sufficient number of neurons [41]. In addition, the sigmoid function is used as the transfer function for the neurons; a back-propagation momentum learning method with a learning rate of 0.2 and a momentum factor of 0.95 is adopted; and the training epoch is set to

1000. As a result, the error for back-propogation training is 1×10^{-5}. It is shown that the optimum thermodynamic parameters of supercritical CO_2 power cycle can be predicted with good accuracy using artificial neural network under variable waste heat conditions. In 2017, a study concerning a thermodynamic and technical optimization of a small scale Organic Rankine Cycle system for waste heat recovery applications was conducted. In the study, it includes seven components: pump, pre-heater, evaporator, super-heater, turbine, regenerator and condenser. An Artificial Neural Network (ANN) model has been developed to maximize the power out (thermodynamic optimization) while keeping the size of the heat exchangers and hence the cost of the plant at its minimum (technical optimization) [15]. Specifically, a multilayer perceptron (MLP) structure has been used and it has been trained and validated using 10-fold cross-validation to improve performance of the network [42]. In addition, one hidden layer was used and the sigmoid function was used as the activation function; the number of neurons of the hidden layer was fixed to 20. A local optimization algorithm was chosen to find the global optimization. The active set algorithm [43] implemented in the fmincon function within the MATLAB Optimization Toolbox [44] was chosen to perform the optimization. As fmincon is a local optimization method, to realize the global optimization with fmincon, a simple multi-start algorithm with 200 random starting points uniformly distributed in Ω has been considered, where Ω is the region defined by the bound constraints which limit the components of the decision variables. The results show that the maximum power output that can be extracted from the heat source is 35.19 kW while the minimum values obtained for the rotational speed and the UA parameter are respectively 24,298 rpm and 44.15 kW/K, with a relative absolute validation error of 0.7981%. The results indicate that the use of ANN is promising in solving complex nonlinear optimization problems in the field of thermodynamics. In 2018, Yang et al. conducted performance prediction and optimization of an organic Rankine cycle (ORC) for diesel engine waste heat recovery based on artificial neural network (ANN) [45]. A test bench of combined diesel engine and ORC waste heat recovery system was developed, and the experimental data used to train and test the proposed ANN model were collected.

Finally, a total of 2,100 typical experimental data samples were obtained and 100 of which were used to test the ANN model. The ANN model was evaluated with different learning rates, training functions and parameter settings, and finally one hidden layer was adopted, and two common metrics to evaluate the network prediction accuracy were used in this work: the mean squared error (MSE) and the correlation coefficient between the prediction value and experimental data. In addition, to improve the prediction precision, the genetic algorithm (GA) is used to optimize the weights of the ANN model by optimizing the hyperparameters, as well as to the prediction accuracy. As the result, the prediction errors of ANN model coupled with the GA are lower than those without using the GA. Specifically, without using GA, the maximum prediction absolute error (AE) can reach up to 0.94 kW, and most of the prediction absolute errors are between –0.4 and 0.4 kW, and the prediction relative errors (RE) are in the range of –14.17% to 17.45%. In comparison, most of the prediction absolute errors (AE) of the ANN model coupled with the GA are between –0.2 and 0.2 kW while the prediction relative errors range from –12.37% to 9.35%. In conclusion, with applying the best parameters, the proposed ANN model shows a strong learning ability and good generalization performance. Compared to the experimental data, the maximum relative error is less than 5%. In 2019, Palagi et al. compared Feedforward, Recurrent and Long-Short-Term-Memory (LSTM) networks in the prediction of the dynamics of a 20 kW ORC system for waste heat recovery. In the study, a training set and a test set have been constructed, which are obtained by collecting measurements from the sensors installed on the ORC test rig. The test rig is composed with four parts: heat transfer loop, ORC loop, cooling loop and transducers. The evaluation metric of all three types of networks used is the mean squared error (MSE). As the result, the Long-Short-Term-Memory architecture achieved the highest performance, in that it correctly predicts the dynamics of the system, showing an error prediction less than 5% and 10% respectively for what concern the predictions of 10 and 60 seconds ahead [46]. In 2020, Dave et al. established an neural networks to provide an accurate and precise multi-dimensional regression of a nuclear reactor's power distribution [47]. The results indicate that neural networks are an appropriate choice for

surrogate models to implement in an autonomous reactor control framework.

The purpose of the thesis study is to build a static surrogate model that replaces the simulation model of the integrated regenerative methanol transcritical cycle with the reactor coolant loop, and then to combine the surrogate model with the optimization method to find the best values of seven design parameters. Compared to previous works in the field, the system to be solved is more complex with more design variables. In developing the surrogate model, both the neural-network based Machine Learning (ML) methods including Deep Neural Network (DNN), 1-D Convolutional Neural Network and ResNet, and the non neural-network based ML methods including Random Forest and Principle Compoent Analysis (PCA) are explored. In addition, to the commonly used GA method for global optimization, other practical optimization methods are also be explored, such as the Pattern Search and the Direct Search. In the end, the optimal combination of surrogate model and the optimization method will be chosen to be incorporated into the final model.

CHAPTER 2

OBJECTIVES

In general, the thesis proposal or dissertation proposal has the following objectives:

1. Develop a baseline surrogate model by experimenting on both the non-neural networks and neural network algorithms that precisely mimic the physical-based model of the integrated regenerative transcritical methanol cycle with the small modular reactor (SMR).

2. Develop an systematic optimization model and incorporate chosen surrogate model into the system. Compare the optimization results from different optimizers with the baseline optimization results.

3. Conduct data analysis on the training dataset in order to further improve the surrogate model-based optimization system.

CHAPTER 3

APPROACH

3.1 Overview

The overall approach of finding the design parameters that optimize the levelized cost of energy (LCOE) and penalty of the integrated regenerative Methanol transcritical cycle is summarized in an algorithm as shown in Fig. 3.1.

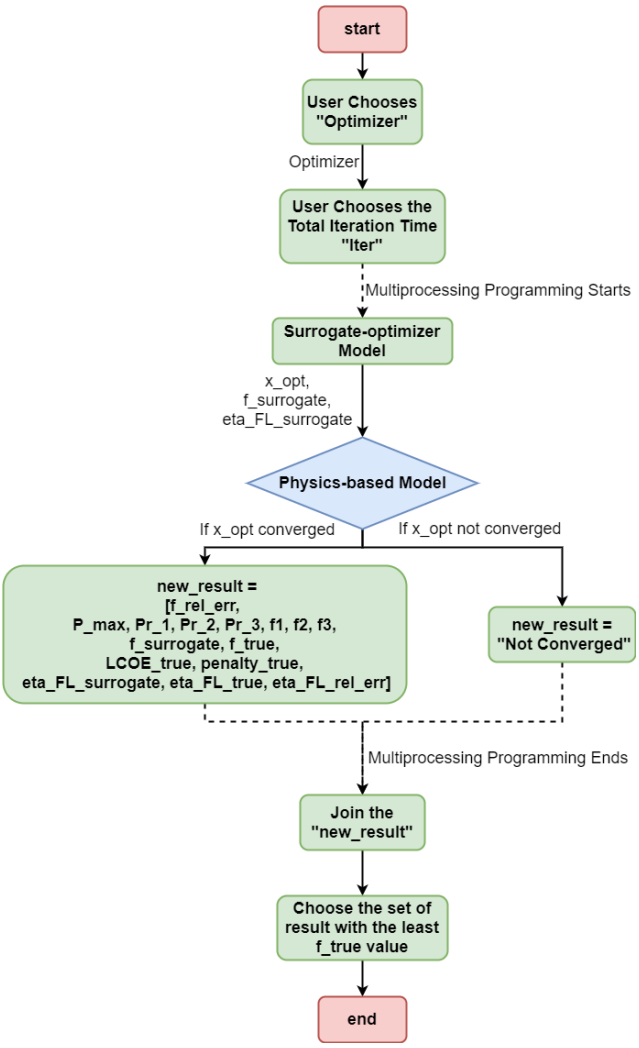


Fig. 3.1: Overall Algorithm of Cycle Optimization with Surrogate Model.

The algorithm starts by having the user choose one of the optimizers, most of which are stochastic in nature. Then, the user chooses the "Total Iteration Time", and one iteration time refers to a complete search with the surrogate-optimizer model, as shown in Fig. 3.2. The surrogate-optimizer model applies the chosen optimizer to find the design parameters that optimize the outcome of ML-based surrogate model, and the details will be illustrated in Sec. 3.2 and 3.3. The reason for executing the surrogate-optimizer model more than once is because the searched design parameters are not guaranteed to converge in the physics-based simulation model. By running the surrogate-optimizer model more than once, it can significantly improve the probability of finding a set of converged design parameters. As shown in Fig. 3.1, after the user chooses the "Total Iteration Times", the multiprocessing programming starts. The design of the multiprocessing programming serves for saving the algorithm execution time by distributing the total iteration into parallel logic cores of the Central Processing Unit (CPU) in the computer. That's being said, the "Total Iteration Times" defined by the user is suggested to not exceed the total number of the logic cores in the computer to guarantee a short execution time. Next, the surrogate-optimizer model would execute the global optimization with the chosen surrogate model and optimizer, then it would output the optimized objective value (or target), $f_{surrogate}$, the corresponding design parameters, $\vec{x}_{opt}$, and the corresponding first law efficiency, $eta_{FL\_surrogate}$, where:

$$f_{surrogate} = \min_{\vec{x}}(LCOE + penalty) \tag{3.1}$$

$$x_{opt} = \arg\min_{\vec{x}}(LCOE + penalty) \tag{3.2}$$

Then, the searched design parameters, $\vec{x}_{opt}$, are fed into the physics-based model. If the $\vec{x}_{opt}$ is physically viable, it is considered as converged, and the corresponding physics-based results would be output; otherwise, the $\vec{x}_{opt}$ are considered as non-converging, and a string "Not Converged" would be output. After all the iterations are executed, the multiprocessing programming ends, and then both the converged and non-converged results are joined into

a final matrix called "new_result". Lastly, from the matrix, the final optimized result with the least objective value, f_{true}, is chosen.

3.2 Optimizers

The optimization algorithm used in the surrogate-optimizer model mentioned in Sec. 3.1 is shown in Fig. 3.2.

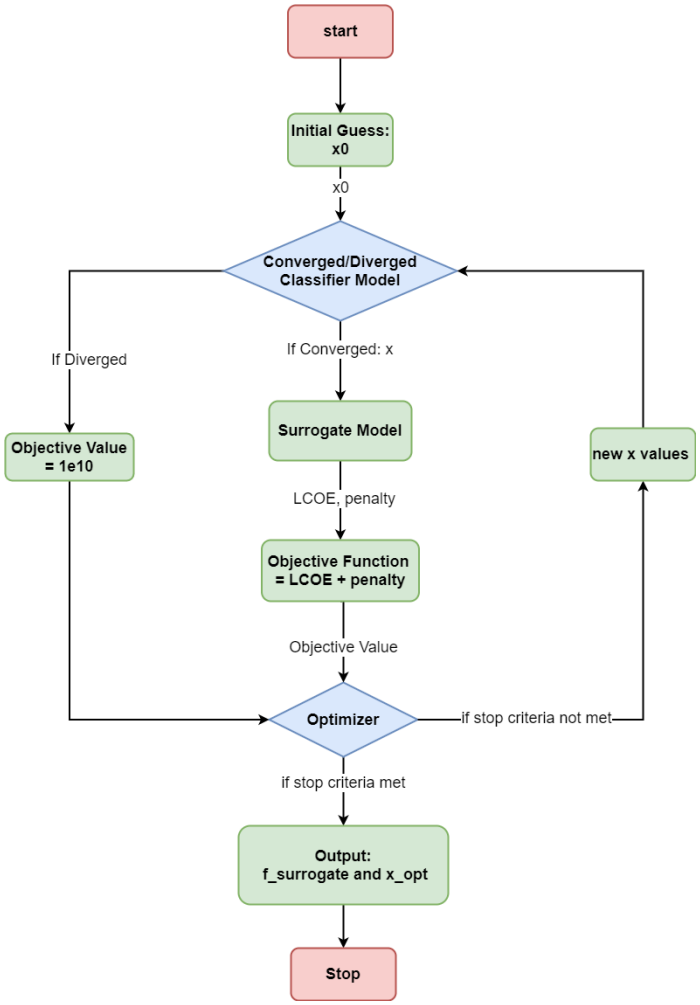


Fig. 3.2: Surrogate-Optimizer model.

To start, the initially guessed design parameters, $\vec{x}_0$, are generated and passed to a "Converged/Diverged Model", which is a machine learning (ML)-based model used to

judge if the incoming design parameters would converge in the surrogate model or not. If the design parameters is classified into "diverge", a large objective value will be fed into the optimizer, otherwise a set of converged design parameters will be passed to the ML-based surrogate model. Next, the target output, f, is passed into the objective function:

$$f = LCOE + penalty \tag{3.3}$$

Then, the corresponding objective value is passed to the optimizer. If the stop criteria in the optimizer are not met, a new set of design parameters, $\vec{x}$, will be updated by the optimizer and fed back to the "Converged/Diverged Classifier Model", and then repeat the process from there until the stop criteria are all met. Finally, the optimized objective value, $f_{surrogate}$, and the corresponding design parameters, $\vec{x}_{opt}$, will be output as the results.

As illustrated, the key of the success of this algorithm would require an accurate ML-based surrogate models and an effective global optimizer. The ML-based surrogate models will be discussed in details in Sec. 3.3, but now the optimizers used to be considered in this project will be described.

3.2.1 Basin-Hopping

The Basin-hopping method stems from the interest in chemical physics for efficiently finding the lowest energy configuration of a (macro)molecular system [48]. These tasks typically have lots of local minima which makes is hard for standard optimization methods because there is a very strong dependency on the initial conditions [49]. Basin-hopping is a stochastic algorithm which attempts to find the global minimum of a smooth scalar function of one or more variables [48, 50–52]. The algorithm is iterative with each cycle composed of the following features [53]:

1. random perturbation of the coordinates
2. local minimization
3. accept or reject the new coordinates based on the minimized function value

Specifically, the algorithm works as follow [49]:

1. Choose a start point.
2. Compute a local minimum.
3. Apply a random perturbation the coordinates of the local minimum.
4. Compute the next local minimum.
5. Compare the local minima and keep the best.

In the original paper that proposed the Basin-hopping method, all coordinates were displaced by a random number in the range $[1, 1]$ times the step size, which was adjusted to give an acceptance ratio of 0.5 at each step [48]. The acceptance test used in the most popular computational package [53] is the Metropolis criterion of standard Monte Carlo algorithms, although there are many other possibilities [51].

3.2.2 Brute Force

The brute-force method was originally proposed for solving the "RECONFIGURATION problem" for radial power distribution networks: given a load profile for a distribution network with a number of tie lines and switching points, find a radial configuration for the network which minimizes the network losses [54]. Brute-force is an exhaustive search algorithm [54] by computing the objective function's value at each point of a multidimensional grid of points, to find the global minimum of the function [55]. The brute force approach is inefficient because the number of grid points increases exponentially - the number of grid points to evaluate is [55]:

$$Ns^{len(x)} \tag{3.4}$$

Where the Ns is the number of grid points along the axes, $len(x)$ is the total number of the axes.

Consequently, even with coarse grid spacing, even moderately sized problems can take a long time to run, and/or run into memory limitations [55].

3.2.3 Differential Evolution

In 1997, Rainer Storn and Kenneth Price presented a new heuristic approach for minimizing possibly nonlinear and non-differentiable continuous space functions [56], called the Differential Evolution.

The general problem formulation is: For an objective function $f : X \subseteq R^D \longrightarrow R$ where the feasible region $X \neq 0$, the minimisation problem is to find $x^* \in X$ such that $f(x^*) \leq f(x) \forall x \in X$, where: $f(x^*) \neq -\infty$ [2].

Global optimization is necessary in fields such as engineering, statistics and finance, but many practical problems have objective functions that are non-differentiable, non-continuous, non-linear, noisy, flat, multi-dimensional or have many local minima, constraints or stochasticity. Such problems are difficult if not impossible to solve analytically, while differential evolution can be used to find approximate solution to such problems [2].

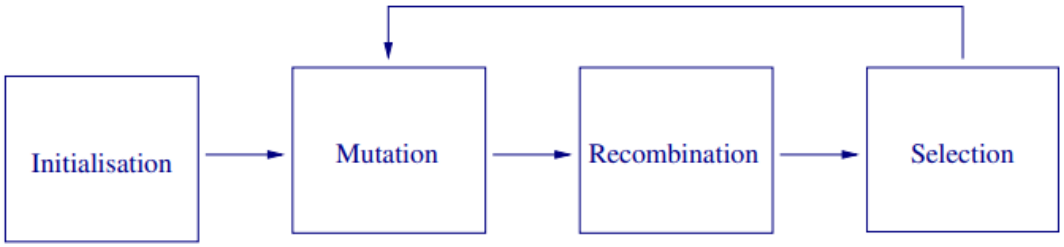


Fig. 3.3: General Evolution Algorithm Procedure [2].

As indicated in Fig 3.3, the differential evolution is an evolutionary algorithm [2]. The differential evolution algorithm includes four steps:

1. **Initialization**: Define upper and lower bounds for each parameter, and randomly select the initial parameter values uniformly on the constrained intervals [2].

2. **Mutation**: For each trial, randomly replace portion of the parent parameter with newly generated parameters.

3. **Recombination/Crossover**: For each trial, recombine portion of the best-elected parent parameters. The crossover increases the diversity of the perturbed parameter vectors [56].

4. **Selection**:The ones with the lowest function value is admitted to the next generation [2].

5. **Iteration**: Mutation, recombination and selection continue until some stopping criterion is reached [2].

3.2.4 SHGO

The simplicial homology global optimisation (SHGO) algorithm is a general purpose global optimisation algorithm based on applications of simplicial integral homology and combinatorial topology. SHGO approximates the homology groups of a complex built on a hypersurface homeomorphic to a complex on the objective function. This provides both approximations of locally convex subdomains in the search space through Sperner's lemma and a useful visual tool for characterising and efficiently solving higher dimensional black and grey box optimisation problems. This complex is built up using sampling points within the feasible search space as vertices. The algorithm is specialised in finding all the local minima of an objective function with expensive function evaluations efficiently which is especially suitable to applications such as energy landscape exploration [57].

SHGO was initially developed as an improvement on the topographical global optimisation (TGO) method. It is proven that the SHGO algorithm will always outperform TGO on function evaluations if the objective function is Lipschitz smooth. While most of the theoretical advantages of SHGO are only proven for when the objective function $f(x)$ is a Lipschitz smooth function, the algorithm is also proven to converge to the global optimum for the more general case where $f(x)$ is non-continuous, non-convex and non-smooth, if the default sampling method is used [57].

3.2.5 Dual Annealing

The Dual Annealing optimization is a stochastic approach derived from [58] combines the generalization of CSA (Classical Simulated Annealing) and FSA (Fast Simulated Annealing) [59] [60] coupled to a strategy for applying a local search on accepted locations [61] [62].

The simulated annealing [63] mentioned above is a stochastic optimization method that is based on an analogy with physical annealing. The procedure is used to bring the atoms that make up the material to their lowest energy configuration. In simulated annealing, the objective of an optimization problem is analogous to the energy of a physical system, the variables are analogous to the position of the atoms, and the feasible region is analogous to the phase space. Introducing a temperature scale to an optimization problem makes the analogy complete [3].

There are in general five steps in realizing the simulated annealing algorithm [64]:

1. We first start with an initial solution $s = S_0$. This can be any solution that fits the criteria for an acceptable solution. We also start with an initial temperature $t = t_0$.

2. Setup a temperature reduction function α.

3. Starting at the initial temperature, loop through n iterations of Step 4 and then decrease the temperature according to alpha. Stop this loop until the termination conditions are reached. The mapping of time to temperature and how fast the temperature decreases is called the **Annealing Schedule**.

4. Given the neighbourhood of solutions $N(s)$, pick one of the solutions and calculate the difference in cost between the old solution and the new neighbour solution.

5. If the difference in cost between the old and new solution is greater than 0 (the new solution is better), then accept the new solution. If the difference in cost is less than 0 (the old solution is better), then generate a random number between 0 and 1 and accept it if it's under the value calculated from the Energy Magnitude equation from before.

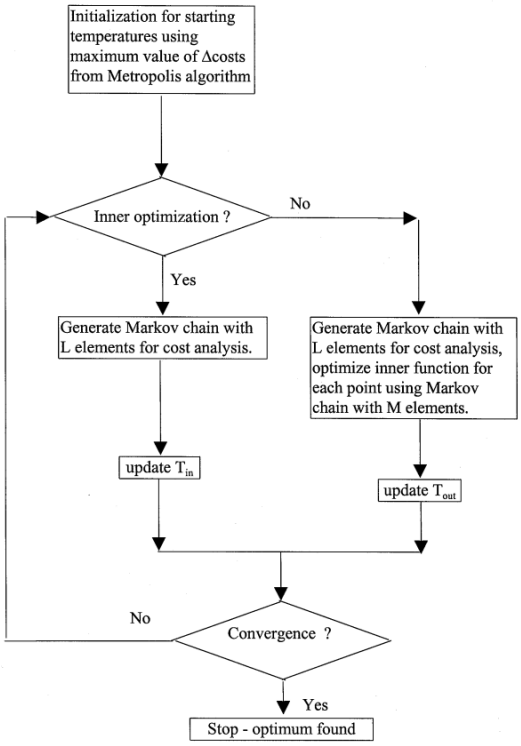


Fig. 3.4: Dual Annealing Algorithm Overview [3]

The outline of the dual annealing algorithm is shown in Fig. 3.4, and there are in general seven steps in realizing the dual annealing algorithm [3]:

1. Generating an initial starting point satisfying the equality and inequality constraints of both the inner and outer problems.

2. Generating new trial points that satisfy the equality and inequality constraints of either the inner or the outer problem.

3. Generating points in the inner searches around the fuzzy region.

4. Generating points in a fuzzy feasible region of the overall bilevel programming problems (BLPP).

5. Generating initial values of T_{out} and T_{in} by generating a sufficiently large Markov chain of trial points.

6. Annealing Schedule: The inner and outer temperatures are annealed with certain cooling schedule.

7. The algorithm stops when both of the following criteria are met:

   - The Euclidean norm between the point obtained after an outer search and the point obtained from the previous outer search is less than ϵ.
   - The following inner optimization does not change the inner objective function value or the the inner optimization variables stay within a range ϵ.

3.2.6 Fmin

Fmin search algorithm uses the Nelder-Mead simplex algorithm as described in Lagarias et al. [65] [66], and the algorithm is not guaranteed to converge to a local minimum [67] (not to mention the global minimum); however, since the fmin is a derivative-free method [67], it can be a very fast solver.

In order to compensate for fmin search algorithm's shortcoming of not capable of global optimization, the method used in [68] is borrowed. In the paper, the function fmincon of the MATLAB Optimization Toolbox has been chosen to perform the optimization, and the global optimization was realized by a simple multi-start algorithm with 200 random starting points uniformly distributed in search domain [68].

Thus, in this project, the global optimization with the fmin search algorithm is realized by applying the following steps:

1. Generating N (sufficiently big enough to cover the whole search domain) initial values over the entire search domain.

2. Doing the N fmin searches simultaneously with the initial values generated in step. 1.

3. Comparing the N search results and pick the optimal one as the global optimization result.

3.3 Converged/Diverged Classifier Model and Surrogate Model

As shown in Fig. 3.2, besides the optimizer, there are two other important components in the surrogate-optimizer model: the Converged/Diverged Classifier Model and the Surrogate Model. Both models are ML-based models, and they share the same dataset in training and evaluating their models. In this section, the dataset used by both models is analyzed, and the converged/diverged classifier model and the surrogate model are described.

3.3.1 Dataset Analysis & Data Pre-processing

The original dataset contains $4,161,536$ samples of design parameters set, which are generated during several days from the physics-based simulation model.

Since the dataset was random-generated, some samples correspond to converging outcomes, while the others do not. The former type of samples is called converged dataset, and the latter type is called diverged dataset. As shown in Fig. 3.5, within the $4,161,536$ oroginal dataset, around 75.11% samples are diverged, and around 24.89% samples are converged.

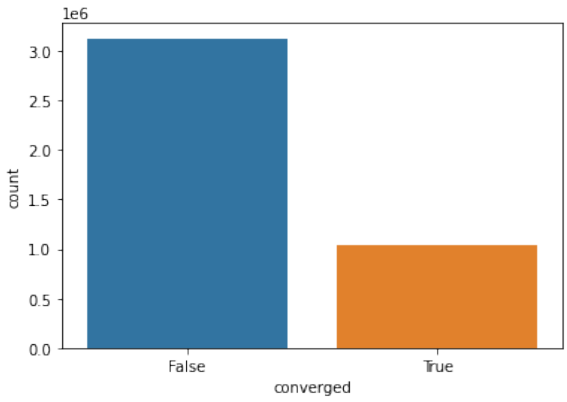


Fig. 3.5: Diverged and Converged Samples in Original Dataset. **False:** Diverged; **True:** Converged

Similarly, as indicated in Fig. 3.2, during the optimization process, some newly generated design parameters x may be diverged samples as well. In order to guarantee the uninterrupted execution of the algorithm, the converged/diverged classifier model that is

capable of filtering out diverged incoming samples would be eagerly needed.

The $1,035,757$ converged samples will be used to train and evaluate the surrogate model, and they are visually illustrated in the Fig. 3.6, in which the corresponding outputs of each sample of design parameters are indicated, including the LCOE percentage change, penalty and the 1^{st} law efficiency. Since the optimal cycle is expected to produce the LCOE as low as possible (and thus LCOE percentage change as low as possible as well) and to produce penalty near 0, the optimal samples of design parameters are ones blackly circled at the lower-left corner in the figure. Those optimal samples are of a very small portion. Thus, in order to help the model to figure out the corresponding outcomes more quickly and accurately, some data pre-processing (DPP) steps are carried out in training both the classifier and the surrogate models.

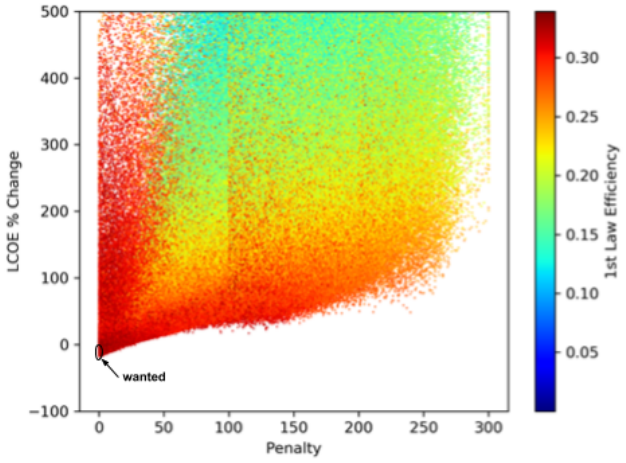


Fig. 3.6: Dataset Visualization

Firstly, the seven design parameters and their corresponding ranges are summarized in the Tab. 1.3. It can be seen that all the design parameters range from zero to one, except the Power Cycle Maximum Pressure, which ranges from $8.22e6$ Pa to $9.22e6$ Pa. In the training process, if some input features' value are significantly larger than the other ones, the model may ignore the features with small values and thus would end up being a biased model. In order to address this issue, the values of the Power Cycle Maximum Pressure,

P_{max}, are normalized between zero and one by:

$$P_{max\_new} = (P_{max} - 8.22e6)/1e6$$

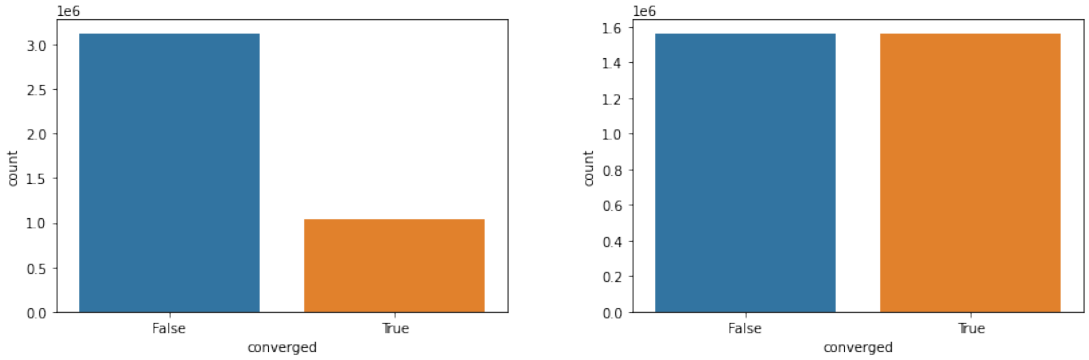
Additionally, for the surrogate model, the desired outcomes and their corresponding ranges are listed in Tab. 1.4. It is obvious that ranges of the three parameters are very different from each other. For the similar reason of the input dataset, if some outputs' value are significantly larger than the rest ones, the model may not train the outcomes with small values so that the model would still end up being biased. To address these issues, different methods are applied for different outcomes. For the surrogate model, due to the following reasons, eta_I is trained alone in one neural network, while the penalty and LCOE are trained together in another one:

1. The target value f is the combination of LCOE and penalty values.
2. Lower values are expected in a optimal model for both the LCOE and the penalty.
3. The design range of both the LCOE and the penalty are much wider than that of eta_I.
4. The training dataset of both the LCOE and the penalty are noisier than ones of the eta_I as indicated in Fig. 3.6.

In addition, based on preliminary studies, it is found that the viable first law efficiency value, eta_I, of the system lies between 20% to 40%, thus, samples with $eta_I < 20\%$ and $eta_I > 40\%$ are eliminated. In the mean time, the $LCOE$ and penalty values in the dataset are untouched. In summary, the training dataset after being pre-processed by the eta_I is used as the training dataset for both the eta_I and the LCOE-penalty models.

3.3.2 Converged/Diverged Classifier Model

The purpose of the converged/diverged classifier model in the surrogate-optimizer model is to make sure the whole algorithm does not break if it runs into any unreasonable input samples, x. As shown in Fig. 3.2, during each iteration, the optimizer would



(a) Dataset Before Re-balancing. (b) Dataset After Re-balancing.

Fig. 3.7: Classifier Model Dataset Re-balancing

update a new set of design parameters which is supposed to produce more optimal results and to be fed into the surrogate model to obtain the corresponding objective value; however, since the new set of design parameters is not guaranteed to converge or result in reasonable objective values, the optimizer may risk running into unexpected errors and break. In order to compensate for the shortcoming, the converged/diverged classifier model is developed and put before the surrogate model, so that when the new set of design parameters could results in diverged results, the algorithm would detour around the surrogate model and directly assign a significantly large objective value in order to keep the optimizer running and not breaking in the middle of execution; otherwise, the new set of design parameters would be fed into the surrogate model in order to obtain the corresponding objective value.

The original dataset of the converged/diverged classifier model is shown in Fig. 3.7a, in which the diverged (False) samples are about triple of the converged (True) samples. This is an so-called class-imbalanced dataset, which could result in biased training model. In the past decade, many machine learning approaches have been developed to cope with imbalanced data classification, most of which have been based on re-sample techniques, cost sensitive learning and ensemble methods [69–71]. Within those methods, the resampling methods are more versatile because they are independent of the selected classifier [72]. In general, the resampling techniques fall into three groups depending on the method used to balance the class distribution [73]:

- **Over-sampling methods:** eliminating the harms of skewed distribution by creating new minority class samples. Two widely-used methods to create the synthetic minority samples are randomly duplicating the minority samples and the Synthetic Minority Over-sampling TEchnique (SMOTE) [74].

- **Under-sampling methods:** eliminating the harms of skewed distribution by discarding the intrinsic samples in the majority class. The simplest yet most effective method is Random UnderSampling (RUS), which involved the random elimination of majority class examples [75].

- **Hybrid methods:** these are a combination of the over-sampling method and the under-sampling method.

For the convenience and effectiveness, the method adopted for re-balancing the dataset of the converged/diverged classifier model is the Over-sampling method by randomly duplicating the minority samples to create the synthetic minority samples. The re-balanced dataset is shown in Fig. 3.7b.

With the training and evaluation dataset ready, another important task for building a successful model is to choose the best Machine Learning Method. In this project, six different Machine Learning classifier methods are built and optimized, respectively, and the one with the best performance will be adopted as the final classifier to be used in the surrogate-optimizer model.

3.3.2.1 Multi-layer Feed-Forward (MLF) Neural Network

Multi-layer Feed-Forward (MLF) neural networks, trained with a back-propagation learning algorithm, are the most popular neural networks. As shown in Fig. 3.8, MLF neural network consists of neurons, that are ordered into layers. The first layer is called the input layer, the last layer is called the output layer, and the layers between are hidden layers. The training mode begins with arbitrary values of the weights - they might be random numbers - and proceeds iteratively. Each iteration of the complete training set is called an epoch. In each epoch the net- work adjusts the weights in the direction that

reduces the error. As the iterative process of incremental adjustment continues, the weights gradually converge to the locally optimal set of values. Many epochs are usually required before training is completed [4].

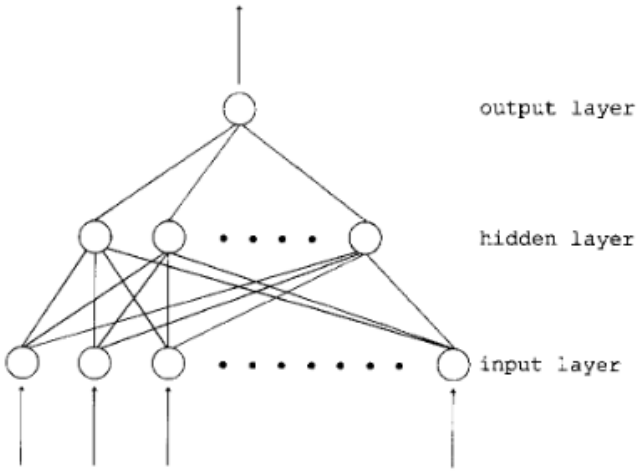


Fig. 3.8: Typical Feed-Forward Neural Network Composed of Three Layers [4].

The schematic of the converged/diverged classifier model is shown in Fig. 3.9, and the corresponding optimized hyper-parameters of the MLF neural network is summarized in Tab. 3.1 based on the parametric studies.

Table 3.1: Hyper-parameters of the Classifier MLF Neural Network.

Hyper-parameters	**Value**
batch size	16,384
epoch	150
input size	7
hidden size 1	1,024
hidden size 2	512
dropout rate	0.5
learning rate	0.001
weight decay	0.0001

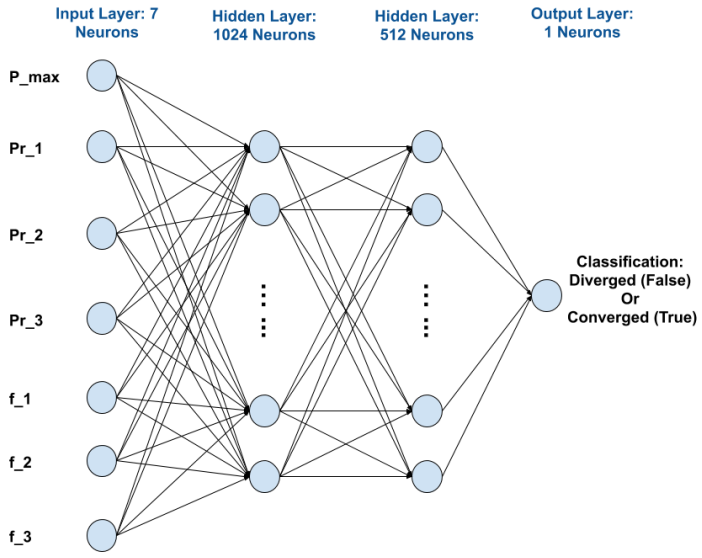


Fig. 3.9: Converged/Diverged Classifier Model Schematic.

Where the batch size is a hyper-parameter that defines the number of samples to work through before updating the internal model parameters. The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset [76], and based on the parametric results, $batchsize = 16,384$ in the give schematics would effectively reduce the oscillation in training process, while avoiding losing any significant information. The number of epochs is the number of complete passes through the training dataset. The number of epochs can be set to an integer value between one and infinity [76], but there exists an optimal value, because too small of the epoch number may cause under-fitting of the model, while too big of the epoch number could cause over-fitting of the model. Under the given schematic, $epoch = 150$ gives the optimal performance.

Additionally, the input size is set to be 7 because there are in total seven design parameters in each input sample, as shown in Fig. 3.9; for the similar reason, the output of the MLF neural network consists only one neuron because the network is expected to output one of two classifications each time: the input sample is either diverged (False) or converged (true).

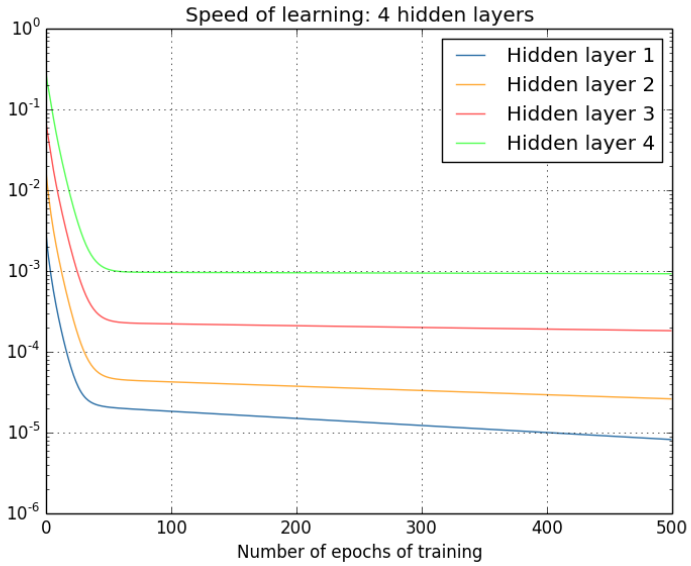
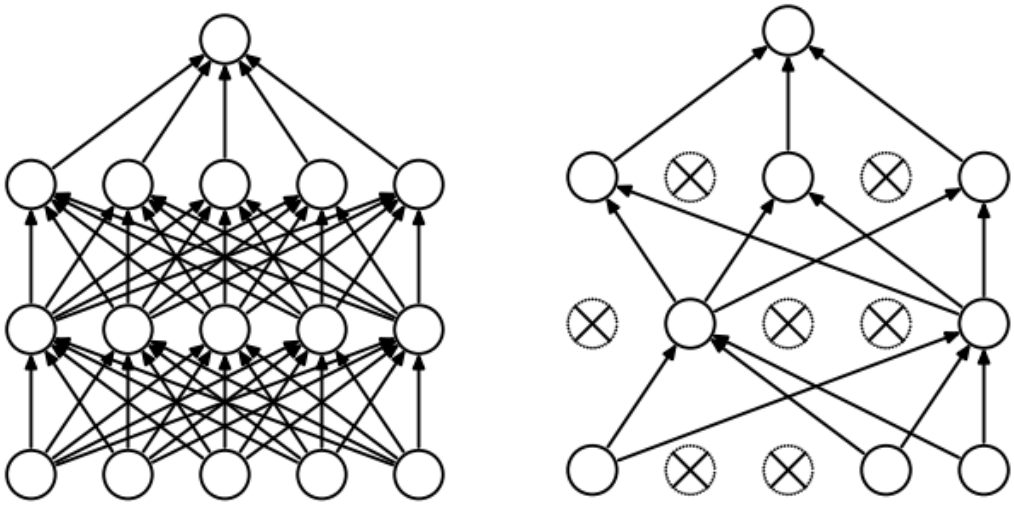


Fig. 3.10: Proof of the Vanishing Gradient Problem: Learning Speeds of Four Hidden Layers of A Deep Network [5].

One of the most important schematics of this classifier MLF neural network is that there are two hidden layers, with the hidden layer 1 of the size 1,024 and the hidden layer 2 of the size 512. It is believed that networks with many more hidden layers to be more powerful, because such networks could use the intermediate layers to build up multiple layers of abstraction. However, the deep networks with too many hidden layers may cause the vanishing gradient problem, which is the phenomenon shown in Fig. 3.10. In an example of deep network with four hidden layers, it is obvious that early hidden layers learn much more slowly than later hidden layers [5]. Therefore, by applying the model hyper-parameter optimization technique, the Grid Search [77], the optimal schematics of the classifier MLF network has been determined as summarized in Tab. 3.1.

Moreover, the dropout is a technique that addresses issues of overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Fig. 3.11 [6]. In this project,



(a) Standard Neural Net [6]. (b) After applying dropout [6]

Fig. 3.11: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped [6].

after the parametric study, the optimal dropout rate is found to be 0.5, meaning temporarily dropping 50% of the hidden neurons during the training process. Another important technique used in the classifier MLF neural network in order to enable faster and more stable training of deep neural networks is called the Batch Normalization (BatchNorm). It is a widely adopted technique achieved by introducing additional network layers that control the first two moments (mean and variance) of these distributions [78], and it is used in both the hidden layers of the classifier MLF neural network. The corresponding training process is shown in Fig. 3.12

The learning rate and weight decay are key hyper-parameters used to define the optimizer of the neural networks. Gradient descent is the preferred way to optimize neural networks and many other machine learning algorithms but is often used as a black box, called the optimizer. The most popular optimizers include: Momentum, Adagrad, RMSprop and Adam [79]. During the last years the Adam (Adaptive Moment Estimation) Optimizer has become one of the most used optimization methods for training neural networks [80], which is designed by Kingma and Ba [81] by combining the advantages of two

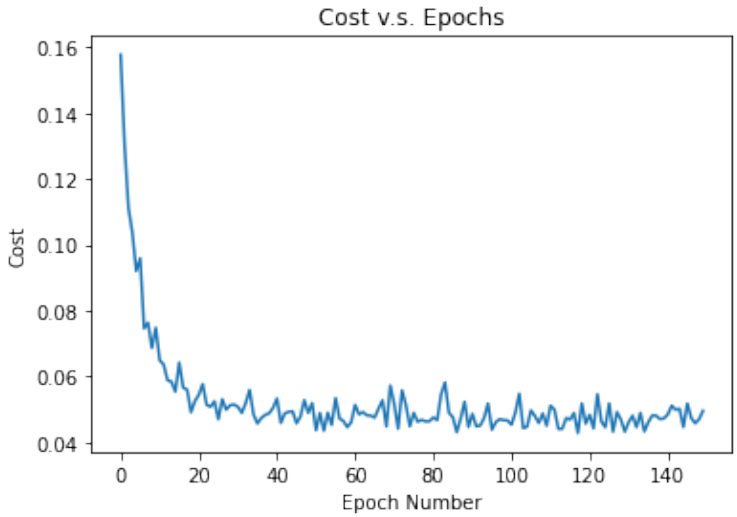


Fig. 3.12: Classifier MLF Neural Network Training Process.

recently popular methods: AdaGrad [82], which works well with sparse gradients, and RMSProp [83], which works well in on-line and non-stationary settings. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients [81]. In the classifier MLF neural network, with the parametric studies, $learning\ rate = 0.001$ and $weight\ decay = 0.0001$ prove to result in the optimal balance between the training effectiveness and the training smoothness during the training process, as shown in Fig. 3.12.

Lastly, the loss function used is the Binary Cross Entropy (BCE) function, for it is the most suitable for the classification problems with only two types of categories. Conventionally, ReLU is used as an activation function in deep neural networks (DNNs), with Softmax function as their classification function [84], which is also what is used in the classifier MLF neural network.

3.3.2.2 Random Forest

In general, a random forest (RF) classifier is an ensemble classifier that produces multiple decision trees, using a randomly selected subset of training samples and variables [85]. As shown in Fig. 3.13, random forest is an ensemble of unpruned classification or regression

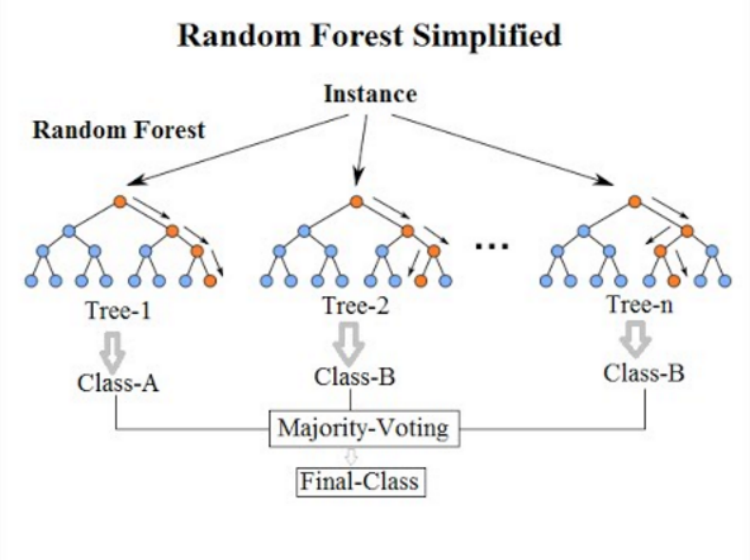


Fig. 3.13: Random Forest Classifier Simplified Example [7]

trees created by using bootstrap samples of the training data and random feature selection in tree induction. Prediction is made by aggregating (majority vote or averaging) the predictions of the ensemble [86].

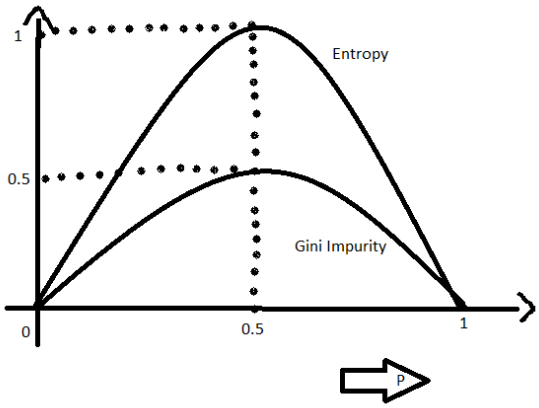
Thus, the algorithm works four steps [87]:

1. Select random samples from a given dataset.
2. Construct a decision tree for each sample and get a prediction result from each decision tree.
3. Perform a vote for each predicted result.
4. Select the prediction result with the most votes as the final prediction.

The most important hyper-parameters of the classifier random forest model of the project are listed in Tab. 3.2. The tree numbers means the number of trees to be assigned in the forest, where too small of the value could result in inaccurate outcomes, while too large of the value could cause much longer time in training process thus reducing the training efficiency. With parametric studies, $tree\ numbers = 200$ in the model would result in optimal results.

Table 3.2: Hyper-parameters of the Classifier Random Forest.

Hyper-parameters	**Value**
tree numbers	200
criterion	Gini
max. depth	None
min. samples split	2
min. samples leaf	1



$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

Fig. 3.14: Gini and Entropy Comparison [8].

Entropy and Gini are two mostly used criteria for measuring the purity of the sub split in the decision trees. The internal working of both methods is very similar, but if we compare both the methods then Gini Impurity is more efficient than entropy in terms of computing power. As you can see in the Fig. 3.14 for entropy, it first increases up to 1 and then starts decreasing, but in the case of Gini impurity it only goes up to 0.5 and then it starts decreasing, hence it requires less computational power. The range of Entropy lies in between 0 to 1 and the range of Gini Impurity lies in between 0 to 0.5. Hence we can conclude that Gini Impurity is better as compared to entropy for selecting the best features [8]. Thus, Gini is chosen as the criteria used in the classifier random forest model.

The maximum depth of the tree implies its meaning. When None is chosen, it means nodes are expanded until all leaves are pure or until all leaves contain less than minimum split samples [88], which is two in our case. In addition, the minimum sample leaf means the minimum number of samples required to be at a leaf node. In this case, a split point at any depth will only be considered if it leaves at least the one training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. Apparently, $max.\ depth = None$, $min.\ sample\ split = 2$ and $min.\ sample\ leaf = 1$ would result in the most accurate outcomes with minor sacrifices of computational time.

3.3.2.3 Gaussian Naïve Bayes

Naïve Bayes is one of the most efficient and effective inductive learning algorithms for machine learning and data mining. Its competitive performance in classification is surprising, because the conditional independence assumption on which it is based is rarely true in real-world applications [89].

Naïve Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n [90]:

$$P(y|x_1, ..., x_n) = \frac{P(y)P(x_1, ..., x_n|y)}{P(x_1, ..., x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) = P(x_i|y)$$

for all i, this relationship is simplified to:

$$P(y|x_1, ..., x_n) = \frac{P(y)\Pi_{i=1}^{n} P(x_i|y)}{P(x_1, ..., x_n)}$$

Since $P(x_1, ..., x_n)$ is constant given the input, we can use the following classification rule:

$$P(y|x_1, ..., x_n) \propto P(y)\Pi_{i=1}^{n} P(x_i|y) \Longrightarrow \hat{y} = \arg\max_{y} P(y)\Pi_{i=1}^{n} P(x_i|y)$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i|y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i|y)$. For example, for Gaussian Naïve Bayes classification, the likelihood of the features is assumed to be Gaussian:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

where the parameters σ_y and μ_y are estimated using maximum likelihood.

Similarly, the Benroulli Naïve Bayes classifier models features using the Benroulli distribution, and Multinomial Naïve Bayes classifier models features using the Multinomial distribution. In the project, the Gaussian Naïve Bayes classifier is applifed, for it is the most suitable for binary classification tasks.

3.3.2.4 K Nearest Neighbor

K Nearest Neighbor (KNN) is a unsupervised machine learning algorithm that classifies data points based on the points that are most similar to it [9], as shown in Fig. 3.15. The key idea of a standard KNN method is to predict the label of a test data point by the majority rule, that is, the label of the test data point is predicted with the major class of its k most similar training data points in the feature space [91]. The overall algorithm of the K Nearest Neighbor is shown in Alg. 3.1 [92].

The most important hyper-parameters applied in the project are listed in Tab.3.3, where the number of neighbors is K in KNN, and it refers to the number of nearest neighbours to include in the majority of the voting process. In practice, choosing the value of k

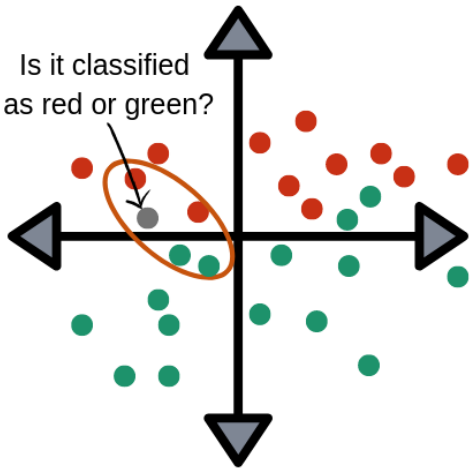


Fig. 3.15: Example of K Nearest Neighbor Classifier [9].

Algorithm 3.1 Overall Algorithm of K Nearest Neighbor [92].

Input:
The training dataset
Initialized the value of k
Output:
The predicted class
Begin
for 1 to total number of training dataset samples
Do
Begin
Calculate the distance between test data and each row of training data.
Sort the calculated distances in ascending order based on distance values
Get top k rows from the sorted array
Get the most frequent class of these rows
Return the predicted class
End
End

is [93]:

$$k = \sqrt{N}$$

where N stands for the number of samples in your training dataset [93]. In this project, since the total dataset after re-balancing is $3,075,218$, and 80% of them is used as training dataset, the corresponding K value is calculated from:

$$k = \sqrt{3,075,218 \times 80\%} \approx 1,568$$

Table 3.3: Hyper-parameters of the Classifier K Nearest Neighbor.

Hyper-parameters	**Value**
number of neighbors	1568
weights	distance

Weights are functions used to calculate the importance of the neighbor points to the new sample point. There are many different methods to represent the functions, but the most popular twos are [94]:

1. **uniform**: uniform weights. All points in each neighborhood are weighted equally.

2. **distance**: weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

In this project, the distance method is applied in order to gain more accurate outcomes.

3.3.2.5 Logistic Regression

Logistic regression (LR) is a standard probabilistic statistical classification model that has been extensively used across many disciplines. Different from linear regression, the outcome of LR on one sample is the probability that it is positive or negative, where the probability depends on a linear measure of the sample [95], as shown in Fig. 3.16. Therefore, LR is actually widely used for classification.

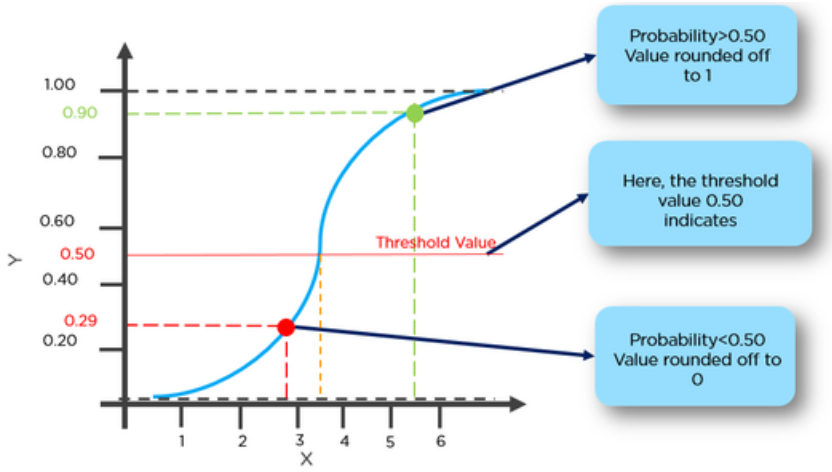


Fig. 3.16: Logistic Regression Logic [10].

More formally, for a sample $x_i \in R^P$ whose label is denoted as y_i, the probability of y_i being positive is predicted to be [95]:

$$P\{y_i = +1\} = \frac{1}{1 + e^{-\beta^T x_i}}$$

given the LR model parameter β. In order to obtain a parameter that performs well, often a set of labeled samples $\{(x_1, y_1), ..., (x_n, y_n)\}$ are collected to learn the LR parameter β which maximizes the induced likelihood function over the training samples.

In the classifier logistic regression model of this project, the key parameters used are listed in Tab. 3.4.

Table 3.4: Hyper-parameters of the Classifier Logistic Regression.

Hyper-parameters	**Value**
penalty	L_2
multi-class	ovr

where penalty, or so-called regularization techniques are used to address over-fitting issues. Specifically, the logistic regression with L_2 penalty adds "squared magnitude" of

coefficient, $\lambda \sum_{j=1}^{p} \beta_j^2$, as penalty term to the loss function, as shown below [96]:

$$cost = \sum_{i=1}^{n} (y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

In comparison with the L_2 penalty, the L_1 penalty replaces the "squared magnitude" of coefficient from $\lambda \sum_{j=1}^{p} \beta_j^2$ to $\lambda \sum_{j=1}^{p} |\beta_j|$. There are many other types of penalties, but L_1 and L_2 are mostly used ones. The key difference between these techniques is that $L_1 penalty$ shrinks the less important feature's coefficient to zero thus, removing some feature altogether. So, this works well for feature selection in case we have a huge number of features [96]. To avoid such extreme cases, the L_2 penalty is adopted for mitigating the over-fitting issues in the logistic regression classification algorithm.

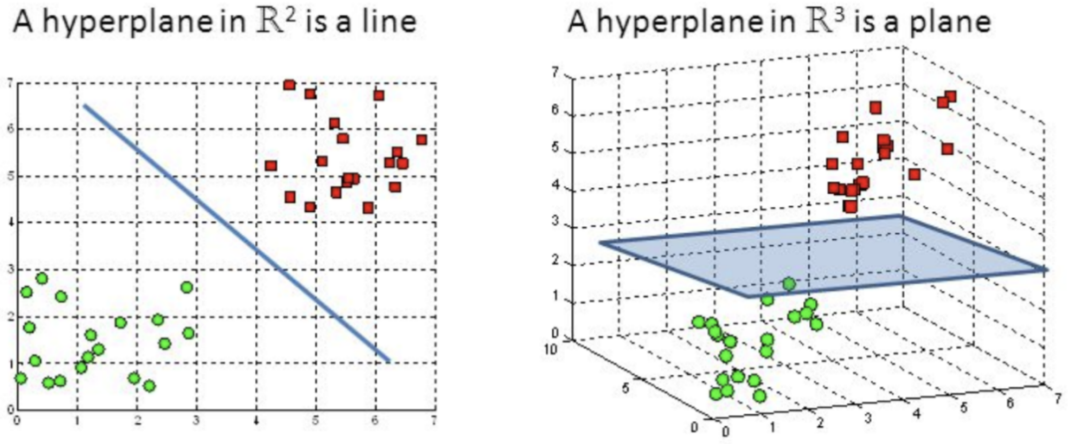
The multi-class is an important functor in the logistic regression solver software [97], in which if the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary; 'auto' selects 'ovr' if the data is binary, and otherwise selects 'multinomial'. In this project, since the classification is either converged or diverged, 'ovr' is the most suitable functor option.

3.3.2.6 Support Vector Machine

Support vector machines (SVMs) form an important part of learning theory. They are very efficient for many applications in science and engineering, especially for classification problems (pattern recognition) [98]. The support vector machines were introduced by Boser, Guyon and Vapnik [99] with polynomials kernels, and by Cortes and Vapnik [100] with general kernels.

The classification of SVMs into respective categories is done by finding the optimal hyperplane that differentiates the two classes in the best possible manner, as shown in Fig. 3.17 [11].

The mathematics behind the SVM binary classifier is illustrated as following. Firstly, the hypothesis function h is defined as:



(a) Hyperplane in two dimensions [11].

(b) Hyperplane in three dimensions [11].

Fig. 3.17: Examples of SVM Hyperplanes. [11].

$$h(x_i) = \begin{cases} +1, & if\ w \cdot x + b \geq 0 \\ -1, & if\ w \cdot x + b < 0 \end{cases}$$

The point above or on the hyperplane will be classified as class $+1$, and the point below the hyperplane will be classified as class -1. Secondly, compute the (soft-margin) SVM classifier amounts to minimizing an expression of the form:

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y_i \cdot (w \cdot x_i - b))\right] + \lambda\|\omega\|^2$$

We focus on the soft-margin classifier since choosing a sufficiently small value for lambda yields the hard-margin classifier for linearly-classifiable input data [11].

Table 3.5: Hyper-parameters of the Classifier Support Vector Machine.

Hyper-parameters	**Value**
penalty	L_2
multi-class	ovr

The key hyper-parameters of the classifier SVM of the project are listed in Tab. 3.5, which are exactly same as the logistic regression, and they also share the same reasons of the same choices.

3.3.3 Surrogate Model

As shown in Fig. 3.2, the machine-learning based surrogate model is used as a replacement for a practically-based simulation model for the integrated regenerative methanol transcritical cycle. After the converged/diverged classifier model filtered out the diverged input samples, the surrogate model would take in the converged samples and output the corresponding objective values, which can be then fed into the optimizer. The main reason for the replacement is to accelerate the system optimization speed. For that, the accuracy of the surrogate model is of top importance.

3.3.3.1 Multi-layer Feed-Forward (MLF) Neural Network

The multi-layer feed-forward neural network, as described in Sec. 3.3.2.1, is used to build the surrogate model with the network's optimized hyper-parameters summarized in Tab. 3.6.

Table 3.6: Hyper-parameters of the Surrogate MLF Neural Network.

Hyper-parameters	**Value**
batch size	16,384
epoch	1,000
input size	7
hidden size 1	1,024
hidden size 2	512
dropout rate	0
learning rate	0.01
weight decay	0.0001

The definition of the terminologies are described in detail in Sec. 3.3.2.1. The corresponding training process is shown in Fig. 3.18.

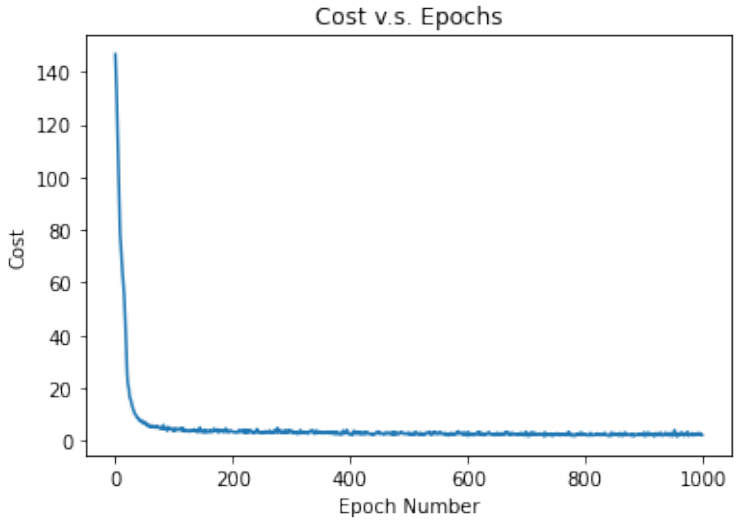


Fig. 3.18: Surrogate MLF Neural Network Training Process.

3.3.3.2 Separate MLF Neural Network

As discussed in Sec. 3.3.1, due to a series of reasons, the penalty and the levelized cost of energy, $LCOE$, are trained together, while the first law efficiency, eta_I, is trained alone in another neural network.

Table 3.7: Hyper-parameters of the Surrogate $penalty + LCOE$ MLF Neural Network.

Hyper-parameters	**Value**
batch size	8,192
epoch	500
input size	7
hidden size 1	256
hidden size 2	128
dropout rate	0
learning rate	0.01
weight decay	0.0001

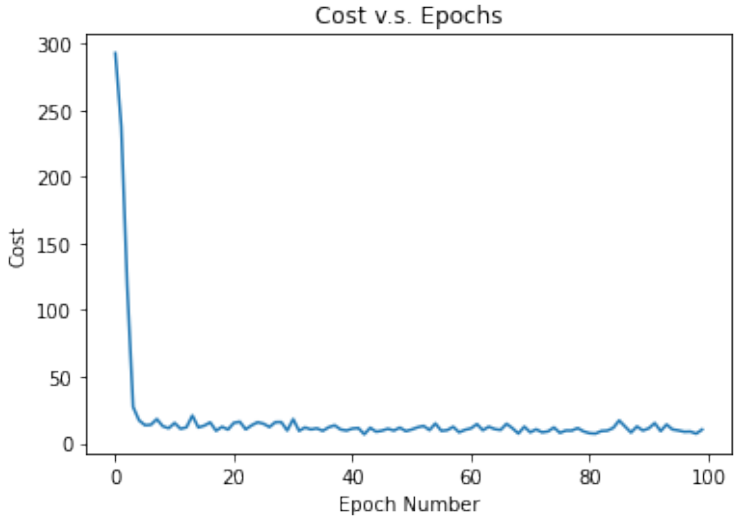


Fig. 3.19: Surrogate $penalty + LCOE$ MLF Neural Network Training Process.

The optimized hyper-parameters for the $penalty + LCOE$ network is summarized in Tab. 3.7, and the corresponding training process is shown in Fig. 3.19.

In addition, the optimized hyper-parameters for the eta_I network is summarized in Tab. 3.8, and the corresponding training process is shown in Fig. 3.20.

Table 3.8: Hyper-parameters of the Surrogate eta_I MLF Neural Network.

Hyper-parameters	**Value**
batch size	16,384
epoch	1000
input size	7
hidden size 1	256
hidden size 2	128
dropout rate	0
learning rate	0.1
weight decay	0.0001

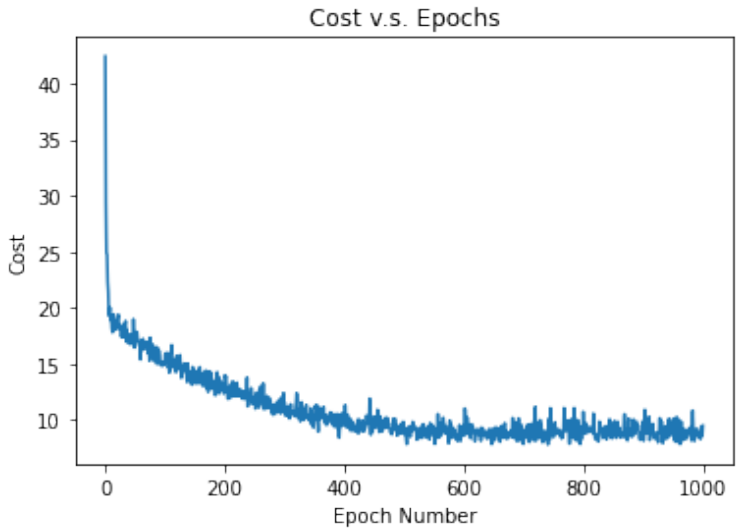


Fig. 3.20: Surrogate eta_I MLF Neural Network Training Process.

3.3.3.3 Penalty Neural Networks

From this point on, all the rest surrogate models are $eta_I + LCOE$ neural networks, and they share the *penalty* surrogate model, which is to be presented in this section.

The reason for training the penalty surrogate model individually is because the penalty dataset is highly noisy and random, which causes that it is extremely hard for any surrogate models to find the internal patterns of the penalty dataset. When the penalty is trained along with the eta_I and $LCOE$, the average prediction accuracy for all targets is deteriorated.

Specifically, since the design parameters corresponding to penalty values smaller than 1 are expected, the penalty dataset is divided into two groups: One with penalty values smaller or equal to 1, and one with penalty values greater than 1. The two types of penalty dataset are used to build the large penalty MLF neural network and the small penalty MLF neural network, respectively.

The hyper-parameters of the surrogate large *penalty* MLF neural network model are summarized in Tab. 3.9 and the corresponding training process is shown in Fig. 3.21.

The hyper-parameters of the surrogate small *penalty* MLF neural network model are summarized in Tab. 3.10 and the corresponding training process is shown in Fig. 3.22.

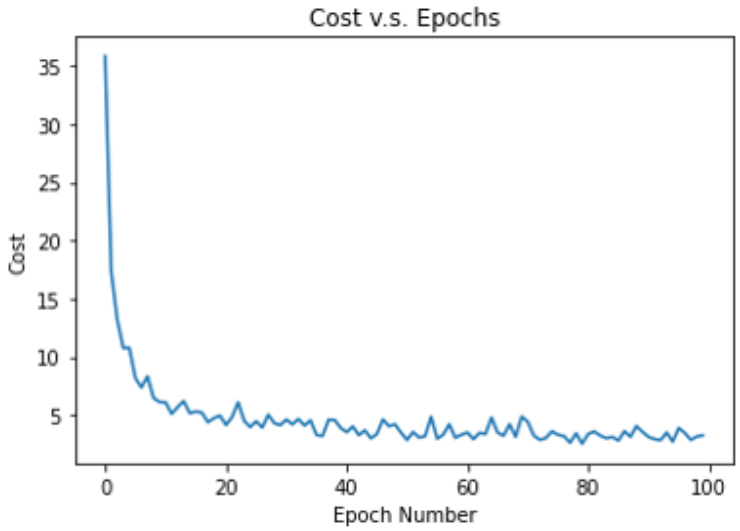


Fig. 3.21: Surrogate Large *penalty* MLF Neural Network Training Process.

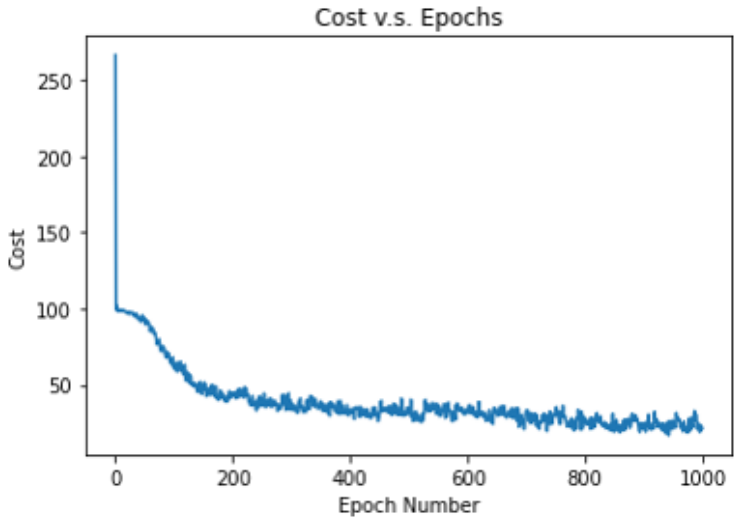


Fig. 3.22: Surrogate Small *penalty* MLF Neural Network Training Process.

Table 3.9: Hyper-parameters of the Surrogate Large *penalty* MLF Neural Network.

Hyper-parameters	**Value**
batch size	16,384
epoch	100
input size	7
hidden size 1	1024
hidden size 2	512
dropout rate	0
learning rate	0.1
weight decay	0.0001

Table 3.10: Hyper-parameters of the Surrogate Small *penalty* MLF Neural Network.

Hyper-parameters	**Value**
batch size	1,500
epoch	1000
input size	7
hidden size 1	1024
hidden size 2	512
dropout rate	0
learning rate	0.001
weight decay	0.0001

It is worth noticing that the batch size of the small penalty MLF network is around one tenth of the large penalty MLF network. This is because the dataset with penalty value smaller than 1 is of a very small portion, as indicated in Fig. 3.6. In addition, the training process of the small penalty model is more volatile than that of the large penalty model. This happens because the dataset with small penalty values are even more noisier than that with large penalty values. Nevertheless, the prediction accuracy of the penalty model can be significantly improved by training them separately.

3.3.3.4 Deep MLF Residual Neural Network

The residual networks (ResNets) alleviated problems of training very deep networks [12]. It had gotten state-of-the-art performance on the ILSVRC 2015 classification task [101] and allow training of extremely deep networks up to more than 1000 layers [102]. As shown in Fig. 3.23 [12], residual networks make use of identity shortcut connections that enable flow of information across layers without attenuation that would be caused by multiple stacked non-linear transformations, resulting in improved optimization [103].

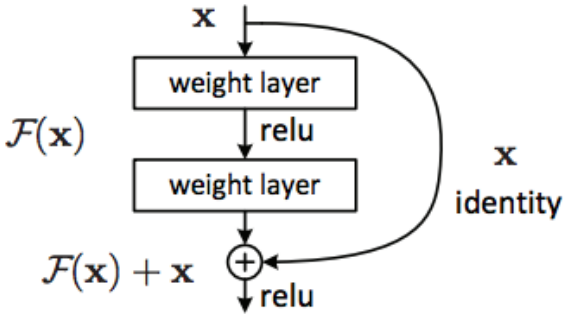


Fig. 3.23: Residual Learning: A Building Block. [12]

In this project, the surrogate model of deep MLF residual neural network is built for the $eta_I + LCOE$ model only, for the penalty model is better to be trained alone due to its randomness. Fig. 3.24 illustrates the $eta_I + LCOE$ deep MLF residual neural network, which includes four building blocks. Each block connect the initial information to the end of the building block so that the earlier information in the deep neural network could be reserved.

The hyper-parameters of the surrogate $eta_I + LCOE$ deep MLF residual neural network model are summarized in Tab. 3.11 and the corresponding training process is shown in Fig. 3.25.

It is worth noticing that the input size is 8 instead of 7 as summarized in Tab. 1.3.

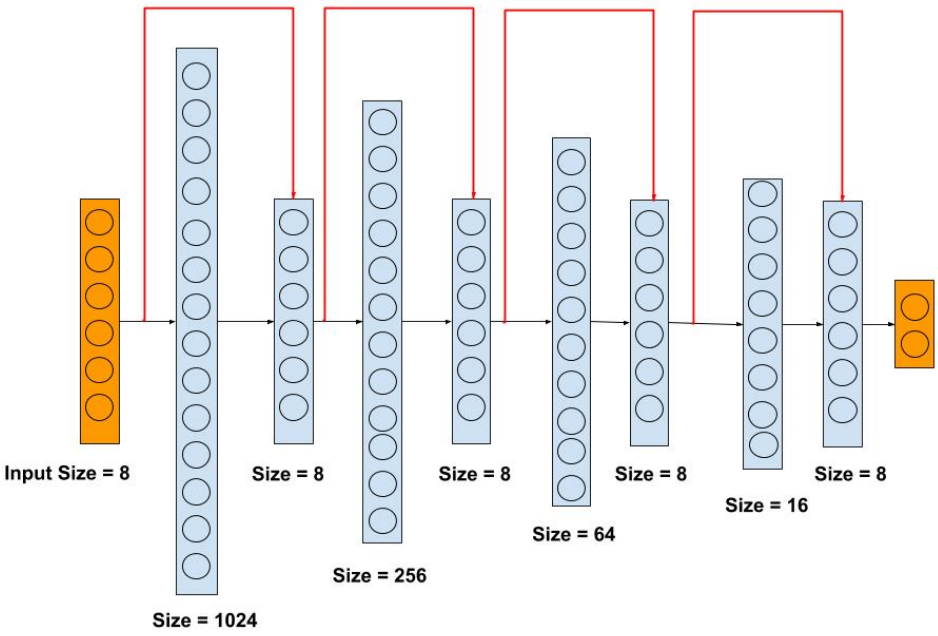


Fig. 3.24: $eta_I + LCOE$ Deep MLF Residual Neural Network.

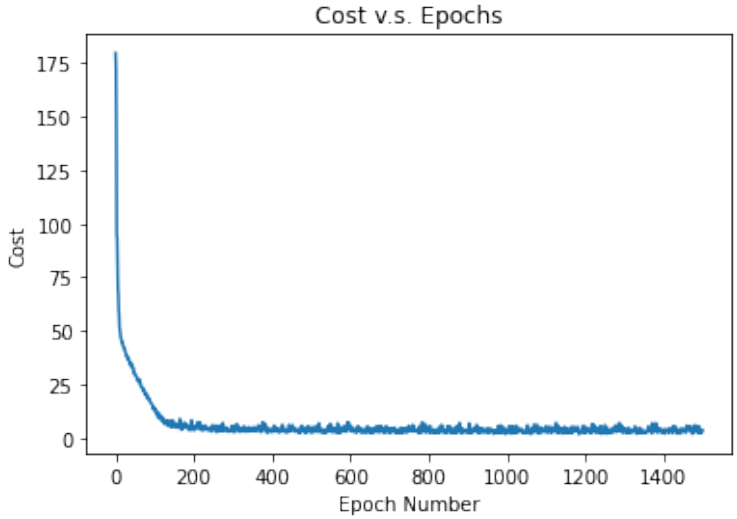


Fig. 3.25: Surrogate $eta_I + LCOE$ Deep MLF Residual Neural Network Training Process.

Table 3.11: Hyper-parameters of the Surrogate $eta_I + LCOE$ Deep MLF Residual Neural Network.

Hyper-parameters	**Value**
batch size	8,192
epoch	1,500
input size	8
hidden size 1	1,024
hidden size 2	8
hidden size 3	256
hidden size 4	8
hidden size 5	64
hidden size 6	8
hidden size 7	16
hidden size 8	8
dropout rate	0
learning rate	0.01
weight decay	0.001

This is because an additional feature, pressure after the high-pressure turbine P_1, is added by feature engineering. Feature engineering is an important but labor-intensive component of machine learning applications [104]. Most machine learning performance is heavily dependent on the representation of the feature vector. As a result, much of the actual effort in deploying machine learning algorithms goes into the design of pre-processing pipelines and data transformations [104,105]. To make use of feature engineering a model's feature vector is expanded by adding new features that are calculations based on the other features [106]. In our model, the pressure after the high-pressure turbine, P_1, is calculated from the power cycle maximum pressure, P_{max}, and the high-pressure ratio, Pr_1:

$$P_1 = P_{max} \times Pr_1$$

P_1 is an important feature and it is proved to helping the $eta_I + LCOE$ deep MLF residual neural network model be more predictably accurate. Importantly, before inputing into the

neural network, the range of P_1 is normalized into the range $[0, 1]$ by Min-Max normalization as described in Sec. 3.3.1. Moreover, the last layer of each building block of the residual network should be in the same size as the the input layer for the convenience of addition. In this case, end layer size of each building block is 8, which is the input layer size.

3.3.3.5 1-D Convolutional Neural Network

Convolutional neural networks (CNNs), as shown in Fig. 3.26 [13], are hierarchical neural networks whose convolutional layers alternate with subsampling layers, reminiscent of simple and complex cells in the human visual cortex [107], following with a fully connected layers, which are identical to multilayer perceptrons (MLP). They primarily mimic the human visual system, which can efficiently recognize the patterns and structures (e.g., objects) in a visual scenery. CNNs are now commonly used for the "deep learning" tasks, such as object recognition in large image achieves while achieving the state-of-the-art performances [108–110].

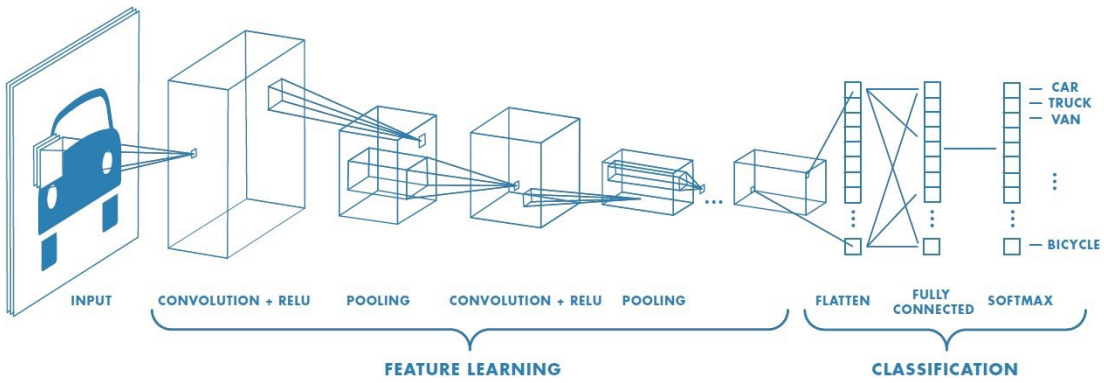


Fig. 3.26: Example of 2-D Convolutional Neural Network [13]

To my knowledge, the 1-d CNN is rarely used; however, in order to test the power of the CNN in the surrogate model, the surrogate $eta_I + LCOE$ 1-d convolutional neural network is built. The 1-d CNN share the exactly the same working mechanism as 2-d CNN, except the input is 1-d features instead of 2-d.

As indicated in Fig. 3.26, the $convolution + relu + pooling$ is one building block of

the feature learning process in the CNN networks. In the surrogate $eta_I + LCOE$ 1-d convolutional neural network model, there are two building blocks followed with the MLP consisting of two hidden layers and one output layer. The hyper-parameters of the surrogate $eta_I + LCOE$ 1-d convolutional neural network model is summarized in Tab. 3.12 and the corresponding training process is shown in Fig. 3.27. As indicated in Tab 3.12, there are lots of terminologies of the convolutional neural network (CNN). Due to the complexity of the CNNs, specific explanations of those terminologies are save to be referred at [111].

Table 3.12: Hyper-parameters of the Surrogate $eta_I + LCOE$ 1-D Convolutional Neural Network.

Hyper-parameters	**Value**
batch size	8,192
epoch	500
first convolution input channel	1
first convolution output channel	32
first convolution kernel size	4
first convolution stride size	1
first convolution padding size	0
first maxpooling kernel size	1
first maxpooling stride size	1
second convolution input channel	32
second convolution output channel	64
second convolution kernel size	2
second convolution stride size	1
second convolution padding size	0
second maxpooling kernel size	2
second maxpooling stride size	1
MLP input size	64 $\times$3
MLP hidden size 1	1024
MLP hidden size 2	512
MLP dropout rate	0
learning rate	0.01
weight decay	0.001

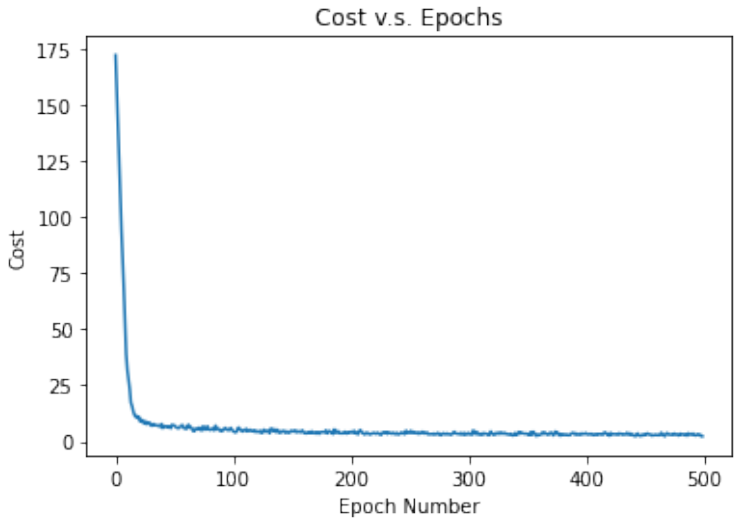


Fig. 3.27: Surrogate $eta_I + LCOE$ 1-D Convolutional Neural Network.

3.3.3.6 1-D Convolutional Residual Neural Network

As the name implies, the 1-d convolutional residual neural network is the combination of the deep MLF residual neural network and the 1-d convolutional neural network. This type of surrogate model is proposed because a deep 1-d convolutional neural network is to be built, and the ResNET is used for alleviating the problem of training the deep network.

Fig. 3.28 illustrates the surrogate $eta_I + LCOE$ 1-d convolutional residua neural network model. It consists two convolutional building blocks without maxpooling, followed with a MLP network with two hidden layers and one output layer. The MLP input layer with 64∗5 neurons is the flattened layer from the convolutional neural network. Importantly, the information of the input neurons are directly connected to the end of the convolutional neural network. The corresponding hyper-parameters of the model is summarized in Tab. 3.13, and the training process is shown in Fig. 3.29.

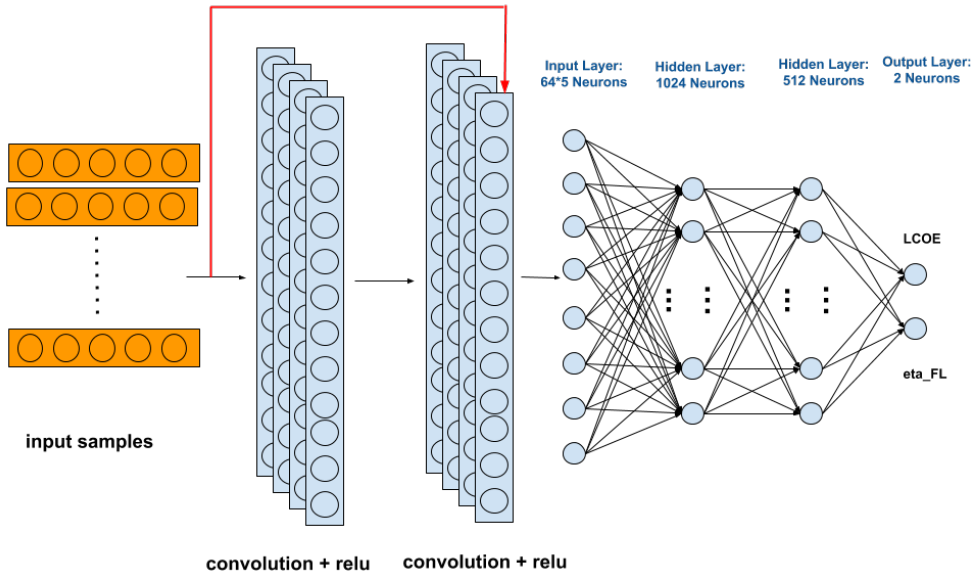


Fig. 3.28: Surrogate $eta_I + LCOE$ 1-D Convolutional Residua Neural Network.

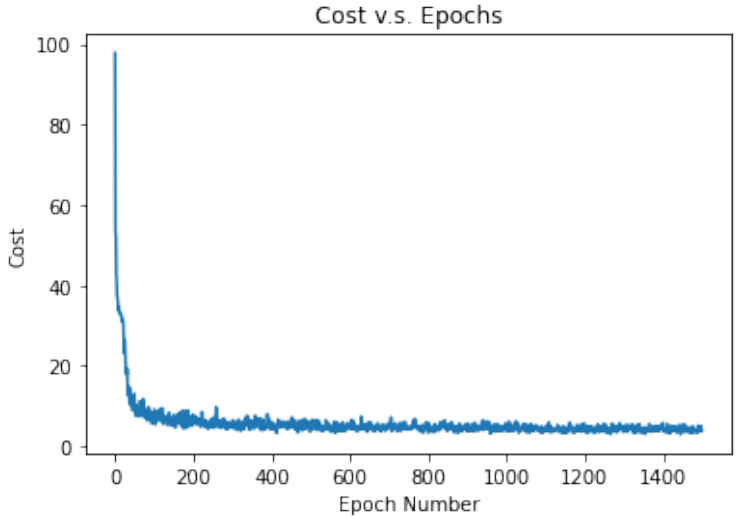


Fig. 3.29: Surrogate $eta_I + LCOE$ 1-D Convolutional Residual Neural Network.

Table 3.13: Hyper-parameters of the Surrogate $eta_I + LCOE$ 1-D Residual Convolutional Neural Network.

Hyper-parameters	**Value**
batch size	8,192
epoch	1,500
first convolution input channel	1
first convolution output channel	256
first convolution kernel size	3
first convolution stride size	1
first convolution padding size	1
second convolution input channel	256
second convolution output channel	64
second convolution kernel size	3
second convolution stride size	1
second convolution padding size	1
MLP input size	64 $\times$5
MLP hidden size 1	1024
MLP hidden size 2	512
MLP dropout rate	0
learning rate	0.1
weight decay	0.0001

CHAPTER 4

RESULTS

4.1 Converged/Diverged Classifier Models & Surrogate Models Comparison

In this section, the best converged/diverged classifier model and the best surrogate model will be selected through metrics-based comparison.

4.1.1 Converged/Diverged Classifier Model Comparison

As described in Sec. 3.3.2, six classifier models are built, and the best-performed one will be selected according to the commonly used evaluation metrics for binary classification models: accuracy, precision, recall (also called sensitivity) and f1 score [112]. The classifier model with the highest score wins the corresponding metric, and the one that wins the most of metrics is going to be selected.

Accuracy is the fraction of predictions that are true. Although this metric is easy to interpret, high accuracy does not necessarily characterize a good classifier. For instance, it tells us nothing about whether False Negatives (FNs) or False Positives (FPs) are more common. Similarly, both True Positives (TPs) and False Positives (FPs) are captured by precision (also called the positive predictive value), which is the proportion of predicted positives that are correct. However, precision captures neither True Negatives (TNs) nor False Negatives (FNs). A useful measure for understanding FNs is recall (also called sensitivity or the true positive rate), which is the proportion of known positives that are predicted correctly. However, neither TNs nor FPs affect this metric, and a classifier that simply predicts that all data points are positive has high recall [112]. F1 Score might be a better measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of TNs). F1 Score is calculated from [113]:

$$f_1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

In this study, the four metrics are used all together for the converged/diverged classifier models. The corresponding results are summarized in Tab. 4.1.

Table 4.1: Converged/Diverged Classifier Models Evaluation Results.

Classifier Model Name	**Accuracy** [-]	**Precision** [-]	**Recall** [-]	**F1 Score** [-]
MLF Neural Network	0.991	0.995	0.983	0.992
Random Forest (RF)	0.988	0.996	0.980	0.988
Gaussian Naïve Bayes (NB)	0.819	0.835	0.808	0.821
K Nearest Neighbor (KNN)	0.869	0.999	0.804	0.891
Logistic Regression (LR)	0.732	0.752	0.723	0.737
Support Vector Machine (SVM)	0.732	0.758	0.719	0.738

As indicated in Tab. 4.1, The MLF Neural Network wins the accuracy, recall and the f1 score metrics, and got just a tiny behind of the Random Forest model in the precision metric. Thus, the MLF Neural Network with the schematics summarized in Tab. 3.1 is selected as the best classifier model.

4.1.2 Surrogate Model Comparison

As described in Sec. 3.3.3, five surrogate models are built and the best-performed one is selected according to the commonly-used evaluation metrics for neural networks: r-squared (r2) score, relative absolute error (RAE) and L1 loss. The surrogate model with the highest value of r2 score wins the r2 metric, while the one with the lowest value of relative absolute error (RAE) wins the RAE metric and the one with the lowest value of L1 loss wins the L1 metrics. Finally, the surrogate model that wins the most of the three metrics is going to be selected.

r2 score is a measurement that provides information about the goodness of fit of a model [114]:

$$R^2 = 1 - \frac{sum\ squared\ regression\ (SSR)}{total\ sum\ of\ squares\ (SST)} = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2} \tag{4.1}$$

Where y is the actual value; $\hat{y}$ is the predicted value; and $\bar{y}$ is the mean of the actual y value. As indicated in Eq. 4.1, r2 score is the proportion of the variance in the dependent variable that is predictable from the independent variable(s) [115]. Best possible r2 score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0 [116].

Relative absolute error (RAE) is a way to measure the performance of a predictive model. It's primarily used in machine learning, data mining, and operations management. The Relative Absolute Error is expressed as a ratio, comparing a mean error (residual) to errors produced by a trivial or naive model. A reasonable model (one which produces results that are better than a trivial model) will result in a ratio of less than one [117]. RAE is calculated from [68]:

$$RAE = \left(\frac{1}{n} \sum_{i=1}^{n} \left| \frac{\hat{y}_i - y_i}{y_i} \right| \right) \cdot 100\% \tag{4.2}$$

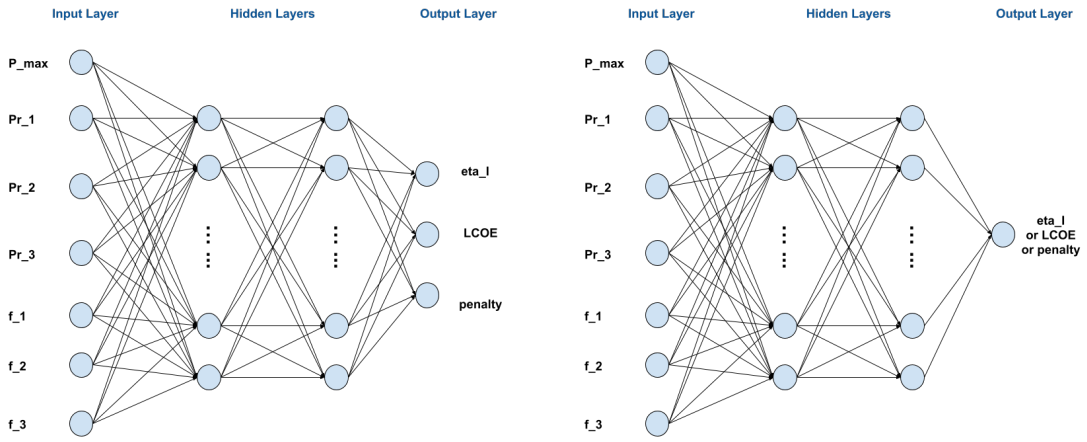
L1 loss is a type of loss evaluation of the neural network calculated from the absolute error of the outputs. It is shown that the quality of the results improves significantly with better loss functions, even when the network architecture is left unchanged [118]. There are in general two types of loss functions, L1 and L2. Typically, L2 loss function penalizes large errors, but is more tolerant to small errors; while L1 loss function does not over-penalize larger errors, and it is calculated from [118]:

$$L1 = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i| \tag{4.3}$$

For the completeness, the L2 loss function is calculated from [119]:

$$L2 = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{4.4}$$

In general, R-squared (r2) score measures the strength of the relationship between the model and the dependent variable, and it is normally used as a goodness-of-fit measure for regression models [120]. Relative Absolute Error (RAE) is a way to measure the performance



(a) Integrated Model. (b) Separate Model

Fig. 4.1: Integrated and Separate Models.

of a predictive model, and it is primarily used in fields of machine learning, data mining and operation management [121]. Particularly, in the field of machine learning, RAE is used in models where the minimal of relative errors between the predicted value and the true true is primarily wanted. L1 loss function is used to minimize the error which is the sum of all the absolute differences between the true value and the predicted value in a model [119]. Particularly, when the differences in the model are not supposed to be over-penalized, the L1 loss function is used.

In this study, the five surrogate models are evaluated on all three metrics for completeness. Specifically, the separate model is evaluated slightly differently from the integrated model. The separated model was built to explore if the model's prediction capabilities can be improved by training each predicted targets separately with their own optimum network architectures. As shown in Fig. 4.1, the separate model has only one output, which can be either eta_I or $LCOE + penalty$; while in the integrated models, all the outputs are trained together. In this project, there are two networks in the separate model for eta_I, $LCOE + penalty$, respectively. Thus, in order to compare the metric values of the separate models with ones of the other integrated models in a fair way, the values of each metrics in r2 score, relative absolute error (RAE) and L1 loss for the model are calculated from the

average value of each output, as indicated in Eq. 4.5:

$$M_{avg} = \frac{1}{2}(M_{eat\_I} + M_{LCOE+penalty}) \tag{4.5}$$

where M refers to the type of metrics to be calculated in the separate model. For example, if the relative absolute error (RAE) of the separate model is to be calculated, it would be calculated from the average RAE scores of the two networks of eta_I and $LCOE + penalty$:

$$RAE_{avg} = \frac{1}{2}(RAE_{eat\_I} + RAE_{LCOE+penalty}) \tag{4.6}$$

It is worth noting that similar equations apply to the surrogate models with the separate *penalty* models.

Finally, the results of the surrogate models obtained from their corresponding testing and validation dataset are summarized in Tab. 4.2. As indicated in the table, since the Separate MLF Neural Network wins all the metrics, it is selected as the best surrogate model along with the schematics summarized in Tab. 3.7 and Tab. 3.8

Table 4.2: Surrogate Models Evaluation Results.

Surrogate Model Name	**R2 Score [-]**	**RAE [%]**	**L1 Loss [-]**
MLF Neural Network	0.966	6.77	10.418
Separate MLF Neural Network	0.987	2.37	1.903
Deep MLF Residual Neural Network	0.956	6.10	3.448
1-D Convolutional Neural Network	0.984	3.68	3.154
1-D Convolutional Residual Neural Network	0.970	5.82	5.071

4.2 Optimizers Comparison

In this section, based on the best converged/diverged classifier model and surrogate model selected, the six optimizers described in Sec. 3.2 are compared by executing each of them with the final-chosen surrogate model as shown in Fig. 3.2. Due to the stochas-

tic nature of most optimizers, the surrogate model is run as the core component in the surrogate-optimizer model as shown in Fig. 3.1. As the results, the optimizer capable of finding the set of design parameters that could produce the lowest LCOE and penalty values in the physics-based model wins. The corresponding results are summarized in Tab. 4.3.

Table 4.3: Optimizer & Overall Results.

Optimizer Name	**LCOE [$/MWh]**	**Penalty [-]**	eta_I [%]	**Run Time [s]**
Basin-Hopping	78.876	0.006	30.54	49
Brute Force	78.108	0.4639	31.68	70
Differential Evolution	-	-	-	-
Shgo	88.455	4.051	32.40	95
Dual Annealing	77.912	0.190	31.01	23
Fmin	81.320	0.774	32.21	41

As shown in Tab. 4.3, for some unknown reason, the overall algorithm has a hard time converging at all with the differential evolution optimizer. Other than that, it is obvious that the Dual Annealing optimizer wins the $LCOE$ and *run time*, while Basin-Hopping optimizer wins the *penalty* with the $LCOE$ value just a little behind the $LCOE$ value of the Dual Annealing optimizer. Meanwhile, the Shgo optimizer wins the first law efficient, eta_I, with unacceptable $LCOE$ value, *penalty* value and the *run time*. Since the lowest $LCOE$ and *penalty* values are expected for the overall system, the Dual Annealing optimizer and the Basin-Hopping optimizer are both selected as the optimal optimizers.

CHAPTER 5
DISCUSSION

In this chapter, the final results from the physics-based model with the optimized design parameters from the surrogate-optimizer model as input are compared with the baseline results from the physics-based model, and the results directly from the surrogate model are compared with the baseline results and the final results.

Specifically, the Tab. 5.1 summarizes the baseline design parameters from the physics-based model, and the optimized design parameters from the surrogate model with the dual-annealing (w/ DA) optimizer and with the basin-hopping (w/ BH) optimizer, respectively. The lower section of the table shows the relative errors between the optimized design parameters from surrogate model (w/ DA) and the baseline design parameters from the physics-based model, and the relative errors between the optimized design parameters from surrogate model (w/ BH) and the baseline design parameters from the physics-based model, respectively. Furthermore, In Tab. 5.2, with the optimized design parameters from the surrogate (w/ DA) and from the surrogate model (w/ BH) as input to the physics-based model, the corresponding outputs of LCOE, penalty and the first-law efficiency values are summarized along with the baseline output results from the physics-based model. The lower section of the table shows the relative errors between the outputs with the DA-optimized design parameters and the baseline outputs from the physics-based model, and the relative errors between outputs with the BH-optimized design parameters and the baseline design parameters from the physics-based model, respectively.

As shown in Tab. 5.2, the *LCOE* relative error (RE) with design parameters from surrogate (w/ DA) is 0.94%, and the RE with design parameters from surrogate (w/ BH) is 2.18%, which are both relatively small. In addition, the *penalty* value absolute error (AE) with design parameters from surrogate (w/ DA) is 0.0936, and the AE with design parameters from surrogate (w/ BH) is -0.0904, which are both small. It is worth noticing that the

Table 5.1: Optimized Design Parameters Comparison with the Baseline.

Design Parameters	**baseline**	surrogate model (w/ DA)	surrogate model (w/ BH)
$\mathbf{Pr_{max}}$ [MPa]	**8.22**	8.31	8.36
$\mathbf{Pr_1}$ [-]	**0.0505**	0.0540	0.0491
$\mathbf{Pr_2}$ [-]	**0.5246**	0.4999	0.4667
$\mathbf{Pr_3}$ [-]	**0.3595**	0.2763	0.7744
$\mathbf{f_1}$ [-]	**0.0140**	0.0100	0.0410
$\mathbf{f_2}$ [-]	**0.0665**	0.0941	0.0167
$\mathbf{f_3}$ [-]	**0.0788**	0.0699	0.1001
$\mathbf{P_{max}}$ **RE** [%]	**-**	1.09	1.70
$\mathbf{Pr_1}$ **RE** [%]	**-**	6.93	-2.77
$\mathbf{Pr_2}$ **RE** [%]	**-**	-4.71	-11.04
$\mathbf{Pr_3}$ **RE** [%]	**-**	-23.14	115.41
$\mathbf{f_1}$ **RE** [%]	**-**	-28.57	192.86
$\mathbf{f_2}$ **RE** [%]	**-**	41.50	-74.89
$\mathbf{f_3}$ **RE** [%]	**-**	-11.29	27.03

Table 5.2: Results Comparison with the Baseline.

Results	**baseline**	surrogate model (w/ DA)	surrogate model (w/ BH)
LCOE [$/MWh]	**77.190**	77.912	78.876
Penalty [-]	**0.0964**	0.1900	0.0060
eta_I [%]	**30.77**	31.01	30.54
LCOE RE [%]	**-**	0.94	2.18
Penalty AE [-]	**-**	0.0936	-0.0904
eta_I RE [%]	**-**	0.78	-0.75

absolute error (AE) instead of the relative error (RE) is used for penalty comparison. This is because the training dataset of the penalty is more like random noise and it is trained with the AE cost function instead of RE. In addition, since a near 0 value is expected for penalty and the baseline penalty value is already small enough, if the relative error were calculated with the baseline value as the denominator, even the actual difference between the optimized penalty and the baseline penalty were very small, the RE value would be

very huge, which doesn't reflect the actual performances of the surrogate-optimizer model. Lastly the *eta_I* relative error (RE) with design parameters from surrogate (w/ DA) is 0.78%, and the RE with design parameters from surrogate (w/ BH) is −0.75%, which are both very small. In summary, the surrogate model with the Dual Annealing optimizer (w/ DA) outperforms the physics-based simulation model in terms of the the *eta_I* design parameters optimization, with the *LCOE* design parameters optimization and *penalty* design parameters optimization just a tiny behind the ones of the physics-based simulation model. Meanwhile, the surrogate model with the Basin-Hopping optimizer (w/ BH) outperforms the physics-based simulation model in terms of the the *penalty* design parameters optimization, with the *LCOE* design parameters optimization and *eta_I* design parameters optimization a little behind the ones of the physics-based simulation model.

Furthermore, from Tab. 5.1, it can be observed that, in general, the baseline design parameters and the optimized design parameters are in the similar range, with the surrogate model with DA optimizer slightly outperforms the surrogate model with BH optimizer in term of the average relative errors.

Table 5.3: Direct Results with Dual-Annealing (DA) optimizer Compared with the Baseline and DA Final Results.

Results	**direct surrogate model**	baseline	surrogate model
LCOE+penalty [-]	**78.3189**	77.2864	78.1020
eta_I [%]	**33.10**	30.77	31.01
LCOE+penalty RE [%]	**-**	1.3359	0.2777
eta_I RE [%]	**-**	7.57	6.74

Moreover, Tab. 5.3 and Tab. 5.4 summarizes the optimized results directly from the surrogate model (direct surrogate model), and they are compared with the baseline results (baseline) and with the final results that are from the physics-based model with the optimized design parameters as inputs (surrogate model). It is worth noticing that the added LCOE and penalty values are compared. That is because in the chosen surrogate model, LCOE and penalty are trained together as one output, so that when the optimized design

Table 5.4: Direct Results with Basin-Hopping (BH) optimize Compared with the Baseline and BH Final Results.

Results	**direct surrogate model**	baseline	surrogate model
LCOE+penalty [-]	**78.0661**	77.2864	78.8820
eta_I [%]	**32.49**	30.77	30.54
LCOE+penalty RE [%]	**-**	1.0088	-1.0343
eta_I RE [%]	**-**	5.59	6.39

parameters are put into the surrogate LCOE+penalty model, the outcome would directly be the corresponding optimized LCOE+penalty value, as summarized in Sec. 3.3.3.2.

Specifically, in Tab. 5.3, with the optimized design parameters from the surrogate-DA-optimizer model as inputs, as summarized in Tab. 5.1, the corresponding results directly from the surrogate-DA-optimizer model (direct surrogate model) is compared with the baseline results (baseline) and with the corresponding results from the physics-based model with the optimized design parameters as inputs (surrogate model). It can be observed that the first law efficiency value from the direct surrogate model is 7.57% higher that from the baseline, and it is 6.74% higher than that from the surrogate model. Meanwhile, the $LCOE+penalty$ value from the direct model is 1.3359% higher that from the baseline, and it is 0.2777% higher than that from the surrogate model. It can be concluded that, in terms of the first law efficiency result, the direct surrogate model way outperforms the physics-based model and the surrogate-DA-optimizer model; however, in terms of the $LCOE + penalty$ result, the direct surrogate model is slightly outperformed by the physics-based model and the surrogate-DA-optimizer model, although not too much.

In Tab. 5.4. with the optimized design parameters from the surrogate-BH-optimizer model as inputs, as summarized in Tab. 5.1, the corresponding results directly from the surrogate-BH-optimizer model (direct surrogate model) is compared with the baseline results (baseline) and with the corresponding results from the physics-based model with the optimized design parameters as inputs (surrogate model). It can be observed that the first law efficiency value from the direct surrogate model is 5.59% higher that from the baseline, and it is 6.39% higher than that from the surrogate model. Meanwhile, the $LCOE+penalty$

value from the direct model is 1.0088% higher that from the baseline, and it is 1.0343% lower than that from the surrogate model. It can be concluded that, in terms of the first law efficiency result, the direct surrogate model again way outperforms the physics-based model and the surrogate-BH-optimizer model; however, in terms of the $LCOE + penalty$ result, the direct surrogate model is slightly outperformed by the physics-based model, but slightly outperforms the surrogate-BH-optimizer model.

In summary, in terms of the first law efficiency, the direct surrogate model produces the optimal value, with the optimized design parameters from either the surrogate-DA-optimizer model or the surrogate-BH-optimizer model. In addition, in terms of the $LCOE + penalty$, the direct surrogate model normally produces the less optimal values in comparison with the baseline and surrogate models, no matter with the optimized design parameters from either the surrogate-DA-optimizer model or the surrogate-BH-optimizer model. In this specific study, only the $LCOE + penalty$ value from the direct surrogate model with BH optimizer slightly outperforms the one from the surrogate-BH-optimizer model.

In general, the ML-based surrogate model is proved to satisfy high-accuracy for predicting outputs with new input values, while only consuming small amount computational power. In this study, it took around 100 hours to find the optimized baseline design parameters and corresponding results in Tab. 5.1 and Tab. 5.2 with the physics-based simulation model; while it took only under 10 minutes with the surrogate-optimizer models. Specifically, the fact that the separate model outperforms the integrated model implies the relation between inputs and outputs of the samples in the dataset are different for the $LCOE + penalty$ and the eta_I models. As indicated in the parametric studies in Sec. 1.2, Eta_I demonstrates clear trends with the seven design parameters; while the $penalty$ and $LCOE$ fluctuates much more, particularly with the the $penalty$ fluctuates almost randomly as the increase of seven design parameters. For this reason, by building ML-based surrogate models for $LCOE + penalty$ and the eta_I separately, with their respective network structure and weight parameters optimized, gives the separate model more accurate prediction capability than that of the integrated model. In addition, in comparison with the integrated

model, where $LCOE, penalty, eta\_I$ are trained in one network, the separate model almost doubled the amount of weight parameters overall, which also could be the reason why the separate model performs better.

For optimization, unlike the ANN-GA methods chosen by [36], [38] and etc., in this study, the basin-hopping and dual annealing algorithms are selected as the best optimizer. According to Tab. 4.3, the differential evolution genetic optimizer has a hard time converging for unknown reasons. It is Hypothesised that the hyperparameters, such as the mutation rate or the recombination ratio, are not chosen properly so that the optimizer searches slowly at the beginning; or the hyperparameters caused the searching dynamics too erratic so that searching process is basically jumping around randomly within the domains. In contrast, the basin-hopping algorithm takes advantages of local searches heavily while leveraging the global searches by perturbation; and the dual annealing algorithm combines the generalization of CSA (Classical Simulated Annealing) and FSA (Fast Simulated Annealing) [59,60] coupled to a strategy for applying a local search on accepted locations [61]. Both algorithms carefully reached a balance between searching too erratically so that being kept bouncing around and searching too locally so that being trapped in a local trough. Within the rest of the optimizers, Shgo algorithm has the worst performance in terms of both optimization capability and time consumption. The reason is hypothesised to be the over-complexity of the Shgo algorithm itself, so that more computational time is needed and more noises are added during the optimization procedure. On the other hand, the brute force and Fmin algorithms show moderate performances, with the brute force showing better optimization capabilities and with the Fmin showing better time efficiency. The brute force is not time efficient to use for big searching tasks. For example, in this study, even though the brute force algorithm produced optimized design parameters that result in good $LCOE$ and Eta_I values, the time consumed is relatively high, while the penalty is outside the range of acceptability. If the penalty is to be reduced, much precise steps need to be applied in the brute force algorithm for finding a better set of design parameters, which would cause exponential increase of the computing time. However, comparing

with using the physics-based model along with the brute search optimizer, the surrogate model helps accelerate the searching process significantly. This obviously violates the initial purpose of applying the surrogate model. The Fmin algorihtm is time-efficient to use, and the *penalty* and Eta_I values resulted from the optimized design parameters it searched are both acceptable. However, the LCOE value is not that optimal, but still acceptable. If more initial searching points are given, Fmin could reach a better optimization capability while sacrificing little time efficiency.

During the development of the program, some obstacles were confronted. Besides, the differential evolution non-converging issue, the training of the *penalty* was also hard, no matter in $LCOE + penalty + eta\_I$ network, in $LCOE + penalty$ network or in *penalty* alone network. As indicated in Tab. 5.2, in comparing the optimized results with the baseline results, the relative errors (RE) of both the Eta_I and $LCOE$ are calculated, but the *penalty* is compared in terms of the absolute error (AE). This is because the *penalty* value predicted from the ML-based surrogate model is trained with the absolute error instead of the relative error. It was found that training dataset of the *penalty* are more like random noises, so that whenever it is trained with the relative error, the error never goes down when it reaches around 200%. For this reason, the absolute error was applied as the loss function in training the $LCOE + penalty$. More efforts on tackling this issue is left for the future work.

CHAPTER 6
CONCLUSION

In this study, a fast surrogate-optimizer model is built to accelerate the optimization process in searching for the design parameters of a integrated regenerative methanol transcritical cycle. Specifically, several machine-learning based surrogate models are built in replacing the physics-based simulation model of the cycle, and the one with the best synthesized evaluation score of $R2$, relative absolute error (RAE) and $L1$ loss is selected as the final model. Prior to the surrogate model, a machine learning-based converged/diverged classifier model is also built to filter out the non-converging input samples of design parameters. Lastly, a optimizer is carefully selected to work along with the classifier and surrogate models in searching for the set of design parameters that could result in optimized $LCOE$, *penalty* and *eta_I* values in the physics-based model.

As the results, the separate Multi-Layer Feedforward (MLF) neural network outperforms all other surrogate models with $R2$ score of 0.987, RAE value of 2.37% and $L1$ loss value of 1.903. Moreover, by executing all the candidate global/semi-global optimizers along with the final-chosen separate MLF neural network surrogate model and the classifier model, it indicates that the the dual-annealing (DA) and the basin-hopping (BH) optimizers produce design parameters that correspond to the best-performed $LCOE$, *penalty* and *eta_I* values from the physics-based model by passing those design parameters into it. Specifically, the $LCOE$ relative absolute error (RAE) corresponding to the DA-optimized design parameters is 0.94%, and the $LCOE$ RAE corresponding to the BH-optimized design parameters is 2.18%; the *penalty* absolute error (AE) corresponding to the DA-optimized design parameters is 0.0936, and the *penalty* RA corresponding to the BH-optimized design parameters is -0.0904; the *eta_I* relative absolute error (RAE) corresponding to the DA-optimized design parameters is 0.78%, and the *eta_I* RAE corresponding to the BH-optimized design parameters is -0.75%.

In summary, there are in total two top surrogate-optimizer model combinations: separate MLF neural network surrogate model + dual-annealing (DA) optimizer, separate MLF neural network surrogate model + basin-hopping (BH) optimizer. Both of them are capable of searching for the optimized design parameters that results in comparable $LCOE$, *penalty* and *eta_I* values from the physics-based model in comparison with the baseline values. Particularly, in terms of the *penalty* value, the results from the surrogate model with the basin-hopping (BH) optimizer slightly outperforms the baseline result; in terms of the *eta_I* value, the results from the surrogate model with the dual-annealing (DA) optimizer slightly outperforms the baseline result; and in terms of the $LCOE$ value, both surrogate-optimizer models are slightly outperformed by the baseline result, with the surrogate model with the dual-annealing (DA) optimizer performs a bit better than the other surrogate-optimizer model.

In building the surrogate-optimizer models, there are three persisting issues to be addressed:

1. Obstacles in training the *penalty* in the related neural networks: the training dataset for *penalty* is more like random noises, and it happens due to the complex nature of the simulated system. Also, it is worth noting that the largest source of penalty is due to fluid in the the pumps, even at the identified optima.

2. The non-converging problem with the differential evolution (DE) optimizer: whenever the optimizer in the surrogate-optimizer model is the DE optimizer, it becomes extremely hard for the surrogate-optimizer model to converge. One hypothesis is that the hyper parameters of the DE optimizer, such as the mutation ratio, are not adjusted to the optima in adapt to the complex surrogate model.

3. Instability of the identified optima: for the converged optimal design parameters found from the surrogate-optimizer models, if the values of them are rounded up a little bit, it would become non-converging in the physics-based model.

For the future work, the issues mentioned above are expected to be addressed. Firstly, for making the *penalty* easier to be trained in the neural network, its dataset generated from the physics-based model should be less noisier. For example, since the fluid in pumps are not desired, future work could seek to more heavily penalize these terms. Also, the pump/regenerator solver in the physics-based model can be reconstructed to explore the possibility of solving for a regenerator that will keep the fluid at the pump inlet subcooled. Moreover, changing the way to define the penalty may also make the dataset easier to be interpreted. Secondly, for reducing the complexity of Machine Learning (ML)-based surrogate model, hybrid physics/ML model may be considered, meaning the ML-based model can be used to approximate just the most time-consuming parts of the physics-based model. This could potentially reduce runtime compared to the full physics-based model and error compared to the full ML-based surrogate model. Lastly, more in-depth exploration on why the instability happens at the optima are expected. For example, parametric studies can be carried out at different sets of the optimal design parameters.

REFERENCES

[1] LLC NuScale Power. How the nuscale module works.

[2] Kelly Fleetwood. An introduction to differential evolution.

[3] Kemal H. Sahin and Amy R. Ciric. A dual temperature simulated annealing approach for solving bilevel programming problems.

[4] DanielSvozil, Vladimir KvasniEka, and JiE Pospichal. Introduction to multi-layer feed-forward neural networks.

[5] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting.

[7] . Random forest.

[8] alex_abhi43. Gini impurity and entropy in decision tree – ml, 2020. GreeksforGreeks.

[9] Madison Schott. K-nearest neighbors (knn) algorithm for machine learning, 2019.

[10] Ashish Anand. What is the difference between linear regression and logistic regression?, 2019.

[11] Nikita Sharma. Understanding the mathematics behind support vector machines, 2020.

[12] Connor Shorten. Introduction to resnets, 2019.

[13] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way, 2018.

[14] Jiangfeng Wang, Zhixin Sun, Yiping Dai, and Shaolin Ma. Parametric optimization design for supercritical co2 power cycle using genetic algorithm and artificial neural network.

[15] Alessandro Massimiani, Laura Palagi, Enrico Sciubba, and Lorenzo Tocci. Neural networks for small scale orc optimization.

[16] Laura Palagi, Enrico Sciubba, and Lorenzo Tocci. A neural network approach to the combined multi-objective optimization of the thermodynamic cycle and the radial inflow turbine for organic rankine cycle applications.

[17] VIRGINIA TORCZON. On the convergence of pattern search algorithms.

[18] Dirk Gorissen, Ivo Couckuyt, Piet Demeester, Tom Dhaene, and Karel Crombecq. A surrogate modeling and adaptive sampling toolbox for computer based design.

[19] D.T.Ingersoll, Z.J.Houghton, R.Bromm, and C.Desportes. Nuscale small modular reactor for co-generation of electricity and water. 340:84–93, 2014.

[20] Yili Zhang, Hailei Wang, Sean Kissick, and Derick Botha. System modeling of an integrated organic regenerative transcritical cycle with a small modular light water reactor.

[21] Venkatesh Dasari. What is a pinch point in a heat exchanger?, 2017.

[22] Rohit K. Tripathy and Ilias Bilionis. Deep uq: Learning deep neural network surrogate models for high dimensional uncertainty quantification.

[23] M. Balesdent, L. Brevault, S. Lacaze, S. Missoum, and J. Morio. Methods for high-dimensional and computationally intensive models.

[24] M M Rashidi, O Anwar Beg, A Basiri Parsa, and F Nazari. Analysis and optimization of a transcritical power cycle with regenerator using artificial neural networks and genetic algorithms.

[25] Jinxing Zhao, Min Xu, Mian Li, Bin Wang, and Shuangzhai Liu. Design and optimization of an atkinson cycle engine with the artificial neural network method.

[26] M.A. José and A. Fernando. Combining neural networks and genetic algorithms to predict and reduce diesel engine emissions.

[27] Xiaoshun Zhang, Bo Yang, Tao Yu, and Lin Jiang. Dynamic surrogate model based optimization for mppt of centralized thermoelectric generation systems under heterogeneous temperature difference.

[28] R. G. Regis. Evolutionary programming for high-dimensional constrained expensive black-box optimization using radial basis functions.

[29] Wahid Ali, Mohd Shariq Khan, Muhammad Abdul Qyyuma, and Moonyong Lee. Surrogate-assisted modeling and optimization of a natural-gas liquefaction plant.

[30] Mustafa Berke Yelten, Ting Zhu, Slawomir Koziel, Paul D. Franzon, and Michael B. Steer. Demystifying surrogate modeling for circuits and systems.

[31] Mazhar F, Khan AM, and Chaudhry IA. On using neural networks in uav structural design for cfd data fitting and classification.

[32] Gang Sun and Shuyue Wang. A review of artificial neural network surrogate modelling in aerodynamic design.

[33] Seung-Woo Kim, Anzy Lee, and Jongyoon Mun. A surrogate modeling for storm surge prediction using an artificial neural network.

[34] RAVIPUDI V. RAO, ANKIT SAROJ, PAWEL OCLON, JAN TALER, , and JAYA LAKSHMI. A posteriori multiobjective self-adaptive multipopulation jaya algorithm for optimization of thermal devices and cycles.

[35] Michael D. Vose. *The Simple Genetic Algorithm: Fundation and Theory.* The MIT Press, Cambridge, Massachusetts, 1971.

[36] M.V.J.J.Suresh, K.S.Reddy, and Ajit KumarKolar. Ann-ga based optimization of a high ash coal-fired supercritical power plant.

[37] Delft University of Technology. Cycle-tempo release 5.0, 2007.

[38] Farzaneh Hajabdollahi, Zahra Hajabdollahi, and Hassan Hajabdollahi. Soft computing based multi-objective optimization of steam cycle power plant using nsga-ii and ann.

[39] Arash Jamali, Pouria Ahmadi, Mohammad Nazri, and Mohd Jaafar. Optimization of a novel carbon dioxide cogeneration system using artificial neural network and multi-objective genetic algorithm.

[40] I.Y. Kim and O.L. de Weck. Adaptive weighted-sum method for bi-objective optimization: Pareto front generation.

[41] Cybenko G. Approximation by superpositions of a sigmoidal function.

[42] Bishop C.M. Neural networks and their applications.

[43] MATLAB. fmincon active set algorithm., 2015.

[44] MATLAB. version 8.5.0.197613 (r2015a), 2015.

[45] Fubin Yang, Heejin Cho, Hongguang Zhang, Jian Zhang, and Yuting Wu. Artificial neural network (ann) based prediction and optimization of an organic rankine cycle (orc) for diesel engine waste heat recovery.

[46] Laura Palagi, Apostolos Pesyridis, Enrico Sciubba, and Lorenzo Tocci. Machine learning for the prediction of the dynamic behavior of a small scale orc system. 166:72–82, 2019.

[47] Akshay J. Dave, Jarod Wilson, and Kaichao Sun. Deep surrogate models for multi-dimensional regression of reactor power. pages 1–4, 2020.

[48] David J. Wales and Jonathan P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms.

[49] Stefan Kühn. Mathematical optimization: What is the basin hopping algorithm?, 2018.

[50] David J. Wales. *Energy Landscapes.* Cambridge University Press, Cambridge, UK, 2003.

[51] Zhenqin Li and Harold A. Scheraga. Monte carlo-minimization approach to the multiple-minima problem in protein folding.

[52] David J. Wales and Harold A. Scheraga. Global optimization of clusters, crystals, and biomolecules. 285:1368–1372, 1999.

[53] SciPy. scipy.optimize.basinhopping.

[54] Anthony B. Morton and Iven M. Y. Mareels. An efficient brute-force solution to the network reconfiguration problem.

[55] SciPy. scipy.optimize.brute.

[56] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces.

[57] Stefan C. Endres, Carl Sandrock, and Walter W. Focke. A simplicial homology algorithm for lipschitz optimisation.

[58] Xiang Y, Sun DY, Fan W, and Gong XG. Generalized simulated annealing algorithm and its application to the thomson model.

[59] Tsallis C. Possible generalization of boltzmann-gibbs statistics.

[60] Tsallis C and Stariolo DA. Generalized simulated annealing.

[61] Xiang Y and Gong XG. Efficiency of generalized simulated annealing.

[62] SciPy. scipy.optimize.dual_annealing.

[63] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi Jr. Optimization by simulated annealing. 220:671–680, 1983.

[64] Frank Liang. Optimization techniques simulated annealing, 2020.

[65] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the nelder-mead simplex method in low dimensions.

[66] MATLAB. fminsearch algorithm.

[67] MATLAB. fminsearch.

[68] Alessandro Massimiani, Laura Palagi, Enrico Sciubba, and Lorenzo Tocci. Neural networks for small scale orc optimization.

[69] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. A review on ensembles for the class imbalance problem: Bagging, boosting, and hybridbased approaches.

[70] Bartosz Krawczyk, Michał Wozniak, and Gerald Schaefer. Costsensitive decision tree ensembles for effective imbalanced classification.

[71] Octavio LoyolaGonzález, José Fco. MartínezTrinidad, Jesús Ariel CarrascoOchoa, and Milton GarcíaBorroto. Study of the impact of resampling methods for contrast pattern basedclassifiers in imbalanced databases. 175:935–947, 2016.

[72] Victoria López, Sara del Río, José Manuel Benítez, and Francisco Herrera. Costsensitive linguistic fuzzy rule based classification systemsunder the mapreduce framework for imbalanced big data.

[73] Guo Haixianga, Li Yijing, Jennifer Shang, Gu Mingyuna, Huang Yuanyuea, and Gong Binge. Learning from class-imbalanced data: Review of methods and applications.

[74] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique.

[75] Muhammad Atif Tahir, Josef Kittler, Krystian Mikolajczyk, and Fei Yan. A multiple expert approach to the class imbalance problem using inverse random under sampling.

[76] Jason Brownlee. What is the difference between a batch and an epoch in a neural network?, 2018. Deep Learning.

[77] Jason Brownlee. How to grid search hyperparameters for deep learning models in python with keras, August 2016. Deep Learning.

[78] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, 2018.

[79] SEBASTIAN RUDER. An overview of gradient descent optimization algorithms, January 2016. OPTIMIZATION.

[80] Sebastian Bock and Martin Weiß. A proof of local convergence for the adam optimizer. In *International Joint Conference on Neural Networks*, 2019.

[81] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *ICLR 2015*, 2015.

[82] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization.

[83] T. Tieleman and G. Hinton. Lecture 6.5 - rmsprop, coursera: Neural networks for machine learning.

[84] Abien Fred M. Agarap. Deep learning using rectified linear units (relu). In *arXiv preprint arXiv:1803.08375*, 2018.

[85] Mariana Belgiu and Lucian Dragut. Random forest in remote sensing: A review of applications and future directions.

[86] Vladimir Svetnik, Andy Liaw, Christopher Tong, J. Christopher Culberson, and Robert P. Sheridan and.

[87] Avinash Navlani. Understanding random forests classifiers in python, May 2018. datacamp.

[88] . sklearn.ensemble.randomforestclassifier.

[89] HARRY ZHANG. Exploring conditions for the optimality of naive bayes.

[90] . Naive bayes.

[91] SHICHAO ZHANG, XUELONG LI, MING ZONG, XIAOFENG ZHU, and DEBO CHENG. Learning k for knn classification.

[92] TAVISH SRIVASTAVA. Introduction to k-nearest neighbors: A powerful machine learning algorithm (with implementation in python r), 2018.

[93] Dhilip Subramanian. A simple introduction to k-nearest neighbors algorithm, 2019.

[94] . sklearn.neighbors.kneighborsclassifier.

[95] Jiashi Feng, Huan Xu, Shie Mannor, and Shuicheng Yan. Robust logistic regression and classification.

[96] Anuja Nagpal. L1 and l2 regularization methods, 2017.

[97] . sklearn.linear_model.logisticregression.

[98] Qiang Wu and Ding-Xuan Zhou. Analysis of support vector machine classification.

[99] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *the Fifth Annual Workshop of Computational Learning Theory 5*, 1992.

[100] Corinna Cortes and Vladimir Vapnik. Support-vector networks.

[101] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database.

[102] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[103] Rupesh K Srivastava, Klaus Greff, and Jurgen Schmidhuber. Training very deep networks.

[104] Y. Bengio, A. Courville, and P. Vincent. "representation learning: A review and new perspectives.

[105] Jeff Heaton. An empirical analysis of feature engineering for predictive modeling. In *SoutheastCon*, 2016.

[106] A. Coates, A. Y. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning.

[107] D. H. Wiesel and T. N. Hubel. Receptive fields of single neurones in the cat's striate cortex.

[108] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jurgen Schmidhuber. Deep big simple neural nets for handwritten digit recognition.

[109] Dominik Scherer, Andreas Muller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *20th International Conference on Artificial Neural Networks (ICANN)*, 2010.

[110] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks.

[111] . Conv1d.

[112] Jake Lever, Martin Krzywinski, and Naomi Altman. Classification evaluation.

[113] Koo Ping Shung. Accuracy, precision, recall or f1?, 2018. towards data science.

[114] Newcastle University. Coefficient of determination, r-squared, 2020.

[115] Coefficient of determination, 2021.

[116] sklearn.metrics.r2_score, 2021.

[117] Stephanie. What is relative absolute error, 2019. Statistics How To.

[118] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for image restoration with neural networks.

[119] Amit Shekhar. What are l1 and l2 loss functions?, 2019.

[120] Jim Frost. How to interpret r-squared in regression analysis, 2021.

[121] Stephanie. What is relative absolute error?, 2019.

APPENDICES

APPENDIX A

Coding Files Hierarchy Diagram

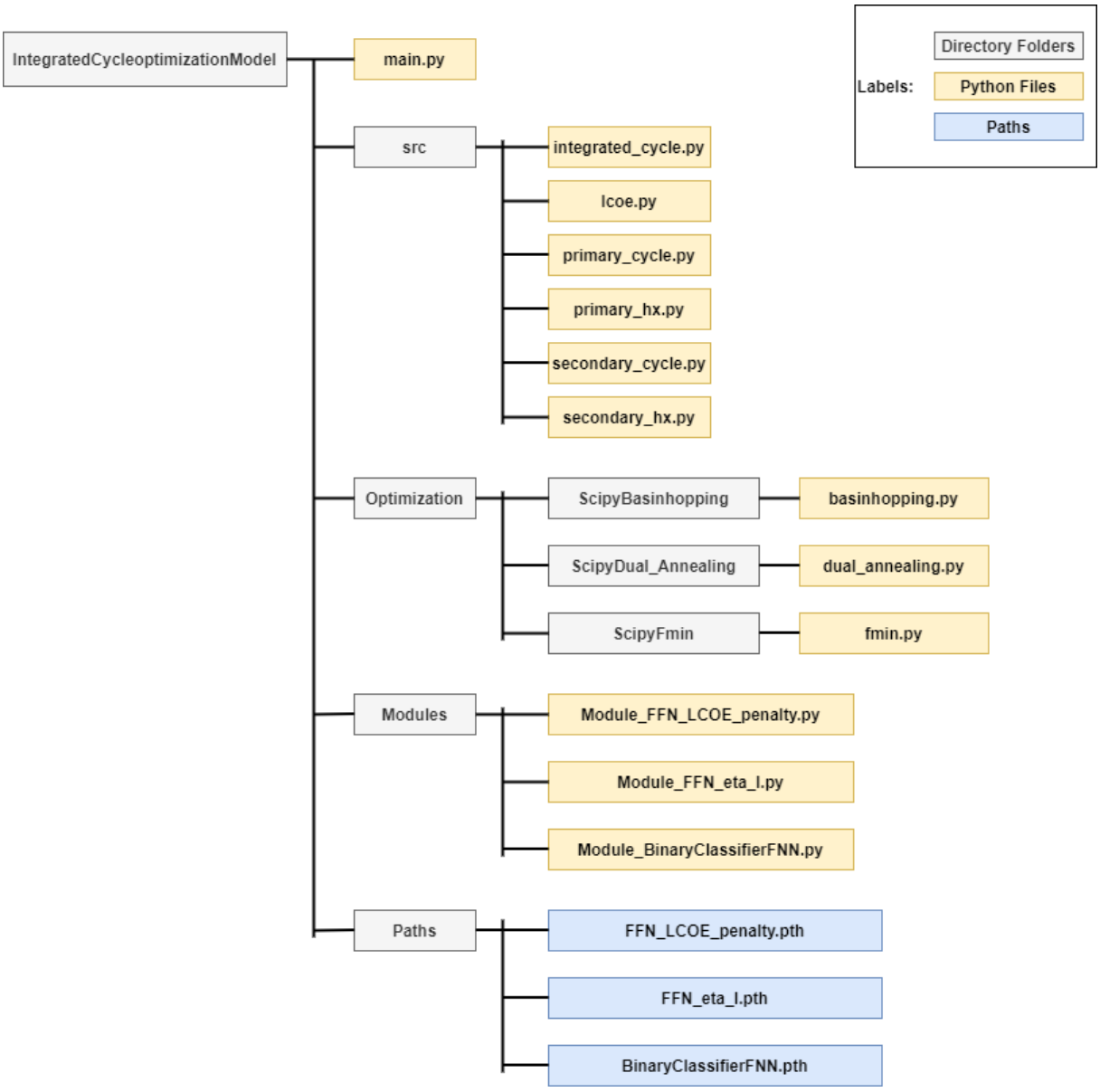


Fig. A.1: Coding Files Diagram.

APPENDIX B

main.py

In Appendix B, the **main.py** file unde the main directory in Fig. A.1 are presented. This is the only file needed to be run.

```
import numpy as np
import concurrent.futures
import time

from src.integrated_cycle import IntegratedCycle
from Optimization.ScipyBasinhopping.basinhopping import
    Basinhopping
from Optimization.ScipyDifferential_Evolution.
    differential_evolution import DifferentialEvolution
from Optimization.ScipyDual_Annealing.dual_annealing import
    DualAnnealing
from Optimization.ScipyFmin.fmin import Fmin


def optimizer_tool():
    """
    For user to choose the optimizer tool.
    """
    user_input = input("Which optimizer to apply?\n"
                       "Type 'B' for 'Basinhopping';\n"
```

```
                    "Type 'D1' for 'Differential Evolution';\n
                        "
                    "Type 'D2' for 'Dual Annealing';\n"
                    "Type 'F' for 'Fmin'.\n")
    if user_input == 'B':
        optimizer = Basinhopping()
    elif user_input == 'D1':
        optimizer = DifferentialEvolution()
    elif user_input == 'D2':
        optimizer = DualAnnealing()
    elif user_input == 'F':
        optimizer = Fmin()
    else:
        raise Exception(ValueError)

    return optimizer


def optimization(optimizer):
    """
    Do optimization with the chosen optimizer and then check the
        results with the physical-based model
    - f_rel_err: Relative errors between the optimized target (
        LCOE_surrogate + penalty_surrogate) from the surrogate
     model and the target (LCOE_true + penalty_true) from the
        physical-based model with the optimized design parameters
        .
```

- *P_max: Surrogate model optimized maximum pressure of the secondary cycle.*
- *Pr_1: Surrogate model optimized high pressure ratio of the secondary cycle.*
- *Pr_2: Surrogate model optimized middle pressure ratio of the secondary cycle.*
- *Pr_3: Surrogate model optimized low pressure ratio of the secondary cycle.*
- *f1: Surrogate model optimized high−pressure mass fraction of the secondary cycle.*
- *f2: Surrogate model optimized middle−pressure mass fraction of the secondary cycle.*
- *f3: Surrogate model optimized low−pressure mass fraction of the secondary cycle.*
- *f_surrogate: LCOE_surrogate + penalty_surrogate.*
- *f_true: LCOE_true + penalty_true.*
- *LCOE_true: The LCOE from the physical−based model with the surrogate model optimized design parameters.*
- *penalty_true: The penalty from the physical−based model with the surrogate model optimized design parameters.*
- *eta_FL_surrogate: The first law efficiency from the surrogate model with the surrogate model optimized design parameters.*
- *eta_FL_true: The first law efficiency from the physical−based model with the surrogate model optimized design parameters.*
- *eta_FL_rel_err: Relative error between the eta_FL_surrogate and the eta_FL_true.*

```
    """
    P_max_0, Pr_1, Pr_2, Pr_3, f1, f2, f3, f_surrogate,
        eta_FL_surrogate = optimizer.solve()
    P_max = (P_max_0 + 8.22) * 1e6

    try:
        model = IntegratedCycle(P_max, Pr_1, Pr_2, Pr_3, f1, f2,
            f3)
        model.solve()
        eta_FL_true = model.eta_FL
        LCOE_true, _ = model.LCOE()
        penalty_true = model.penalty()
        f_true = LCOE_true + penalty_true

        f_rel_err = (f_surrogate - f_true) / f_true
        eta_FL_rel_err = (eta_FL_surrogate - eta_FL_true) /
            eta_FL_true

        new_result = np.array([[f_surrogate, f_rel_err, P_max,
            Pr_1, Pr_2, Pr_3, f1, f2, f3, f_true, LCOE_true,
                                penalty_true, eta_FL_surrogate,
                                    eta_FL_true, eta_FL_rel_err]])
        convergence = True
    except:
        print('Surrogate Optimization Not Converged')
        new_result = [P_max_0, Pr_1, Pr_2, Pr_3, f1, f2, f3]
        convergence = False
```

```
    return convergence, new_result


def search_around(new_result, *args):
    """
    Search the viability of optimization around the non-converged
        outcome.
    - new_result: The not converged design parameters from the
       surrogate optimization output. Type: List
    - args: Tuple input of (replacement, delta)
    - replacement: The design parameter being replaced
    - delta: Value change on the replaced design parameter
    - result: The corresponding design parameters and outcomes of
        the newly searched design parameters
    """
    print(f'Start Optimization...')
    (replacement, delta) = args
    index = ['P_max', 'Pr_1', 'Pr_2', 'Pr_3', 'f1', 'f2', 'f3'].
       index(replacement)
    new_result[index] = new_result[index] + delta


    try:
        model = IntegratedCycle(new_result[0], new_result[1],
            new_result[2], new_result[3], new_result[4],
                                new_result[5], new_result[6])
        model.solve()
        eta_FL_true = model.eta_FL
        LCOE_true, _ = model.LCOE()
```

```
        penalty_true = model.penalty()
        f_true = LCOE_true + penalty_true

        result = np.array([[new_result[0], new_result[1],
            new_result[2], new_result[3], new_result[4],
            new_result[5],
                                new_result[6], f_true, LCOE_true,
                                    penalty_true, eta_FL_true]])
    except:
        result = 'Not Converged'
    return result


if __name__ == '__main__':
    """
    Run the program with multi-processors
    """
    optimizer = optimizer_tool()  # Get user-chosen optimization
        tool

    start = time.perf_counter()

    convergence, new_result = optimization(optimizer)
    if convergence:
        print('Surrogate Optimization Converged!')
        print(new_result)

        finish = time.perf_counter()
```

```
    print(f'Finish in {round(finish - start, 2)} second(s)')
else:
    delta = 0.5 # Try: 0.1, 0.01, 0.001
    check_points = [('P_max', delta*1e6), ('P_max', -delta*1
        e6), ('Pr_1', delta), ('Pr_1', -delta), ('Pr_2', delta
        ),
                    ('Pr_2', -delta), ('Pr_3', delta), ('Pr_3
                        ', -delta), ('f1', delta), ('f1', -
                        delta),
                    ('f2', delta), ('f2', -delta), ('f3',
                        delta), ('f3', -delta)] # No more than
                         20, since we only have 20 logical
                        processors
    with concurrent.futures.ProcessPoolExecutor() as executor
        :

        results = [executor.submit(search_around, new_result,
            *args) for args in check_points]

        for f in concurrent.futures.as_completed(results):
            print(f.result())

    finish = time.perf_counter()
    print(f'Finish in {round(finish-start, 2)} second(s)')
```

APPENDIX C

src

In Appendix C, all Python files under the **src** directory in Fig. A.1 are presented. The **src** directory includes all files of physical-based simulation of the Integrated Regenerative Methanol Transcritical Cycle.

C.1 integrated_cycle.py

```
import numpy as np
import scipy.optimize as opt
# import CoolProp.CoolProp as CoolProp
import CoolProp

from src.lcoe import hx_cost, pump_cost, condenser_cost
from src.secondary_hx import SecondaryHX
from src.primary_cycle import PrimaryCycle
from src.primary_hx import PrimaryHX
from src.secondary_cycle import SecondaryCycle


class IntegratedCycle:
    """
    IntegratedCycle models the NuScale SMR with a secondary
        transcritical Rankine cycle with Methanol as its working
    fluid. The primary and secondary cycles are solved
        simultaneously.
```

```
@author: Jacob Bryan
@date: 13 August 2020
"""
# ========================================
#   LCOE & Penalty Model Constants       =
# ========================================
CEPCI_2019 = 607.9248  # "Chemical Engineering Plant Cost
    Index in 2018"
CEPCI_2016 = 541.7  # "Chemical Engineering Plant Cost Index
    in 2016"
LCOE_2016 = 85  # [$/MWh] "Levelized cost of electricity of
    NuScale plant from: http://www.nuscalepower.com/smr-
    benefits/economical/operating-costs"
LCOE_2019 = LCOE_2016 * CEPCI_2019 / CEPCI_2016
OvernightEquipCost_frac = 0.08067  # "fraction of equipment
    costs of baseline cycle that contributes to the plant
    overnight costs (12 modules total): from baseline cycel
    model"
BaselineCycleCost = 20841690  # [$]
LCOECapital_frac = 0.5  # "fraction of LCOE that covers
    capital costs of the plant from: http://www.nuscalepower.
    com/smr-benefits/economical/construction-cost"
T_pinch = 5  # [K] pinch point delta temperature in feedwater
     heaters
T_normal = 10  # [K] value used to normalize hxer temperature
     constraints
T_out_normal = 1  # [K] value used to normalize hxer
    temperature constraints
```

```
x_normal = 0.1  # value used to normalize quality at mixing
    exit constraints
x_expansion_limit = 0.87
eta_Baseline = 0.3122

def __init__(self, P_max, Pr_1, Pr_2, Pr_3, f1, f2, f3, p_fld
    ='Water', s_fld='Methanol', p_backend='HEOS', s_backend='
    REFPROP'):
    """
    Parameters
    ----------
    P_max : float
        Maximum secondary cycle pressure [Pa]

    Pr_1 : float
        Pressure ratio of the first turbine. Ranges from 0 to
            1.

    Pr_2 : float
        Pressure ratio of the second turbine. Ranges from 0
            to 1.

    Pr_3 : float
        Pressure ratio of the third turbine. Ranges from 0 to
            1.

    f1 : float
```

Mass fraction of the first splitter. Ranges from 0 to 1.

f2 : float

Mass fraction of the second splitter. Ranges from 0 to 1.

f3 : float

Mass fraction of the third splitter. Ranges from 0 to 1.

p_fld : string (Optional)

Working fluid of the primary cycle. Default is 'Water'.

s_fld : string (Optional)

Working fluid of the secondary cycle. Default is 'Methanol'.

p_backend : string (Optional)

Specifies which backend CoolProp should use when calculating properties for the primary cycle fluid. Default

is 'HEOS'. NOTE: Using the REFPROP backend for both primary and secondary fluids will cause the model to run

significantly slower!

```
s_backend : string (Optional)
    Specifies which backend CoolProp should use when
        calculating properties for the secondary cycle
        fluid. Default
    is 'REFPROP'. NOTE: Using the REFPROP backend for
        both primary and secondary fluids will cause the
        model to run
    significantly slower!
"""
self.P = np.zeros(27)
self.T = np.zeros(27)
self.h = np.zeros(27)
self.m_dot = np.zeros(27)
self.w_t = np.zeros(4)
self.w_p = np.zeros(4)

self.P_max = P_max
self.Pr_1 = Pr_1
self.Pr_2 = Pr_2
self.Pr_3 = Pr_3
self.f1 = f1
self.f2 = f2
self.f3 = f3

self.p_fld = CoolProp.AbstractState(p_backend, p_fld)
self.s_fld = CoolProp.AbstractState(s_backend, s_fld)

self.eta_FL = np.nan
```

```
        self.DeltaT_SGmin = np.nan
        self.W_dot_t = np.nan
        self.W_dot_p = np.nan
        self.W_dot_net = np.nan
        self.Q_dot_in = np.nan

        self.hxers = None

        self.primary_cycle = PrimaryCycle(self.p_fld)
        self.m_dot_H = 0
        self.m_dot_C = 0

        self.use_surrogate = False
        self.tune = False
        self.rtol_surr = 1e-6

    def solve(self, rtol=1e-6, maxiter=100):
        """
        Solves the integrated primary and secondary cycles.
        """
        # Solve secondary cycle
        sec_cycle = SecondaryCycle(self.P_max, self.Pr_1, self.
            Pr_2, self.Pr_3, self.f1, self.f2, self.f3, self.s_fld
            , rtol, maxiter)
        sec_cycle_converged = sec_cycle.solve()

        if not sec_cycle_converged:
```

```
        raise ValueError('Secondary cycle solver did not
            converge!')

    # Copy over secondary cycle states to parent class object
        for ease of access
    self.P = sec_cycle.P
    self.T = sec_cycle.T
    self.h = sec_cycle.h
    self.m_dot = sec_cycle.m_dot
    self.w_t = sec_cycle.w_t
    self.w_p = sec_cycle.w_p

    # Specific turbine, pump, and net power; first law
        efficiency of secondary cycle calculation
    w_dot_t = np.sum(self.w_t * self.m_dot[[1, 3, 5, 7]])
    w_dot_p = np.sum(self.w_p * self.m_dot[[10, 13, 16, 19]])
    w_dot_net = w_dot_t - w_dot_p
    self.eta_FL = w_dot_net / (self.h[0] - self.h[20])

    # Seem to run into occasional non-physical solutions
        which give eta_FL outside of possible bounds. Still
        looking
    # into a way to keep the secondary solver bounded to
        avoid this problem or look for multiple roots.
    if self.eta_FL < 0 or self.eta_FL > 1:
        raise ValueError('eta was negative!')

    # Main loop initial guesses (default secant method)
```

```
        x0 = 160e6  # baseline
        x1 = 400 * (self.h[0] - self.h[20])  # physics-informed
            guess based on secondary cycle solution and '
            reasonable' m_dot_C value

        # Main loop (primary cycle and primary heat exchanger)
            solution
        res = opt.root_scalar(f=self._main_loop_objective, x0=x0,
             x1=x1, method='secant', rtol=rtol, maxiter=maxiter)

        if not res.converged:
            raise ValueError('Integrated cycle solve has not
                converged!')

        # Final calculation of total power terms
        self.Q_dot_in = res.root
        self.W_dot_t = np.sum(self.w_t * self.m_dot_C * self.
            m_dot[[1, 3, 5, 7]])
        self.W_dot_p = np.sum(self.w_p * self.m_dot_C * self.
            m_dot[[10, 13, 16, 19]])
        self.W_dot_net = self.W_dot_t - self.W_dot_p

    def _main_loop_objective(self, Q_dot_in):
        """
        Objective function for main loop solver. Contains calls
            for the primary cycle and primary heat exchanger
            solvers.
        """
```

```
        # Solve primary and secondary cycle mass flow rates as
            function of Q_dot_in estimate
        self.m_dot_C = Q_dot_in / (self.h[0] - self.h[20])
        self.m_dot_H = self.primary_cycle.solve(Q_dot_in)

        # Primary HX Solver
        primary_hx = PrimaryHX(self.P_max, self.m_dot_H, self.
            m_dot_C, self.p_fld, self.s_fld)
        primary_hx.solve()
        Q_dot_update = primary_hx.Q_dot_in
        self.DeltaT_SGmin = primary_hx.DeltaT_SGmin

        return Q_dot_update - Q_dot_in

    def LCOE(self):
        """
        Calculates levelized cost of energy. Returns LCOE and
            percent change in LCOE.
        """
        # Scale m_dot array (stored as mass fractions) to actual
            mass flow rates
        self.m_dot *= self.m_dot_C

        # Required heat transfer rates for secondary cycle heat
            exchangers
        Q_dot_hx1 = self.m_dot[7] * (self.h[7] - self.h[8])
        Q_dot_hx2 = self.m_dot[25] * (self.h[25] - self.h[26])
        Q_dot_hx3 = self.m_dot[23] * (self.h[23] - self.h[24])
```

```
Q_dot_hx4 = self.m_dot[21] * (self.h[21] - self.h[22])

# Solve secondary HX geometries & profiles
self.hxers = [SecondaryHX(self.s_fld, self.s_fld, self.h
    [7], self.h[10], self.m_dot[7], self.m_dot[10],
    Q_dot_hx1, self.P[7], self.P[10]),
                SecondaryHX(self.s_fld, self.s_fld, self.h
                    [25], self.h[13], self.m_dot[25], self.
                    m_dot[13], Q_dot_hx2, self.P[25], self.P
                    [13]),
                SecondaryHX(self.s_fld, self.s_fld, self.h
                    [23], self.h[16], self.m_dot[23], self.
                    m_dot[16], Q_dot_hx3, self.P[23], self.P
                    [16]),
                SecondaryHX(self.s_fld, self.s_fld, self.h
                    [21], self.h[19], self.m_dot[21], self.
                    m_dot[19], Q_dot_hx4, self.P[21], self.P
                    [19])]

# Calculate cost terms
C = np.zeros(9)
C[0] = hx_cost(self.hxers[0].area, self.P[10] * 1e-6)
C[1] = hx_cost(self.hxers[1].area, self.P[13] * 1e-6)
C[2] = hx_cost(self.hxers[2].area, self.P[16] * 1e-6)
C[3] = hx_cost(self.hxers[3].area, self.P[19] * 1e-6)
C[4] = pump_cost(self.w_p[0] * self.m_dot[10] * 1e-3,
    self.P[10] * 1e-6)
```

```
        C[5] = pump_cost(self.w_p[1] * self.m_dot[13] * 1e-3,
            self.P[13] * 1e-6)
        C[6] = pump_cost(self.w_p[2] * self.m_dot[16] * 1e-3,
            self.P[16] * 1e-6)
        C[7] = pump_cost(self.w_p[3] * self.m_dot[19] * 1e-3,
            self.P[19] * 1e-6)
        C[8] = condenser_cost(self.W_dot_t)

        C_total = np.sum(C)

        # Calculate LCOE of the cycle
        eff_frac = self.Q_dot_in * self.eta_FL / (160e6 * self.
            eta_Baseline)
        cost_percent_change = (C_total - self.BaselineCycleCost)
            / self.BaselineCycleCost
        LCOE_new = (self.LCOE_2019 + self.LCOE_2019 * self.
            LCOECapital_frac * self.OvernightEquipCost_frac *
            cost_percent_change) / eff_frac
        LCOE_percent_change = (LCOE_new - self.LCOE_2019) / self.
            LCOE_2019 * 100

        return LCOE_new, LCOE_percent_change

    def penalty(self):
        """
        Calculates and returns penalty term.
        """
        P_crit = self.s_fld.p_critical()
```

```
g = np.zeros(16)

if self.hxers is None:
    self.LCOE()  # HX objects are created in LCOE(); call
        this function if they haven't been made yet to
        avoid throwing an error. TODO: This isn't pretty
        ...

for i in range(4):  # g1 to g4 -- condenser and heat
    exchanger regen
    g[i] = self.hxers[i].penalty / self.T_normal

for i, j in zip([4, 5, 6, 7], [10, 13, 16, 19]):  # g6 to
    g9 -- temperature change in cold side of heat
    exchangers
    g[i] = max(0, (self.T[j] - self.T[j + 1]) / self.
        T_out_normal)

for i, j in zip([8, 9, 10], [12, 15, 18]):  # g10 to g12
    -- pump quality constraints
    g[i] = self.quality_penalty(self.h[j], self.P[j],
        P_crit) / self.x_normal

for i, j in zip([11, 12, 13, 14], [1, 3, 5, 7]):  # g13
    to g16 -- turbine expansion limits
    self.s_fld.update(CoolProp.HmassP_INPUTS, self.h[j],
        self.P[j])
```

```
            g[i] = max(0, (self.x_expansion_limit - self.
                _adjust_quality(self.s_fld.Q(), self.s_fld.phase()
                ))) / self.x_normal

        g[15] = max(0, self.T_pinch - self.DeltaT_SGmin) / self.
            T_normal  # g17 -- primary heat exchanger pinch temp

        penalty = np.sum(g ** 2)  # Using element-wise square of
            individual terms to discourage higher individual
            penalty values

        return penalty

    def quality_penalty(self, h, P, P_crit):
        if P > P_crit:
            return 0
        else:
            self.s_fld.update(CoolProp.HmassP_INPUTS, h, P)
            Q = self.s_fld.Q()
            phase = self.s_fld.phase()
            return self._adjust_quality(Q, phase)

    @staticmethod
    def _adjust_quality(x, phase):
        """
        Returns an appropriate quality value given the input
            phase. The quality value returned by CoolProp for
            phases
```

```
        other than two-phase mixtures don't have an interpretable
            quality value. This returns a value appropriate for
        the phase.
        """
        if phase == 6:  # two-phase
            x_ret = x
        elif phase == 0:  # liquid
            x_ret = 0
        elif phase in [1, 2, 3, 5]:  # supercritical,
           supercritical gas, supercritical liquid, gas
            x_ret = 1
        else:
            raise ValueError('Unsupported phase: {}'.format(phase
                ))
        return x_ret
```

C.2 lcoe.py

```
from math import floor, log10


CEPCI_2019 = 607.9248  # "Chemical Engineering Plant Cost Index
    in 2018"


def hx_cost(A, P):
    K1 = 4.3247
    K2 = -0.3030
    K3 = 0.1634
```

```
B1 = 1.74
B2 = 1.55

P_g = P - 0.101325
if P_g < 0.5:
    C1 = 0
    C2 = 0
    C3 = 0
else:
    C1 = 0.03881
    C2 = -0.11272
    C3 = 0.08183

F_p = _pressure_correction(P_g * 0.1, C1, C2, C3)
F_m = 2.73  # "For Stainless Steel"

# "if area > 1000 need to compute multiple heat exchangers"
max_area = 1000  # [m^2]
if A > max_area:
    n = floor(A / max_area)
    rem = A % max_area
    unit_cost = _bare_module_cost(max_area, K1, K2, K3, B1,
        B2, F_p, F_m)
    rem_cost = _bare_module_cost(rem, K1, K2, K3, B1, B2, F_p
        , F_m)
    cost = n * unit_cost + rem_cost
else:
```

```
        cost = _bare_module_cost(A, K1, K2, K3, B1, B2, F_p, F_m)

    return cost


def _pressure_correction(P, C1, C2, C3):
    if C1 == C2 == C3 == 0:
        return 1
    else:
        F_p = 10 ** (C1 + -C2 * log10(P) + C3 * (log10(P)) ** 2)
    return F_p


def _bare_module_cost(A, K1, K2, K3, B1, B2, F_p, F_m):
    CEPCI_1 = 397  # "Chemical Engineering Plant Cost Index in
        2001"
    if A <= 1:
        C_p_o = 0  # [$]
    else:
        C_p_o = 10 ** (K1 + K2 * (log10(A)) + K3 * (log10(A)) **
            2) * CEPCI_2019 / CEPCI_1  # "Non-corrected equipment
            cost 2019 dollars"
    cost = C_p_o * (B1 + B2 * F_p * F_m)  # "Bare module cost"
    return cost


def pump_cost(W, P):
    K1 = 3.3892
```

```
    K2 = 0.0536
    K3 = 0.1538

    B1 = 1.89
    B2 = 1.35

    P_g = P - 0.101325
    if P_g < 10:
        C1 = 0
        C2 = 0
        C3 = 0
    else:
        C1 = -0.3935
        C2 = 0.3957
        C3 = -0.00226

    F_P = _pressure_correction(P * 0.1, C1, C2, C3)
    F_m = 2.3

    cost = _bare_module_cost(W, K1, K2, K3, B1, B2, F_P, F_m)

    return cost


def condenser_cost(W_dot_t):
    a = 2913.64828
    b = 0.70920
    F_m = 3
```

```
    cost = a * (1e-3 * W_dot_t * 1) ** b * F_m
    return cost
```

C.3 primary_cycle.py

```
import numpy as np
import CoolProp.CoolProp as CP
import scipy.optimize as opt


class PrimaryCycle:
    T_2 = 583.15  # [K]
    P_primary = 12.8e6  # [Pa]
    # Primary Heights in Both Steam Generator and Reactor Core [m
        ]
    z_1 = 0.6622034
    z_2 = 6.0026042
    z_3 = 8.0518
    z_4 = 14.404975
    # Height differences in each section [m]
    dh = np.array([z_2 - z_1,
                   z_4 - z_2,
                   z_4 - z_3,
                   z_3 - z_1])
    # Flow areas [m^2]
    A_1 = 0.9606432
    A_3 = 2.660188228
    k_loss = 27.6  # loss constant
```

```
def __init__(self, fld):
    self.fld = fld

def solve(self, Q_dot_in, rtol=1e-6, maxiter=100):
    primary_cycle_soln = opt.root_scalar(f=self.
        _objective_primary, bracket=[273, 583.149], method='
        brentq',
                                         args=(Q_dot_in),
                                            rtol=rtol,
                                            maxiter=maxiter)

    if not primary_cycle_soln.converged:
        raise ValueError(primary_cycle_soln.flag)

    T_4 = primary_cycle_soln.root
    T_1 = 0.5 * (self.T_2 + T_4)

    self.fld.update(CP.PT_INPUTS, self.P_primary, T_1)
    cp = self.fld.cpmass()
    m_dot_H = Q_dot_in / (cp * (self.T_2 - T_4))
    if m_dot_H < 0:
        raise ValueError('m_dot_H is negative! T_4 = {}'.
            format(T_4))

    return m_dot_H

def _objective_primary(self, T_4, Q_dot_core):
    T_1 = (self.T_2 + T_4) / 2  # also is T_3
```

```
        T_p = np.array([T_1, self.T_2, T_1, T_4])
        rho_p = np.zeros(4)

        self.fld.update(CP.PT_INPUTS, self.P_primary, T_1)
        cp = self.fld.cpmass()
        rho_p[[0, 2]] = self.fld.rhomass()

        self.fld.update(CP.PT_INPUTS, self.P_primary, T_p[1])
        rho_p[1] = self.fld.rhomass()

        self.fld.update(CP.PT_INPUTS, self.P_primary, T_p[3])
        rho_p[3] = self.fld.rhomass()

        m_dot = Q_dot_core / (cp * (self.T_2 - T_4))
        P_p = rho_p * 9.80665 * self.dh  # hydrostatic pressure [
            Pa]

        u_1 = m_dot / self.A_1 / rho_p[0]
        u_3 = m_dot / self.A_3 / rho_p[2]

        P_drive_P = P_p[2] + P_p[3] - P_p[0] - P_p[1]
        P_loss_P = 0.5 * self.k_loss * rho_p[0] * (u_1 ** 2 + u_3
            ** 2)
        Delta_P = P_drive_P - P_loss_P

        return Delta_P
```

C.4 primary_hx.py

```
import numpy as np
import CoolProp.CoolProp as CP


def euler(f, y0, z0, t_span, N):
    t = np.linspace(t_span[0], t_span[1], N + 1)
    y_dims = (N + 1,) if (isinstance(y0, float) or len(y0) == 1)
        else (N + 1, len(y0))
    y = np.zeros(y_dims)
    z = np.zeros(y_dims)
    y[0] = y0
    z[0] = z0
    for i in range(N):
        h = t[i + 1] - t[i]
        fn = f(t[i], y[i])
        y[i + 1] = y[i] + fn[:2] * h
        z[i + 1] = fn[2:]
    return y.T, z.T


class PrimaryHX:
    T_p = 310 + 273.15  # hot side inlet
    T_s = 301 + 273.15  # cold side outlet
    tube_material = 'Inconel_718'  # tube material
    RelRough = 0.035  # tube roughness
    d_o = 0.015875  # [m] "outside diameter of steam generator
        tube. Data from Kevin Drost email"
```

```
d_i = 0.013335  # [m] "inside diameter of steam generator
    tube.  Data from Kevin Drost email"
A_primary_flow_min = 1.597932  # [m^2] "minimum flow area
    between tubes for primary side.  Data from Kevin Drost
    email"
N_t = 1380  # Total number of tubes. Data from drawing #:
    NP12-01-A011-M-SA-2689-S02
r_c = 0.95885254  # [m] "Averge tube column radius. Data from
     drawing #: NP12-01-A011-M-SA-2689-S02"
d_c = 2 * r_c  # average tube column diameter
k_t = 15.095  # [W/m-K] Thermal Resistance Analogy
L = 24.4812312  # [m] "Average total tube length. Data from
    drawing #: NP12-01-A011-M-SA-2689-S02"
P_primary = 12.8e6  # [Pa]

# Some terms that remain constant; precalculated to save time
     in the long run
R_foul_dprime = 5e-6  # [m^2-K/W] "Fouling factor for tube
    walls. Value selected to match temp profile to NuScale
    Analysis"
R_prime_wall = np.log(d_o / d_i) / (2 * np.pi * k_t)
R_prime_H_foul = R_foul_dprime / (np.pi * d_o)
R_prime_C_foul = R_foul_dprime / (np.pi * d_i)
R_prime_wall_tot = R_prime_wall + R_prime_H_foul +
    R_prime_C_foul

C_z = 0.27
m_z = 0.63
```

```
n_z = 0.36
C_nuscale = 1
h_H_const1 = C_nuscale * C_z / d_o * (d_o /
    A_primary_flow_min) ** m_z

def __init__(self, P_max, m_dot_H, m_dot_C, h_fld, c_fld,
    rtol=1e-6, maxiter=100):
     self.m_dot_H = m_dot_H
     self.m_dot_C = m_dot_C
     self.m_dot_H_tube = self.m_dot_H / self.N_t
     self.m_dot_C_tube = self.m_dot_C / self.N_t
     self.P_max = P_max

     self.i_H = None
     self.i_C = None
     self.T_H = None
     self.T_C = None
     self.x = None
     self.Q_dot_in = None
     self.DeltaT_SGmin = None

     self.p_fld = h_fld
     self.s_fld = c_fld

     self.h_H = 9850
     self.h_H_old = 9850
     self.h_C = 7750
     self.h_C_old = 7750
```

```
        self.rtol = rtol
        self.maxiter = int(maxiter)

        self.h_H_const2 = m_dot_H ** self.m_z

    def solve(self, N=50):
        self.p_fld.update(CP.PT_INPUTS, self.P_primary, self.T_p)
        i_h_in = self.p_fld.hmass()
        self.s_fld.update(CP.PT_INPUTS, self.P_max, self.T_s)
        i_C_out = self.s_fld.hmass()

        (self.i_H, self.i_C), (self.T_H, self.T_C) = euler(self.
            enthalpy_diff, t_span=(0, self.L), y0=[i_h_in, i_C_out
            ], z0=[self.T_p, self.T_s], N=N)
        self.Q_dot_in = self.m_dot_H * (i_h_in - self.i_H[-1])
        self.DeltaT_SGmin = np.min(self.T_H - self.T_C)

    def enthalpy_diff(self, t, x):
        i_H, i_C = x

        # Use linear approximation to precondition loop. Usually
            keeps the solver limited to just 2-3 iterations!
        h_H, self.h_C, T_H, T_C, dqdx_tube, converged = self.
            _convection_coeffs(2 * self.h_H - self.h_H_old, i_H,
            i_C)
        self.h_H_old = self.h_H
        self.h_H = h_H
```

```
        if not converged:
            raise ValueError('Convection coefficient solution has
                not converged')

        di_H = -dqdx_tube / self.m_dot_H_tube
        di_C = -dqdx_tube / self.m_dot_C_tube

        return np.array([di_H, di_C, T_H, T_C])

    def _convection_coeffs(self, h_H, i_H, i_C):
        """
        Iteratively solves for the convection coefficients of a
            heat exchanger.
        """
        # Get hot fluid properties
        self.p_fld.update(CP.HmassP_INPUTS, i_H, self.P_primary)
            # 0.03 sec
        Pr_H_avg = self.p_fld.Prandtl()
        k_H = self.p_fld.conductivity()
        T_H = self.p_fld.T()
        h_H_const3 = Pr_H_avg ** (self.n_z + 0.25) * k_H

        # Get cold fluid properties
        self.s_fld.update(CP.HmassP_INPUTS, i_C, self.P_max)  #
            0.4 sec
        T_C = self.s_fld.T()
        Pr_C = self.s_fld.Prandtl()
```

```
k_C = self.s_fld.conductivity()
mu_C = self.s_fld.viscosity()
Re_C = 4 * self.m_dot_C_tube / (np.pi * self.d_i * mu_C)
Nu_C = 0.023 * abs(Re_C) ** 0.8 * abs(Pr_C) ** 0.4
h_C = Nu_C * k_C / self.d_i
R_prime_C = 1 / (h_C * np.pi * self.d_i)
R_prime_other = R_prime_C + self.R_prime_wall_tot

dqdx_tube = 0  # default value that shouldn't ever
    actually be used, but it keeps the linter from yelling
     at me...

converged = False

for i in range(self.maxiter):
    R_prime_H = 1 / (h_H * np.pi * self.d_o)
    R_prime_T = R_prime_H + R_prime_other

    dqdx_tube = (T_H - T_C) / R_prime_T
    T_w_o = T_H - dqdx_tube * R_prime_H

    self.p_fld.update(CP.PT_INPUTS, self.P_primary, T_w_o
        )  # 0.3 sec
    Pr_H_s = self.p_fld.Prandtl()
    mu_H_local = self.p_fld.viscosity()
```

```
            h_H_update = self.h_H_const1 * self.h_H_const2 *
                h_H_const3 * mu_H_local ** (-self.m_z) * Pr_H_s **
                (-0.25)

            diff_h = abs(h_H_update - h_H) / h_H_update

            h_H = h_H_update

            if diff_h < self.rtol:
                converged = True
                break

        return h_H, h_C, T_H, T_C, dqdx_tube, converged
```

C.5 secondary_cycle.py

```
import numpy as np
import CoolProp.CoolProp as CP


class SecondaryCycle:
    T_max = 301 + 273.15  # maximum src temperature
    T_pinch = 5  # pinch temperature for heat exchangers
    T_min = 35 + 273.15  # minimum src temperature
    eta_p = 0.75  # pump efficiency
    eta_t = 0.85  # turbine efficiency

    def __init__(self, P_max, Pr_1, Pr_2, Pr_3, f1, f2, f3, fld,
        rtol=1e-6, maxiter=100,):
```

```
self.fld = fld
self.rtol = rtol
self.maxiter = maxiter

self.P_max = P_max
self.Pr_1 = Pr_1
self.Pr_2 = Pr_2
self.Pr_3 = Pr_3
self.f1 = f1
self.f2 = f2
self.f3 = f3

self.P = self.init_P()
self.m_dot = self.init_m_dot()
self.T = np.zeros(27)
self.T[0] = self.T_max
self.h = np.zeros(27)
self.w_t = np.zeros(4)
self.w_p = np.zeros(4)

self.m_dot_tot = 0

#
    ==========================================================

# ==== These states are defined by the design parameters
    ====
```

```
#
    ================================================================


# State 1 properties are given by the max temperature and
    pressure of the src
self.fld.update(CP.PT_INPUTS, self.P_max, self.T_max)
self.h[0] = self.fld.hmass()

# Solve state 2 properties
self.w_t[0], self.h[1], self.T[1] = self._turbine(self.h
    [0], self.P[0], self.P[1])

# Solve state 3 properties
self.T[[2, 21]] = self.T[1]
self.h[[2, 21]] = self.h[1]

# Solve state 4 properties
self.w_t[1], self.h[3], self.T[3] = self._turbine(self.h
    [2], self.P[2], self.P[3])

# Solve state 5 properties
self.T[[4, 23]] = self.T[3]
self.h[[4, 23]] = self.h[3]

# Solve state 6 properties
self.w_t[2], self.h[5], self.T[5] = self._turbine(self.h
    [4], self.P[4], self.P[5])
```

```
        # Solve state 7 properties
        self.T[[6, 25]] = self.T[5]
        self.h[[6, 25]] = self.h[5]

        # Solve state 8 properties
        self.fld.update(CP.QT_INPUTS, 0, self.T_min)
        P_min = self.fld.p()
        self.P[[7, 8, 9]] = P_min
        self.fld.update(CP.PT_INPUTS, self.P[6], self.T[6])
        self.w_t[3], self.h[7], self.T[7] = self._turbine(self.h
            [6], self.P[6], self.P[7])

        # Solve state 10 properties
        self.fld.update(CP.PQ_INPUTS, self.P[9], 0)
        self.T[9] = self.fld.T()
        self.h[9] = self.fld.hmass()
        s10 = self.fld.smass()

        # Solve state 11 properties
        self.w_p[0], self.h[10], self.T[10] = self._pump(self.h
            [9], s10, self.P[10])

        # Solve state 12 enthalpy
        self.h[11] = self._hot_pinch_temp(i_hot_in=7, i_cold_in
            =10)

    def solve(self, h13o=64e3, h16o=293e3, h19o=630e3):
        converged = False
```

```
for j in range(self.maxiter):
    for i, (ind, ho) in enumerate(zip([13, 16, 19], [h13o
        , h16o, h19o])):
        self.fld.update(CP.HmassP_INPUTS, ho, self.P[ind
            - 1])
        self.w_p[i + 1], self.h[ind], self.T[ind] = self.
            _pump(ho, self.fld.smass(), self.P[ind])

    # Solves output of cold side of each heat exchanger
    for a, b, c in [[14, 25, 13], [17, 23, 16], [20, 21,
        19]]:
        self.h[a] = self._hot_pinch_temp(b, c)

    # Update enthalpies based on this mass flow rate
    self.h[22] = self.h[21] - (self.h[20] - self.h[19]) /
        self.f1
    self.h[24] = self.h[23] - (self.h[17] - self.h[16]) /
        self.f2
    self.h[26] = self.h[25] - (self.h[14] - self.h[13]) /
        self.f3
    self.h[12] = (self.m_dot[11] * self.h[11] + self.
        m_dot[26] * self.h[26]) / self.m_dot[12]
    self.h[15] = (self.m_dot[14] * self.h[14] + self.
        m_dot[24] * self.h[24]) / self.m_dot[15]
    self.h[18] = (self.m_dot[17] * self.h[17] + self.
        m_dot[22] * self.h[22]) / self.m_dot[18]

    # Errors in our root estimates
```

```
            errs = np.array([(self.h[12] - h13o) / self.h[12],
                             (self.h[15] - h16o) / self.h[15],
                             (self.h[18] - h19o) / self.h[18]])

            # Next estimates are the newly calculated values (
                fixed point iteration)
            h13o = self.h[12]
            h16o = self.h[15]
            h19o = self.h[18]

            # Check rtol exit criteria
            if np.all(abs(errs) < self.rtol):
                converged = True
                break

        # Solve state 9 enthalpy
        self.h[8] = self.h[7] - self.h[11] + self.h[10]

        # Other temperatures
        for ind in [11, 13, 14, 16, 17, 19, 20]:
            self.fld.update(CP.HmassP_INPUTS, self.h[ind], self.P
                [ind])
            self.T[ind] = self.fld.T()

        return converged

    def init_P(self):
        P = np.zeros(27)
```

```
        P[[0, 19, 20]] = 1
        P[[1, 2, 16, 17, 18, 21, 22]] = self.Pr_1
        P[[3, 4, 13, 14, 15, 23, 24]] = self.Pr_1 * self.Pr_2
        P[[5, 6, 10, 11, 12, 25, 26]] = self.Pr_1 * self.Pr_2 *
            self.Pr_3
        P *= self.P_max
        return P

    def init_m_dot(self):
        m_dot = np.ones(27)
        m_dot[[2, 3, 15, 16, 17]] = 1 - self.f1
        m_dot[[4, 5, 12, 13, 14]] = (1 - self.f1) * (1 - self.f2)
        m_dot[[6, 7, 8, 9, 10, 11]] = (1 - self.f1) * (1 - self.
            f2) * (1 - self.f3)
        m_dot[[21, 22]] = self.f1
        m_dot[[23, 24]] = (1 - self.f1) * self.f2
        m_dot[[25, 26]] = (1 - self.f1) * (1 - self.f2) * self.f3
        return m_dot

    def _turbine(self, h_in, P_in, P_out):
        self.fld.update(CP.HmassP_INPUTS, h_in, P_in)
        x1 = self.fld.Q()
        phase1 = self.fld.phase()
        x_in = self._adjust_quality(x1, phase1)
        s_in = self.fld.smass()
        s_s_out = s_in
        self.fld.update(CP.PSmass_INPUTS, P_out, s_s_out)
        h_s_out = self.fld.hmass()
```

```
        x2 = self.fld.Q()
        phase2 = self.fld.phase()
        x_out_s = self._adjust_quality(x2, phase2)
        x_a = 0.5 * (x_in + x_out_s)
        eta_a = self.eta_t * (1 - 0.72 * (1 - x_a))
        W_dot_t_s_m_dot = h_in - h_s_out
        W_dot_t_m_dot = W_dot_t_s_m_dot * eta_a
        h_out = h_in - W_dot_t_m_dot
        self.fld.update(CP.HmassP_INPUTS, h_out, P_out)
        T_out = self.fld.T()
        return W_dot_t_m_dot, h_out, T_out

    def _pump(self, h_in, s_in, P_out):
        self.fld.update(CP.PSmass_INPUTS, P_out, s_in)  # output
            with no entropy change
        h_s_out = self.fld.hmass()
        W_dot_s_p_m_dot = h_s_out - h_in  # pump work needed w/
            no entropy change
        W_dot_p_m_dot = W_dot_s_p_m_dot / self.eta_p  # actual
            pump work needed d/t efficiency losses
        h_out = h_in + W_dot_p_m_dot  # output enthalpy
        self.fld.update(CP.HmassP_INPUTS, h_out, P_out)  # actual
             output state
        T_out = self.fld.T()
        return W_dot_p_m_dot, h_out, T_out

    def _adjust_quality(self, x, phase):
        """
```

```
        Returns an appropriate quality value given the input
            phase. The quality value returned by CoolProp for
            phases
        other than two-phase mixtures don't have an interpretable
             quality value. This returns a value appropriate for
        the phase.
        """
        if phase == 6:  # two-phase
            x_ret = x
        elif phase == 0:  # liquid
            x_ret = 0
        elif phase in [1, 2, 3, 5]:  # supercritical,
            supercritical gas, supercritical liquid, gas
            x_ret = 1
        else:
            raise ValueError('Unsupported phase: {}'.format(phase
                ))
        return x_ret


    def _hot_pinch_temp(self, i_hot_in, i_cold_in):
        """
        This is a bunch of conditions for what's essentially just

        h_c_out = h_c_pinch + delta_h * m_dot_H / m_dot_C

        This is a function only of the input hot and cold states.
            Note that we know the ratio of mass flow rates from
```

```
    the design parameters; we don't need to know the actual
        mass flow rates!
    """
    P_crit = self.fld.p_critical()
    self.fld.update(CP.HmassP_INPUTS, self.h[i_hot_in], self.
        P[i_hot_in])
    T_h_in = self.fld.T()
    self.fld.update(CP.HmassP_INPUTS, self.h[i_cold_in], self
        .P[i_cold_in])
    T_c_in = self.fld.T()

    if T_h_in - T_c_in > self.T_pinch:  # temp difference is
        greater than pinch temp
        if self.P[i_cold_in] < P_crit and self.P[i_hot_in] <
            P_crit:  # below supercritical
            self.fld.update(CP.PQ_INPUTS, self.P[i_hot_in],
                1)
            T_h_sat, h_h_sat = self.fld.T(), self.fld.hmass()
            self.fld.update(CP.PQ_INPUTS, self.P[i_cold_in],
                1)
            T_c_sat = self.fld.T()
            if T_h_sat - self.T_pinch == T_c_sat:
                self.fld.update(CP.PQ_INPUTS, self.P[
                    i_cold_in], 0)
                h_c_pinch = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] - h_h_sat)
            elif T_h_sat - self.T_pinch > T_c_in:
```

```
                self.fld.update(CP.PT_INPUTS, self.P[
                    i_cold_in], T_h_sat - self.T_pinch)
                h_c_pinch = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] - h_h_sat)
            else:
                h_c_pinch = self.h[i_cold_in]
                self.fld.update(CP.PT_INPUTS, self.P[i_hot_in
                    ], T_c_in + self.T_pinch)
                h_superheat = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] -
                    h_superheat)
        elif self.P[i_hot_in] < P_crit:  # hot side not
            supercritical
            self.fld.update(CP.PQ_INPUTS, self.P[i_hot_in],
                1)
            T_h_sat, h_h_sat = self.fld.T(), self.fld.hmass()
            if T_h_sat - T_c_in > self.T_pinch:  # temp
                difference greater than pinch temp
                self.fld.update(CP.PT_INPUTS, self.P[
                    i_cold_in], T_h_sat - self.T_pinch)
                h_c_pinch = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] - h_h_sat)
            else:
                h_c_pinch = self.h[i_cold_in]
                self.fld.update(CP.PT_INPUTS, self.P[i_hot_in
                    ], T_c_in + self.T_pinch)
                h_superheat = self.fld.hmass()
```

```
                delta_h = max(0, self.h[i_hot_in] -
                    h_superheat)
        else:  # both are supercritical
            P_f1 = P_crit - 1
            self.fld.update(CP.PQ_INPUTS, P_f1, 1)
            h_h_pinch = self.fld.hmass()
            self.fld.update(CP.HmassP_INPUTS, h_h_pinch, self
                .P[i_cold_in])
            T_h_pinch = self.fld.T()
            if T_h_pinch - T_c_in > self.T_pinch:
                self.fld.update(CP.PT_INPUTS, self.P[
                    i_cold_in], T_h_pinch - self.T_pinch)
                h_c_pinch = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] - h_h_pinch
                    )
            else:
                h_c_pinch = self.h[i_cold_in]
                self.fld.update(CP.PT_INPUTS, self.P[i_hot_in
                    ], T_c_in + self.T_pinch)
                h_superheat = self.fld.hmass()
                delta_h = max(0, self.h[i_hot_in] -
                    h_superheat)
        Q_dot = self.m_dot[i_hot_in] * delta_h
    else:
        h_c_pinch = self.h[i_cold_in]
        Q_dot = 0

    return h_c_pinch + Q_dot / self.m_dot[i_cold_in]
```

C.6 secondary_hx.py

```
import numpy as np
import CoolProp.CoolProp as CP


class SecondaryHX:

    N_area = 30
    T_pinch = 5

    def __init__(self, h_fld, c_fld, h_hot_in, h_cold_in,
        m_dot_hot, m_dot_cold, Q_dot, P_hot, P_cold):
        self.h_fld = h_fld
        self.c_fld = c_fld
        self.P_hot = P_hot
        self.P_cold = P_cold

        self.h_h = np.zeros(self.N_area + 1)
        self.T_h = np.zeros(self.N_area + 1)
        self.phase_h = np.zeros(self.N_area + 1)

        self.h_c = np.zeros(self.N_area + 1)
        self.T_c = np.zeros(self.N_area + 1)
        self.phase_c = np.zeros(self.N_area + 1)

        self.area = self._hx_area(h_hot_in, h_cold_in, m_dot_hot,
            m_dot_cold, Q_dot, P_hot, P_cold)
```

```
        self.penalty = self._check_hx_valid(self.T_h, self.T_c,
            self.T_pinch)

    def _hx_area(self, h_hot_in, h_cold_in, m_dot_hot, m_dot_cold
        , Q_dot, P_hot, P_cold):
        # Solve states for hot and cold sides at their inputs
        self.h_h[-1] = h_hot_in
        self.h_fld.update(CP.HmassP_INPUTS, h_hot_in, P_hot)
        self.T_h[-1] = self.h_fld.T()
        self.phase_h[-1] = self.h_fld.phase()

        self.h_c[0] = h_cold_in
        self.c_fld.update(CP.HmassP_INPUTS, h_cold_in, P_cold)
        self.T_c[0] = self.c_fld.T()
        self.phase_c[0] = self.c_fld.phase()

        # Find specific heat transferred on each side and the
            corresponding enthalpy change
        q_h = Q_dot / m_dot_hot
        q_c = Q_dot / m_dot_cold
        deltah_h = q_h / self.N_area
        deltah_c = q_c / self.N_area

        for j in range(1, self.N_area + 1):
            self.h_h[-j - 1] = self.h_h[-j] - deltah_h
            self.h_fld.update(CP.HmassP_INPUTS, self.h_h[-j - 1],
                P_hot)
            self.T_h[-j - 1] = self.h_fld.T()
```

```
            self.phase_h[-j - 1] = self.h_fld.phase()

            self.h_c[j] = self.h_c[j - 1] + deltah_c
            self.c_fld.update(CP.HmassP_INPUTS, self.h_c[j],
                P_cold)
            self.T_c[j] = self.c_fld.T()
            self.phase_c[j] = self.c_fld.phase()

        area = 0

        for k in range(self.N_area):
            U_o = self._get_u(self.phase_h[k], self.phase_c[k],
                P_hot, P_cold)
            deltaT_lm = self._log_mean_temp(self.T_h[k + 1], self
                .T_h[k], self.T_c[k], self.T_c[k + 1])
            deltaQ_dot = m_dot_hot * (self.h_h[k + 1] - self.h_h[
                k])
            if U_o * deltaT_lm != 0:
                area += deltaQ_dot / (U_o * deltaT_lm)

        return area

    def _get_u(self, phase_h, phase_c, P_h, P_c):
        d_o = 0.033401  # [m] tube outside diameter
        d_i = 0.0266446  # [m] tube inside diameter
        k_w = 16.3  # [W/m-K] approximate metal conductivity at
            average temperature
```

```
        h_h = self._conv_coeff(phase_h, 0, P_h)
        h_c = self._conv_coeff(phase_c, 1, P_c)

        h_o, h_i = (h_c, h_h) if h_h > h_c else (h_h, h_c)

        u = 1 / (1 / h_o + d_o * np.log(d_o / d_i) / (2 * k_w) +
            d_o / (d_i * h_i))

        return u

    @staticmethod
    def _check_hx_valid(T_h, T_c, T_pinch):
        deltaT = T_c - T_h + T_pinch
        penalty = 0 if ((T_h[-1] - T_h[0] < T_pinch) and (T_c[-1]
            - T_c[0] < T_pinch)) else max(max(deltaT), 0)
        return penalty

    @staticmethod
    def _conv_coeff(phase, side, P):
        p = P * 1e-6  # convert to [MPa] so I don't have to
            figure out what's going on with units with these
            interpolations...
        if phase in [1, 2, 3, 4,
                     5]:  # is supercritical/saturated vapor/
                          superheated gas ("sensible gas"); only
                          leaves twophase for other cases
            if p < 0.2:
                h = 0.1
```

```
            elif p < 1:
                h = 0.1 * (1 - (p - 0.2) / 0.8) + 0.325 * (p -
                    0.2) / 0.8
            elif p < 10:
                h = 0.325 * (1 - (p - 1) / 9) + 0.625 * (p - 1) /
                    9
            else:
                h = 0.625
        elif side == 0:  # hot side ("condensing")
            if p < 0.01:
                h = 1.75
            elif p < 0.1:
                h = 1.75 * (1 - (p - 0.01) / 0.09) + 3 * (p -
                    0.01) / 0.09
            elif p < 1:
                h = 3 * (1 - (p - 0.1) / 0.9) + 3.5 * (p - 0.1) /
                    9
            else:
                h = 3.5
        else:  # cold side ("boiling")
            h = 1.75

        return h * 1e3  # convert to [W/m^2-K]

    @staticmethod
    def _log_mean_temp(Thi, Tho, Tci, Tco):
        dT1 = Thi - Tco
        dT2 = Tho - Tci
```

```
if abs(dT1 - dT2) < 1e-8 or abs((dT1 - dT2) / dT1) < 1e
    -8:
     return Thi - Tci
else:
     return (dT1 - dT2) / np.log(dT1 / dT2)
```

APPENDIX D

Optimization

In Appendix D, all Python files under the **optimization** directory in Fig. A.1 are presented. The **optimization** directory includes all optimization methods for the surrogate model of the Integrated Regenerative Methanol Transcritical Cycle.

D.1 basinhopping.py

```
"""
Import libraries
"""
from scipy.optimize import basinhopping
import torch
import numpy as np
from Modules import Module_FFN_LCOE_penalty
from Modules import Module_BinaryClassifierFNN
from Modules import Module_FFN_eta_I


class Basinhopping:
    """
    Optimize LCOE+penalty with Basinhopping algorithm.

    @author: Yili Zhang
    @date: 8/22/2020
    """

    def __init__(self):
```

```
        self.device = torch.device('cuda' if torch.cuda.
            is_available() else 'cpu')

        # FNN_LCOE_penalty Model Hyper-parameters
        self.input_size_LCOE_penalty = 7
        self.hidden_size1_LCOE_penalty = 256
        self.hidden_size2_LCOE_penalty = 128
        self.num_out_LCOE_penalty = 1
        self.dropout_rate_LCOE_penalty = 0

        # FFN_BinaryClassifier Model Hyper-parameters
        self.input_size_BinaryClassifierFNN = 7
        self.hidden_size1_BinaryClassifierFNN = 256
        self.hidden_size2_BinaryClassifierFNN = 128
        self.num_out_BinaryClassifierFNN = 1
        self.dropout_rate_BinaryClassifierFNN = 0

        # FFN_eta_I Model Hyper-parameters
        self.input_size_eta_I = 7
        self.hidden_size1_eta_I = 256
        self.hidden_size2_eta_I = 128
        self.num_out_eta_I = 1
        self.dropout_rate_eta_I = 0.3

    def solve(self):
        """
        Solve the optimization problem with the Basin-hopping
            Algorithm
```

```
        """
        x0 = [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]  # Initial Guess
        ret = basinhopping(self.func, x0, niter=500, stepsize
            =0.1, interval=50)  # Basin-hopping
        p_max = round(np.abs(ret.x[0]), 2)
        pr1 = round(np.abs(ret.x[1]), 4)
        pr2 = round(np.abs(ret.x[2]), 4)
        pr3 = round(np.abs(ret.x[3]), 4)
        f1 = round(np.abs(ret.x[4]), 4)
        f2 = round(np.abs(ret.x[5]), 4)
        f3 = round(np.abs(ret.x[6]), 4)
        f_surrogate = round(ret.fun, 4)

        eta_FL_surrogate = self.get_effi(np.abs(ret.x))
        return p_max, pr1, pr2, pr3, f1, f2, f3, f_surrogate,
            eta_FL_surrogate

    def FFN_LCOE_penalty(self):
        """
        Load the FFN_LCOE_penalty surrogate model
        """
        FFN_LCOE_penalty = Module_FFN_LCOE_penalty.MyModule(self.
            input_size_LCOE_penalty, self.
            hidden_size1_LCOE_penalty, self.
            hidden_size2_LCOE_penalty, self.num_out_LCOE_penalty,
            self.dropout_rate_LCOE_penalty)  # Load model
            parameters
```

```
    model_path_LCOE_etaI = './Paths/
        FFN_LCOE_penalty_samllerRange.pth'
    FFN_LCOE_penalty.load_state_dict(torch.load(
        model_path_LCOE_etaI, map_location=self.device))
    FFN_LCOE_penalty.eval()  # for restore the batch
        normalization and drop out info.
    return FFN_LCOE_penalty


def FFN_eta_I(self):
    """
    Load the FFN_eta_I surrogate model
    """
    FFN_eta_I = Module_FFN_eta_I.MyModule(self.
        input_size_eta_I, self.hidden_size1_eta_I, self.
        hidden_size2_eta_I, self.num_out_eta_I, self.
        dropout_rate_eta_I)
    model_path_eta_I = './Paths/FFN_eta_I.pth'
    FFN_eta_I.load_state_dict(torch.load(model_path_eta_I,
        map_location=self.device))
    FFN_eta_I.eval()  # for restore the batch normalization
        and drop out info.
    return FFN_eta_I


def BinaryClassifierFNN(self):
    """
    Load the BinaryClassifierFNN surrogate model in order to
        distinguish between the converged and the diverged
```

```
        design parameters.
        """
        BinaryClassifierFNN = Module_BinaryClassifierFNN.MyModule
            (self.input_size_BinaryClassifierFNN, self.
            hidden_size1_BinaryClassifierFNN, self.
            hidden_size2_BinaryClassifierFNN, self.
            num_out_BinaryClassifierFNN, self.
            dropout_rate_BinaryClassifierFNN).to(self.device)
        model_path_BinaryClassifierFNN = './Paths/
            BinaryClassifierFNN.pth'
        BinaryClassifierFNN.load_state_dict(torch.load(
            model_path_BinaryClassifierFNN, map_location=self.
            device))
        BinaryClassifierFNN.eval()  # for restore the batch
            normalization and drop out info.
        return BinaryClassifierFNN


    def data_clearning(self, Dataset, Col_Name, LThre=None, HThre
        =None):
        """
        Data Clearning: Clearn the Outliers in the dataset:
            Eliminate all samples with the first law efficiency
            lower than 20%
        """
        if (LThre == None) and (HThre == None):
          newData = Dataset
        elif HThre == None:
          newData = Dataset[Dataset[Col_Name]>LThre]
```

```
        elif LThre == None:
            newData = Dataset[Dataset[Col_Name]<HThre]
        else:
            newData = Dataset[Dataset[Col_Name]<HThre]
            newData = newData[newData[Col_Name]>LThre]
        return newData

    def col_Drop(self, Dataset, Columns_to_drop):
        """
        Drop Undesired columns in the dataset
        """
        newData = Dataset.drop(columns=Columns_to_drop)
        return newData

    def data_prep(self, design_params):
        """
        Function: Data preparation
        """
        feature = design_params.reshape(1, -1)  # Transfer data
            from numpy array to tensor
        feature = torch.tensor(feature)
        return feature  # Return tranformed sample

    def get_Multi_Task_LCOE_penalty(self, design_params):
        """
        Function: Get outcome from new set of input sample
        """
```

```
    feature = self.data_prep(design_params)  # Data
        preparation
    feature = feature.to(self.device)  # Move tensors to the
        configured device

    FFN_LCOE_penalty = self.FFN_LCOE_penalty()
    Multi_Task_prelim = FFN_LCOE_penalty(feature.float())  #
        Forward pass
    Multi_Task_prelim = Multi_Task_prelim.cpu().detach().
        numpy()  # Inverse Transform LCOE to its original
        value

    target = Multi_Task_prelim[0, 0].reshape(1, -1)
    return target[0, 0]  # Return output

def get_effi(self, design_params):
    # Data preparation
    feature = self.data_prep(design_params)

    # Move tensors to the configured device
    feature = feature.to(self.device)

    # Forward pass
    FFN_eta_I = self. FFN_eta_I()
    effi = FFN_eta_I(feature.float())
    effi = effi.cpu().detach().numpy()

    # Return output
```

```
        effi = effi[0, 0]
        return effi

    def if_converged(self, design_params):
        """
        Data preparation: Transfer data from numpy array to
            tensor
        """
        feature = design_params.reshape(1, -1)
        feature = torch.tensor(feature)

        # Move tensors to the configured device
        feature = feature.to(self.device)

        # Forward pass
        BinaryClassifierFNN = self.BinaryClassifierFNN()
        converged = BinaryClassifierFNN(feature.float()).cpu().
            detach().numpy()[0,0].reshape(1, -1)

        # Determine if converged
        if (converged>0.5) == True:
          out = True
        else:
          out = False

        # Print the Results
        return out
```

```
    def func(self, x):
        """
        Objective Function
        """
        x = np.abs(x)  # Making sure the optimized variable
            finally being processed are positive values
        if self.if_converged(x) == True:
            target = self.get_Multi_Task_LCOE_penalty(x)
            f = target
        else:
            f = 1e10
        return f
```

D.2 dual_annealing.py

```
"""
Import libraries
"""
from scipy.optimize import dual_annealing
import torch
import numpy as np
from Modules import Module_FFN_LCOE_penalty
from Modules import Module_BinaryClassifierFNN
from Modules import Module_FFN_eta_I


class DualAnnealing:
    """
    Optimize LCOE+penalty with Dual Annealing algorithm.
```

```
@author: Yili Zhang
@date: 8/22/2020
"""

def __init__(self):
    self.device = torch.device('cuda' if torch.cuda.
        is_available() else 'cpu')

    # FNN_LCOE_penalty Model Hyper-parameters
    self.input_size_LCOE_penalty = 7
    self.hidden_size1_LCOE_penalty = 256
    self.hidden_size2_LCOE_penalty = 128
    self.num_out_LCOE_penalty = 1
    self.dropout_rate_LCOE_penalty = 0

    # FFN_BinaryClassifier Model Hyper-parameters
    self.input_size_BinaryClassifierFNN = 7
    self.hidden_size1_BinaryClassifierFNN = 256
    self.hidden_size2_BinaryClassifierFNN = 128
    self.num_out_BinaryClassifierFNN = 1
    self.dropout_rate_BinaryClassifierFNN = 0

    # FFN_eta_I Model Hyper-parameters
    self.input_size_eta_I = 7
    self.hidden_size1_eta_I = 256
    self.hidden_size2_eta_I = 128
    self.num_out_eta_I = 1
    self.dropout_rate_eta_I = 0.3
```

```
def solve(self):
    """
    Solve the optimization problem with the Dual Annealing
        Algorithm
    """
    bounds = [(0,1), (0,1), (0,1), (0,1), (0,1), (0,1), (0,1)
        ]
    ret = dual_annealing(self.func, bounds, maxiter=1000)
    p_max = round(np.abs(ret.x[0]), 2)
    pr1 = round(np.abs(ret.x[1]), 4)
    pr2 = round(np.abs(ret.x[2]), 4)
    pr3 = round(np.abs(ret.x[3]), 4)
    f1 = round(np.abs(ret.x[4]), 4)
    f2 = round(np.abs(ret.x[5]), 4)
    f3 = round(np.abs(ret.x[6]), 4)
    f_surrogate = round(ret.fun, 4)

    eta_FL_surrogate = self.get_effi(np.abs(ret.x))
    return p_max, pr1, pr2, pr3, f1, f2, f3, f_surrogate,
        eta_FL_surrogate

def FFN_LCOE_penalty(self):
    """
    Load the FFN_LCOE_penalty surrogate model
    """
    FFN_LCOE_penalty = Module_FFN_LCOE_penalty.MyModule(self.
        input_size_LCOE_penalty, self.
```

```
        hidden_size1_LCOE_penalty, self.
        hidden_size2_LCOE_penalty, self.num_out_LCOE_penalty,
        self.dropout_rate_LCOE_penalty)  # Load model
        parameters
    model_path_LCOE_etaI = './Paths/
        FFN_LCOE_penalty_samllerRange.pth'
    FFN_LCOE_penalty.load_state_dict(torch.load(
        model_path_LCOE_etaI, map_location=self.device))
    FFN_LCOE_penalty.eval()  # for restore the batch
        normalization and drop out info.
    return FFN_LCOE_penalty


def FFN_eta_I(self):
    """
    Load the FFN_eta_I surrogate model
    """
    FFN_eta_I = Module_FFN_eta_I.MyModule(self.
        input_size_eta_I, self.hidden_size1_eta_I, self.
        hidden_size2_eta_I, self.num_out_eta_I, self.
        dropout_rate_eta_I)
    model_path_eta_I = './Paths/FFN_eta_I.pth'
    FFN_eta_I.load_state_dict(torch.load(model_path_eta_I,
        map_location=self.device))
    FFN_eta_I.eval()  # for restore the batch normalization
        and drop out info.
    return FFN_eta_I
```

```
def BinaryClassifierFNN(self):
    """
    Load the BinaryClassifierFNN surrogate model in order to
        distinguish between the converged and the diverged
    design parameters.
    """
    BinaryClassifierFNN = Module_BinaryClassifierFNN.MyModule
        (self.input_size_BinaryClassifierFNN, self.
        hidden_size1_BinaryClassifierFNN, self.
        hidden_size2_BinaryClassifierFNN, self.
        num_out_BinaryClassifierFNN, self.
        dropout_rate_BinaryClassifierFNN).to(self.device)
    model_path_BinaryClassifierFNN = './Paths/
        BinaryClassifierFNN.pth'
    BinaryClassifierFNN.load_state_dict(torch.load(
        model_path_BinaryClassifierFNN, map_location=self.
        device))
    BinaryClassifierFNN.eval()  # for restore the batch
        normalization and drop out info.
    return BinaryClassifierFNN


def data_clearning(self, Dataset, Col_Name, LThre=None, HThre
    =None):
    """
    Data Clearning: Clearn the Outliers in the dataset:
        Eliminate all samples with the first law efficiency
        lower than 20%
    """
```

```
        if (LThre == None) and (HThre == None):
            newData = Dataset
        elif HThre == None:
            newData = Dataset[Dataset[Col_Name]>LThre]
        elif LThre == None:
            newData = Dataset[Dataset[Col_Name]<HThre]
        else:
            newData = Dataset[Dataset[Col_Name]<HThre]
            newData = newData[newData[Col_Name]>LThre]
        return newData

    def col_Drop(self, Dataset, Columns_to_drop):
        """
        Drop Undesired columns in the dataset
        """
        newData = Dataset.drop(columns=Columns_to_drop)
        return newData

    def data_prep(self, design_params):
        """
        Function: Data preparation
        """
        feature = design_params.reshape(1, -1)  # Transfer data
            from numpy array to tensor
        feature = torch.tensor(feature)
        return feature  # Return tranformed sample

    def get_Multi_Task_LCOE_penalty(self, design_params):
```

```
        """
        Function: Get outcome from new set of input sample
        """
        feature = self.data_prep(design_params)  # Data
            preparation
        feature = feature.to(self.device)  # Move tensors to the
            configured device

        FFN_LCOE_penalty = self.FFN_LCOE_penalty()
        Multi_Task_prelim = FFN_LCOE_penalty(feature.float())  #
            Forward pass
        Multi_Task_prelim = Multi_Task_prelim.cpu().detach().
            numpy()  # Inverse Transform LCOE to its original
            value

        target = Multi_Task_prelim[0, 0].reshape(1, -1)
        return target[0, 0]  # Return output

    def get_effi(self, design_params):
        # Data preparation
        feature = self.data_prep(design_params)

        # Move tensors to the configured device
        feature = feature.to(self.device)

        # Forward pass
        FFN_eta_I = self. FFN_eta_I()
        effi = FFN_eta_I(feature.float())
```

```
        effi = effi.cpu().detach().numpy()

        # Return output
        effi = effi[0, 0]
        return effi

    def if_converged(self, design_params):
        """
        Data preparation: Transfer data from numpy array to
            tensor
        """
        feature = design_params.reshape(1, -1)
        feature = torch.tensor(feature)

        # Move tensors to the configured device
        feature = feature.to(self.device)

        # Forward pass
        BinaryClassifierFNN = self.BinaryClassifierFNN()
        converged = BinaryClassifierFNN(feature.float()).cpu().
            detach().numpy()[0,0].reshape(1, -1)

        # Determine if converged
        if (converged>0.5) == True:
          out = True
        else:
          out = False
```

```
        # Print the Results
        return out


    def func(self, x):
        """
        Objective Function
        """
        x = np.abs(x)  # Making sure the optimized variable
            finally being processed are positive values
        if self.if_converged(x) == True:
            target = self.get_Multi_Task_LCOE_penalty(x)
            f = target
        else:
            f = 1e10
        return f
```

D.3 fmin.py

```
"""
Import libraries
"""
from scipy.optimize import fmin
import torch
import numpy as np
from random import random
from Modules import Module_FFN_LCOE_penalty
from Modules import Module_BinaryClassifierFNN
from Modules import Module_FFN_eta_I
```

```
class Fmin:
    """
    Optimize LCOE+penalty with fmin algorithm.

    @author: Yili Zhang
    @date: 8/22/2020
    """

    def __init__(self):
        self.device = torch.device('cuda' if torch.cuda.
            is_available() else 'cpu')

        # FNN_LCOE_penalty Model Hyper-parameters
        self.input_size_LCOE_penalty = 7
        self.hidden_size1_LCOE_penalty = 256
        self.hidden_size2_LCOE_penalty = 128
        self.num_out_LCOE_penalty = 1
        self.dropout_rate_LCOE_penalty = 0

        # FFN_BinaryClassifier Model Hyper-parameters
        self.input_size_BinaryClassifierFNN = 7
        self.hidden_size1_BinaryClassifierFNN = 256
        self.hidden_size2_BinaryClassifierFNN = 128
        self.num_out_BinaryClassifierFNN = 1
        self.dropout_rate_BinaryClassifierFNN = 0

        # FFN_eta_I Model Hyper-parameters
        self.input_size_eta_I = 7
```

```
        self.hidden_size1_eta_I = 256
        self.hidden_size2_eta_I = 128
        self.num_out_eta_I = 1
        self.dropout_rate_eta_I = 0.3

    def solve(self):
        """
        Solve the optimization problem with the fmin Algorithm
        """
        ret = self.iter_fmin(maxcheckpoints=100)
        p_max = round(np.abs(ret[0][0]), 2)
        pr1 = round(np.abs(ret[0][1]), 4)
        pr2 = round(np.abs(ret[0][2]), 4)
        pr3 = round(np.abs(ret[0][3]), 4)
        f1 = round(np.abs(ret[0][4]), 4)
        f2 = round(np.abs(ret[0][5]), 4)
        f3 = round(np.abs(ret[0][6]), 4)
        f_surrogate = round(ret[1], 4)

        eta_FL_surrogate = self.get_effi(np.abs(ret[0]))
        return p_max, pr1, pr2, pr3, f1, f2, f3, f_surrogate,
            eta_FL_surrogate

    def FFN_LCOE_penalty(self):
        """
        Load the FFN_LCOE_penalty surrogate model
        """
```

```
    FFN_LCOE_penalty = Module_FFN_LCOE_penalty.MyModule(self.
        input_size_LCOE_penalty, self.
        hidden_size1_LCOE_penalty, self.
        hidden_size2_LCOE_penalty, self.num_out_LCOE_penalty,
        self.dropout_rate_LCOE_penalty)  # Load model
        parameters
    model_path_LCOE_etaI = './Paths/
        FFN_LCOE_penalty_samllerRange.pth'
    FFN_LCOE_penalty.load_state_dict(torch.load(
        model_path_LCOE_etaI, map_location=self.device))
    FFN_LCOE_penalty.eval()  # for restore the batch
        normalization and drop out info.
    return FFN_LCOE_penalty


def FFN_eta_I(self):
    """
    Load the FFN_eta_I surrogate model
    """
    FFN_eta_I = Module_FFN_eta_I.MyModule(self.
        input_size_eta_I, self.hidden_size1_eta_I, self.
        hidden_size2_eta_I, self.num_out_eta_I, self.
        dropout_rate_eta_I)
    model_path_eta_I = './Paths/FFN_eta_I.pth'
    FFN_eta_I.load_state_dict(torch.load(model_path_eta_I,
        map_location=self.device))
    FFN_eta_I.eval()  # for restore the batch normalization
        and drop out info.
    return FFN_eta_I
```

```
def BinaryClassifierFNN(self):
    """
    Load the BinaryClassifierFNN surrogate model in order to
        distinguish between the converged and the diverged
    design parameters.
    """
    BinaryClassifierFNN = Module_BinaryClassifierFNN.MyModule
        (self.input_size_BinaryClassifierFNN, self.
        hidden_size1_BinaryClassifierFNN, self.
        hidden_size2_BinaryClassifierFNN, self.
        num_out_BinaryClassifierFNN, self.
        dropout_rate_BinaryClassifierFNN).to(self.device)
    model_path_BinaryClassifierFNN = './Paths/
        BinaryClassifierFNN.pth'
    BinaryClassifierFNN.load_state_dict(torch.load(
        model_path_BinaryClassifierFNN, map_location=self.
        device))
    BinaryClassifierFNN.eval()  # for restore the batch
        normalization and drop out info.
    return BinaryClassifierFNN


def data_clearning(self, Dataset, Col_Name, LThre=None, HThre
    =None):
    """
```

```
        Data Clearning: Clearn the Outliers in the dataset:
            Eliminate all samples with the first law efficiency
            lower than 20%
        """
        if (LThre == None) and (HThre == None):
          newData = Dataset
        elif HThre == None:
          newData = Dataset[Dataset[Col_Name]>LThre]
        elif LThre == None:
          newData = Dataset[Dataset[Col_Name]<HThre]
        else:
          newData = Dataset[Dataset[Col_Name]<HThre]
          newData = newData[newData[Col_Name]>LThre]
        return newData


    def col_Drop(self, Dataset, Columns_to_drop):
        """
        Drop Undesired columns in the dataset
        """
        newData = Dataset.drop(columns=Columns_to_drop)
        return newData


    def data_prep(self, design_params):
        """
        Function: Data preparation
        """
        feature = design_params.reshape(1, -1)  # Transfer data
            from numpy array to tensor
```

```
        feature = torch.tensor(feature)
        return feature  # Return tranformed sample

    def get_Multi_Task_LCOE_penalty(self, design_params):
        """
        Function: Get outcome from new set of input sample
        """
        feature = self.data_prep(design_params)  # Data
            preparation
        feature = feature.to(self.device)  # Move tensors to the
            configured device

        FFN_LCOE_penalty = self.FFN_LCOE_penalty()
        Multi_Task_prelim = FFN_LCOE_penalty(feature.float())  #
            Forward pass
        Multi_Task_prelim = Multi_Task_prelim.cpu().detach().
            numpy()  # Inverse Transform LCOE to its original
            value

        target = Multi_Task_prelim[0, 0].reshape(1, -1)
        return target[0, 0]  # Return output

    def get_effi(self, design_params):
        # Data preparation
        feature = self.data_prep(design_params)

        # Move tensors to the configured device
        feature = feature.to(self.device)
```

```
        # Forward pass
        FFN_eta_I = self. FFN_eta_I()
        effi = FFN_eta_I(feature.float())
        effi = effi.cpu().detach().numpy()

        # Return output
        effi = effi[0, 0]
        return effi

    def if_converged(self, design_params):
        """
        Data preparation: Transfer data from numpy array to
            tensor
        """
        feature = design_params.reshape(1, -1)
        feature = torch.tensor(feature)

        # Move tensors to the configured device
        feature = feature.to(self.device)

        # Forward pass
        BinaryClassifierFNN = self.BinaryClassifierFNN()
        converged = BinaryClassifierFNN(feature.float()).cpu().
            detach().numpy()[0,0].reshape(1, -1)

        # Determine if converged
        if (converged>0.5) == True:
```

```
            out = True
        else:
            out = False

        # Print the Results
        return out

    def func(self, x):
        """
        Objective Function
        """
        x = np.abs(x)  # Making sure the optimized variable
            finally being processed are positive values
        if self.if_converged(x) == True:
            target = self.get_Multi_Task_LCOE_penalty(x)
            f = target
        else:
            f = 1e10
        return f

    def iter_fmin(self, maxcheckpoints=1000):
        x = np.arange(7)  # initial row in the x matrix
        f = np.arange(1)  # initial row in the objective function
            vertical vector

        for i in range(maxcheckpoints):
            # fmin Algorithm
```

```
        x0 = [random(), random(), random(), random(), random
            (), random(), random()] # Initial Guess
        ret = fmin(self.func, x0, xtol=0.0001, ftol=0.0001,
            maxiter=500, maxfun=500, full_output=True, disp=
            False)

        # Return Optimization Results
        x_opt = ret[0]
        func_opt = ret[1]

        # attach x_opt array and the func_opt value to the x
            and f array
        x = np.vstack((x, x_opt))
        f = np.vstack((f, func_opt))

    # Delete the first rows
    x = np.delete(x, 0, 0)
    f = np.delete(f, 0, 0)

    # Find the minimum f value & its index & corresponding x
        values
    f_final_opt = min(f)[0]
    f_final_opt_index = np.argmin(f)
    x_final_opt = x[f_final_opt_index, :]
    return x_final_opt, f_final_opt
```

APPENDIX E

Modules

In Appendix E, all Python files under the **Modules** directory in Fig. A.1 are presented. The **Modules** directory includes all files of Machine Learning patterns/structures of the corresponding surrogate models of the Integrated Regenerative Methanol Transcritical Cycle for quickly finding the desired output parameters.

E.1 Module_FFN_LCOE_penalty.py

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

'''
Network Model: Fully connected neural network with four hidden
    layer
'''
class MyModule(nn.Module):

  def __init__(self, input_size, hidden_size1, hidden_size2,
      num_out, dropout_rate):
    super(MyModule, self).__init__()
    self.fc1 = nn.Linear(input_size,hidden_size1)
    self.fc2 = nn.Linear(hidden_size1,hidden_size2)
    self.dropout = nn.Dropout(p=dropout_rate)
    self.output = nn.Linear(hidden_size2,num_out)
```

```
        self.m1 = nn.BatchNorm1d(hidden_size1)
        self.m2 = nn.BatchNorm1d(hidden_size2)

    def forward(self, x):
        # x = F.relu(self.fc1(x))
        x = F.leaky_relu(self.m1(self.fc1(x)))
        x = self.dropout(x)
        # x = F.relu(self.fc2(x))
        x = F.leaky_relu(self.m2(self.fc2(x)))
        x = self.dropout(x)
        x = F.leaky_relu(self.output(x))
        return x

    def init_weight(x):
        if type(x) == nn.Linear:
            torch.nn.init.xavier_uniform(x.weight)
            x.bias.data.fill_(0.01)
```

E.2 Module_FFN_eta_I.py

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


'''
Network Model: Fully connected neural network with four hidden
    layer
'''
```

```
class MyModule(nn.Module):

  def __init__(self, input_size, hidden_size1, hidden_size2,
      num_out, dropout_rate):
    super(MyModule, self).__init__()
    self.fc1 = nn.Linear(input_size,hidden_size1)
    self.fc2 = nn.Linear(hidden_size1,hidden_size2)
    self.dropout = nn.Dropout(p=dropout_rate)
    self.output = nn.Linear(hidden_size2,num_out)
    self.m1 = nn.BatchNorm1d(hidden_size1)
    self.m2 = nn.BatchNorm1d(hidden_size2)

  def forward(self, x):
    x = F.relu(self.m1(self.fc1(x)))
    x = self.dropout(x)
    x = F.relu(self.m2(self.fc2(x)))
    x = self.dropout(x)
    x = F.relu(self.output(x))
    return x

  def init_weight(x):
    if type(x) == nn.Linear:
      torch.nn.init.xavier_uniform(x.weight)
      x.bias.data.fill_(0.01)
```

E.3 Module_BinaryClassifierFNN.py

```
import torch.nn as nn
import torch
```

```
import torch.nn.functional as F


'''
Network Model: Fully connected neural network with four hidden
    layer
'''
class MyModule(nn.Module):

  def __init__(self, input_size, hidden_size1, hidden_size2,
      num_out, dropout_rate):
    super(MyModule, self).__init__()
    self.fc1 = nn.Linear(input_size,hidden_size1)
    self.fc2 = nn.Linear(hidden_size1,hidden_size2)
    self.dropout = nn.Dropout(p=dropout_rate)
    self.output = nn.Linear(hidden_size2,num_out)
    self.m1 = nn.BatchNorm1d(hidden_size1)
    self.m2 = nn.BatchNorm1d(hidden_size2)


  def forward(self, x):
    # x = F.relu(self.fc1(x))
    x = F.relu(self.m1(self.fc1(x)))
    x = self.dropout(x)
    # x = F.relu(self.fc2(x))
    x = F.relu(self.m2(self.fc2(x)))
    x = self.dropout(x)
    x = torch.sigmoid(self.output(x))
    return x
```

```
def init_weight(x):
    if type(x) == nn.Linear:
        torch.nn.init.xavier_uniform(x.weight)
        x.bias.data.fill_(0.01)
```

APPENDIX F

Paths

The **Paths** directory in Fig. A.1 includes all the PTH files (.pth) that store the learned parameters for the corresponding modules in Appendix. E. Due to the difficulty of showing the content of the .pth files explicitly, please find the corresponding .pth files in my Github: https://github.com/Yili-Zhang/Dissertation-Path-Files