Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

Muhammad Gibran Alfarizi

Well Control Optimization in Waterflooding Using Genetic Algorithm Coupled with Machine Learning Models

Master's thesis in Petroleum Engineering
Supervisor: Professor Milan Stanko
June 2020

NTNU
Kunnskap for en bedre verden

Muhammad Gibran Alfarizi

Well Control Optimization in Waterflooding Using Genetic Algorithm Coupled with Machine Learning Models

Master's thesis in Petroleum Engineering
Supervisor: Professor Milan Stanko
June 2020

Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

”Essentially, all models are wrong, but some are useful.” - George Box.

Summary

Generally, optimum well controls to maximize net present value (NPV) in a waterflooding operation are obtained from a numerical reservoir simulation in combination with an optimization algorithm. This procedure is often computationally expensive and time-consuming to run because of the complexity in numerical reservoir simulation. This complexity also poses a challenge to implement gradient-based optimization algorithms, as there are multiple variables involved and numerous simulations are needed for model evaluations.

This thesis proposes using a machine learning model, specifically an Artificial Neural Network (ANN), to replicate the numerical reservoir simulator outputs. The ANN model is used to predict cumulative oil production, cumulative water injection, and cumulative water production based on sets of well control values, i.e., flowing bottom-hole pressure. Then, the ANN model will be combined with a genetic algorithm (GA) optimization (a derivative-free optimization) to find the optimum well controls that maximize the NPV of a synthetic reservoir model. The optimization results of this model were compared against the results using the open-source FieldOpt software, that optimizes well control values using the genetic algorithm and the reservoir model.

The data generated from reservoir simulation was used as the building blocks to the machine learning model. Hyperparameters optimization was done to create the best machine learning architecture. Several variables were also tested to find the best configuration for the genetic algorithm optimization. The developed ANN model was capable of reproducing the results of the original reservoir model within an acceptable accuracy (1.89%). The genetic algorithm improved the project NPV successfully from the base case and achieved similar results to FieldOpt but 43 hours faster.

Preface

This thesis is submitted in partial fulfilment of the requirements for the MSc degree in Reservoir Engineering & Petrophysics at Departement of Geoscience and Petroleum - Norwegian University of Science and Technology (NTNU).

I would like to express my gratitude to PhD candidate Timur Bikmukhametov for introducing me to machine learning and guiding me as a friend during the summer intern at SUBPRO up until now. I also would like to thank my supervisor, Professor Milan Edvard Wolf Stanko, for his meaningful insight and guidance during my final year as a master's student in NTNU. Moreover, I thank my colleagues and the academic community at the Department of Geoscience and Petroleum for providing a supportive and excellent working environment.

I am deeply grateful to my family in Indonesia for their unconditional support during my study in Trondheim. I finally want to thank my girlfriend, Dea Lana Asri, for her support, attention, love, and happiness that she brings into my life.

Trondheim, June 2020
Muhammad Gibran Alfarizi

Table of Contents

List of Tables

List of Figures

Abbreviations

AI	=	Artificial Intelligence
ANN	=	Artificial Neural Network
BHP	=	Bottom Hole Pressure
GA	=	Genetic Algorithm
ML	=	Machine Learning
MSE	=	Mean Squared Error
NPV	=	Net Present Value
NRMSE	=	Normalized Root Mean Squared Error
OPM	=	Open Porous Media
PVT	=	Pressure-Volume-Temperature
ReLU	=	Rectified Linear Units
SGD	=	Stochastic Gradient Descent
TPE	=	Tree-structured Parzen Estimator

Chapter 1

Introduction

This opening chapter presents a brief explanation of the motivation behind this master's thesis and its objective.

1.1 Background

Primary recovery is the first step in oil and gas production. Crude oil extraction from a new well relies on the natural rise of the oil due to the differences in pressure between the separator and the reservoir. As the pressure at the reservoir decreases because of the oil extraction over time, the primary recovery eventually cannot provide an economically attractive oil recovery. Thus, a secondary recovery method is needed.

Secondary recovery is the oil recovery technique where gas or water is injected to maintain the reservoir pressure (Khan and Islam, 2007). The most common method of secondary recovery in oil fields is waterflooding. This method involves injecting water from injection wells. The water then physically sweeps the displaced oil to adjacent production wells. This process pushes more residual oil into the production tubing.

In order to maximize total oil production, or more importantly, the net present value (NPV) of the oil field, one needs to determine the optimum well control values in waterflooding (liquid rate or bottom-hole pressure of injection and producer wells). Generally, these optimum well control values are obtained from a numerical reservoir simulator in combination with an optimizer. This procedure is often computationally expensive and time-consuming to run because a numerical reservoir simulation is a complex process of solving fluid flow equations and consider lots of information, including geological information, rock and fluid properties, and well completion. This complexity also presents a challenge to implement gradient-based optimization algorithms, as the estimation of the objective function with respect to well control values involves multiple variables and multiple simulations for model evaluations. A single model evaluation may take many hours for a grid with a vast number of cells (Sarma et al., 2005).

This thesis proposes the use of a machine learning model, specifically the Artificial Neural Networks (ANN), to describe the nonlinear behavior of the reservoir instead of the numerical reservoir simulator. The ANN was built upon a synthetic reservoir simulation model. The proposed model, combined with a derivative-free optimization algorithm called genetic algorithm, will determine the optimum well control values. The run time of the proposed model will also be compared with the run time of FieldOpt, a software which could optimize the well control values using genetic algorithm in combination with numerical reservoir simulator.

This thesis also presented a broad literature review of the machine learning model used in this study. This with the intention that a person without a proper background in machine learning, can still understand the study that has been performed.

1.2 Objective

The master's thesis's main objective is to develop an ANN model that replicates the numerical reservoir simulator outputs. This model has to be robust enough to predict cumulative oil production, cumulative water injection, and cumulative water production based on sets of well control values. This model will be combined with a genetic optimization algorithm to find the optimum well control values that maximize the NPV of a synthetic reservoir model.

These tasks are performed to approach the main objective:

1. Create a synthetic field with a complete reservoir and well models.
2. Build a dataset consisting of the necessary information extracted from the reservoir simulator and then divide it into training, validation, and test dataset.
3. Build the ANN model to predict cumulative oil production, cumulative water production, and cumulative water injection based on sets of well control values.
4. Evaluate the ANN model's performance with an appropriate metric.
5. Set base-case well control values for the reservoir model and calculate its NPV.
6. Find the optimum well control values that maximize NPV of the synthetic reservoir field with the combination of genetic optimization algorithm and the ANN model.
7. Simulate the optimum well control values obtained from the optimizer in the numerical reservoir simulator and calculate its NPV.
8. Compare the optimum NPV with the base case NPV. Analyze the result.
9. Compare the run time between the ANN-GA combination and the FieldOpt operation. Analyze the result.

1.3 Outline

This thesis is presented in five chapters. Introduction, as the first chapter, gives the motivation upon the thesis and its objective. The second chapter is Theory, which explains the underlying theories involved in the thesis. The third chapter is Methodology, which tells more in detail on how the tasks mentioned before are performed to reach the main objective. Then, the results from the thesis are presented and discussed in chapter four, Results and Discussion. Finally, in the fifth chapter, Conclusion and Recommendation, the results are concluded, and ideas for future researches are provided.

Chapter 2

Theory

This chapter covers the theory in reservoir simulation, artificial intelligence, and genetic algorithm.

2.1 Reservoir Simulation[1]

Reservoir simulation is the use of computers to numerically solve the fluid flow equations in a reservoir model (Aziz and Settari, 1979). The purpose of reservoir simulation is to mimic the behavior of the considered reservoir. Therefore, a good characterization of the reservoir is essential to achieve accurate results.

Many of the reservoir modeling disciplines, such as geologists, geophysicists, petrophysicists, and reservoir engineers, are involved in reservoir simulation. It is the product of the acquisition and interpretation of data from these disciplines. Expertise in mathematics, physics, and computer programming is also required to build a reservoir simulation model. The need for broad knowledge in the development of reservoir simulation can be seen as both a challenge and a reward. The challenge comes from getting these disciplines together, and as for the reward, it gives a thorough overview of the workflows toward reservoir management decisions (Berg, 2019).

Based on how the reservoir simulators handle fluid composition, it can be classified into two main categories. **Black-oil** simulators assume a constant composition of all phases, but it still allows for dropout of condensate from oil or gas dissolved in oil. **Compositional** simulators allow changes in the composition of the fluid phases based on an underlying equation of state.

Reservoir simulation software, often called reservoir simulator, usually employ finite difference methods for solving fluid flow equations. Those equations are conservation of

[1]This section has been presented in the author's specialization project.

mass Eq. (2.1), the extended Darcy theory for fluid flow through porous media Eq. (2.2), and fluid phase behavior (PVT). Below ρ is the fluid density, ϕ is the porosity, q is the volumetric fluid flux, k is the permeability, μ is the viscosity, and p is the pressure.

”mass in” − ”mass out” = ”change in mass”

$$-\nabla \cdot (\rho q) = \frac{\partial}{\partial t}(\phi \rho) \quad , \tag{2.1}$$

$$q = -\frac{k}{\mu}\nabla p \quad . \tag{2.2}$$

Inserting Eq. (2.2) into Eq. (2.1) and asusuming constant permeability, k, and pressure independent viscosity, μ, one can get

$$\frac{k}{\mu}\nabla \cdot (\rho \nabla p) = \frac{\partial}{\partial t}(\phi \rho) \quad . \tag{2.3}$$

Compressibility, c, is a measure of the relative volume change as a response to a pressure change. Liquid compressibility, c_l, formation compressibility, c_ϕ, and total compressibility can be expressed as

$$c = -\frac{1}{\rho}\frac{\partial \rho}{\partial p} \quad , \tag{2.4}$$

$$c_\phi = \frac{1}{\phi}\frac{\partial \phi}{\partial p} \quad , \tag{2.5}$$

$$c_t = c_l + c_\phi \quad . \tag{2.6}$$

Inserting Eq. (2.4), Eq. (2.5), and Eq. (2.6) into Eq. (2.3) and applying the chain rule, the equation derived into

$$\frac{k}{\mu \phi c_t}\nabla^2 p = \frac{\partial}{\partial t}p \quad . \tag{2.7}$$

If one only consider x-direction, the Eq. (2.7) becomes

$$\eta \frac{\partial^2 p}{\partial x^2} = \frac{\partial p}{\partial t} \quad , \tag{2.8}$$

where

$$\eta = \frac{k}{\mu \phi c_t} \quad .$$

To solve the two derivative terms $\frac{\partial^2 p}{\partial x^2}$ and $\frac{\partial p}{\partial t}$ in Eq. (2.8), the finite difference method is used to replace the partial derivatives by finite difference quotients and solve the resulting linear algebraic system. Our system needs to be discretized both spatially and in time. First, the x-coordinate must be subdivided into several discrete grid blocks, and the time coordinate must be divided into discrete time steps. Then, the pressure in each block can be solved numerically for each time step.

OPM Flow (Rasmussen et al., 2019), a black-oil simulator, is used for this study. The underlying principles from this simulator are explained in the subsequent subsection.

2.1.1 Black-oil Model

The black-oil model has three different fluid phases (aqueous, oleic, and gaseous) and three different components (water, oil, and gas). This model allows the mixing of oil and gas, i.e., both oil and gas can be found in the oleic phase, in the gaseous phase, or both. The quantities of dissolved gas in the oleic phase and vaporized oil in the gaseous phase must be tracked. The subscripts w, o, g will be used to indicate each component's quantities.

Model Equations

The black-oil model equations are deduced from conservation of mass for each component coupled with Darcy's Law and initial and boundary conditions. The equations are discretized in space and in time using an implicit Euler scheme. The resulting equations are then solved simultaneously in a fully implicit fashion.

Continuous equations. The conservation of mass for each component α, forms a system of partial differential equations:

$$\frac{\partial}{\partial t}(\phi_{\text{ref}} A_\alpha) + \nabla \cdot \mathbf{u}_\alpha + q_\alpha = 0 \quad , \tag{2.9}$$

where the accumulation terms and fluxes are given by

$$A_w = m_\phi b_w s_w \quad , \qquad \mathbf{u}_w = b_w \mathbf{v}_w \quad , \tag{2.10a}$$
$$A_o = m_\phi (b_o s_o + r_{og} b_g s_g) \quad , \qquad \mathbf{u}_o = b_o \mathbf{v}_o + r_{og} b_g \mathbf{v}_g \quad , \tag{2.10b}$$
$$A_g = m_\phi (b_g s_g + r_{go} b_o s_o) \quad , \qquad \mathbf{u}_g = b_g \mathbf{v}_g + r_{go} b_o \mathbf{v}_o \quad . \tag{2.10c}$$

The phase fluxes, $\mathbf{v}_\alpha$, given by Darcy's law:

$$\mathbf{v}_\alpha = -\lambda_\alpha \mathbf{K}(\nabla p_\alpha - \rho_\alpha \mathbf{g}) \quad . \tag{2.11}$$

Furthermore, these relations should hold:

$$s_w + s_o + s_g = 1 \quad , \tag{2.12}$$
$$p_{c,ow} = p_o - p_w \quad , \tag{2.13}$$
$$p_{c,og} = p_o - p_g \quad . \tag{2.14}$$

See List of Symbols for the definition of the symbols.

Discrete equations. Let subscript i represent a discrete quantity defined in cell i and subscript ij represent a discrete quantity defined at the connection between cell i and j. Every quantity is taken at the end of the time step, expect the one with superscript 0, which means it's taken at the start of the time step. For each component α and cell i, the discretized equations and residuals are:

$$R_{\alpha,i} = \frac{\phi_{\text{ref},i} V_i}{\Delta t}(A_{\alpha,i} - A^0_{\alpha,i}) + \sum_{j \in C(i)} u_{\alpha,ij} + q_{\alpha,i} = 0 \quad , \tag{2.15}$$

where A_α are as in Eq. (2.10) and u_α given by:

$$u_w = b_w v_w \quad , \tag{2.16a}$$
$$u_o = b_o v_o + r_{og} b_g v_g \quad , \tag{2.16b}$$
$$u_g = b_g v_g + r_{go} b_o v_o \quad . \tag{2.16c}$$

The relations in Eq. (2.12), (2.13), and (2.14) are also hold for every cell i. For each connection ij, the fluxes are:

$$(b_\alpha v_\alpha)_{ij} = (b_\alpha \lambda_\alpha m_T)_{U(\alpha,ij)} T_{ij} \Delta\Phi_{\alpha,ij} \quad , \tag{2.17}$$
$$(r_{\beta\alpha} b_\alpha v_\alpha)_{ij} = (r_{\beta\alpha} b_\alpha \lambda_\alpha m_T)_{U(\alpha,ij)} T_{ij} \Delta\Phi_{\alpha,ij} \quad , \tag{2.18}$$
$$\Delta\Phi_{\alpha,ij} = p_{\alpha,i} - p_{\alpha,j} - g\rho_{\alpha,ij}(z_i - z_j) \quad , \tag{2.19}$$
$$\rho_{\alpha,ij} = (\rho_{\alpha,i} + \rho_{\alpha,j})/2 \quad , \tag{2.20}$$
$$U(\alpha, ij) = \begin{cases} i, & \Delta\Phi_{\alpha,ij} \geq 0 \quad , \\ j, & \Delta\Phi_{\alpha,ij} < 0 \quad . \end{cases} \tag{2.21}$$

See List of Symbols for the definition of the symbols.

2.1.2 Standard Well Model

The flow condition for each well in the standard model is expressed with a single set of primary variables (Holmes, 1983). In a three-phase black oil system, four primary variables are introduced: Q_t is the weighted total flow rate, F_w is the weighted fractions of water, F_g is the fluid composition within the wellbore, and p_{bhp} is the bottom-hole pressure, i.e., the pressure in the wellbore at the datum depth. Eq. (2.22) below shows the relation between the primary variable with the flow rates:

$$Q_t = \sum_{\alpha \in \{o,g,w\}} g_\alpha Q_\alpha \quad , \tag{2.22a}$$
$$F_w = \frac{g_w Q_w}{Q_t} \quad , \tag{2.22b}$$
$$F_g = \frac{g_g Q_g}{Q_t} \quad . \tag{2.22c}$$

Q_α is the flow rate of component α under surface conditions and g_α is a weighting factor. A small value of factor is typically chosen for gas, e.g. 0.01, to avoid gas fractions close to unity (Holmes, 1983).

Volumetric inflow rates at reservoir conditions are expressed as:

$$q^r_{\alpha,j} = T_{w,j} M_{\alpha,j} [p_j - (p_{\text{bhp},w} + h_{w,j})] \quad , \tag{2.23}$$

where $q^r_{\alpha,j}$ is the flow rate of phase α through connection j. For other symbols explanation, see List of Symbols.

Conservation equations for each component are introduced to keep the system closed,

$$R_{\alpha,w} = \frac{A_{\alpha,w} - A^0_{\alpha,w}}{\Delta t} + Q_\alpha - \sum_{j \in C(w)} q_{\alpha,j} = 0 \quad , \tag{2.24}$$

and solved in a fully implicit and coupled fashion with the black-oil equations (2.15). $C(w)$ is the set of connections of the well w, $q_{\alpha,j}$ is the flow rate of phase α through connection j under surface condition, $A_{\alpha,w}$, the storage term, is the amount of component α in the wellbore.

Finally, the equations below describe how the wells are controlled. For a well controlled by a predefined BHP target,

$$R_{c,w} = p_{\text{bhp},w} - p^{\text{target}}_{\text{bhp},w} = 0 \quad , \tag{2.25}$$

while for a well controlled by a predefined rate target,

$$R_{c,w} = Q_\alpha - Q^{\text{target}}_\alpha = 0 \quad , \tag{2.26}$$

where Q^{target}_α is the desired surface-volume rate of component α, usually oil rate for a production well.

2.1.3 Solving the System of Equations

The reservoir and the well equations mentioned so far are a large set of fully implicit nonlinear equations that can be rewritten into a compact residual form of $R(y) = 0$, where y is the vector of primary variables. This system is then solved using the Newton-Raphson method. Let $y_n = (p_o, s_w, x)$ be the primary variables after n Newton iterations. With an initial state y_0, the solution of $R(y) = 0$ can be found with iterative solving of

$$J(y_n)(y_{n+1} - y_n) = -R(y_n) \quad , \tag{2.27}$$

until $R(y_n) < \epsilon$, where ϵ is the error tolerance. The central computational function of the simulator is to construct the Jacobian matrix $J(y_n)$ and to solve this linearized problem, and how this part is programmed will dictate its performance. As the number of cells increases in a reservoir model, so does its equations, which will also increase the time needed to solve the equations.

2.1.4 A Paradigm Shift Towards Data-Driven Reservoir Model

History of science and technology can be divided into several eras (Hey, 2009). It all started with experimental science at the early age of science. After that, scientific theories began to emerge, such as Newton's law, Maxwell's law, Kepler's law, etc. Coming into the 21st century, computational science is the norm where fast computers have provided the means for simulation and modeling in areas such as computational fluid dynamics, meteorological and climatological, and hydrocarbon reservoir simulation. Now, humanity has entered the age of *data-intensive science* or *data science*, according to Jim Gray, a renowned computer scientist. In this era, massive amounts of data are collected from physical phenomena and simulations, and new models can then be built based on these data.

Assumption Differences between Numerical Reservoir Simulation and AI-based Reservoir Simulation

The functional relationships used in numerical simulation and modeling are the law of mass conservation, Darcy's Law, and fluid phase behavior. These functional relationships are believed to be true, deterministic, and unchangeable. Thus, if the simulation results do not match our measurements from the field, the conclusion is that the input for reservoir characteristics (the static model) may not be accurately measured and interpreted; hence it must be modified in order to achieve this match. The assumption is that physics holds true for every instance, even though it's extremely hard to model physical phenomena because of its nature of having so many variables and complex relationships between those variables. As discussed in the previous subsection, the equations to model the physics phenomena in the porous media involve numerous variables, and the solutions are strenuous to find. Therefore, engineers often concentrate on altering the reservoir characterization parameters such as permeability, instead of questioning the possibility of inappropriate physics model for that particular reservoir model, in order to achieve a match between simulation results and measurement results. This approach has been the standard for a long time and it often achieves excellent results; therefore, it is not disputed. This approach merely explained in order to emphasize the differences between these two methods.

AI-based reservoir modeling has different assumptions from numerical modeling. Instead of holding the functional relationships constant, AI-based reservoir modeling allows these relationships to change in addition to the possibility of modifying the reservoir characteristics (Mohaghegh, 2011). The functional relationships are generated through the AI-based pattern recognition technology. Moreover, one set of reservoir characteristics can also be modified by another set if it is believed to be better by the geoscientists. Once the geoscientists are confident with the data set, they are not modified during the history matching process. Instead, the functional relationships are modified until a match is attained.

Numerical simulation is built based on the first principle physics of the phenomena that occur in the reservoir. On the other hand, AI-based reservoir modeling builds the model through observation and pattern recognition. Instead of using physics in its first principle and explicit form, physics is used as inspiration for creating a database.

This database is used to train a model that can predict the outcomes of the reservoir simulation by modifying the weights that connecting the parameters in the database. The model will eventually converge to a state where it can mimic the reservoir's behavior as the training process continues. This model built using a generalization from a portion of the database. The generalization is validated and tested using another portion of the database that kept hidden from the model, i.e., the model has not seen the data before. Using this technique, the model is not explicitly formulating physics. Instead, the physics is deduced from the observations in an implicit fashion.

2.2 Artificial Intelligence[2]

There are lots of definitions of artificial intelligence. One of them defined artificial intelligence (AI) as the ability that can be imparted to computers which enables these machines to understand data, learn from the data, and make decisions based on patterns hidden in the data, or inferences that could otherwise be very difficult (to almost impossible) for humans to create manually. AI also enables machines to adjust their "knowledge" based on new inputs that were not part of the data used for training these machines. When talking about artificial intelligence, people sometimes also talking about machine learning. These two words often used interchangeably.

Machine Learning (ML) is the learning in which machines can learn on their own without being explicitly programmed. But, what does learning mean for a machine? (Mitchell, 1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Experience E is often referred to as the dataset fed into the ML. Tasks T are the problems that the machine is expected to solve, e.g., classifying emails as not spam and spam. Performance measure P is the metrics used to quantitatively evaluate the ML performance, e.g., accuracy and error rate. ML is considered a subset of AI. After knowing about ML, the discussion now can continue to deep learning.

Deep learning is a specific kind of machine learning. It is a subset of machine learning and is called deep learning because it makes use of deep neural networks. The machine uses different layers to learn from the data. The depth of the model is represented by the number of layers in the model. Deep learning is the new state of the art in terms of AI. In deep learning, the learning phase is done through an artificial neural network (ANN), which will be discussed in the next section. In short, ANN is an architecture where the layers are stacked on top of each other. To summarize, Fig. 2.1 shows the Venn diagram of artificial intelligence, machine learning, and deep learning.

Based on the learning methods, machine learning algorithms can be classified into supervised learning and unsupervised learning.

Supervised Learning

Supervised learning as the name indicates the presence of a supervisor as a teacher. Basically, supervised learning is a learning in which human teach or train the machine using data which is well labeled, that means some data is already tagged with the correct answer. After that, the machine is provided with a new set of examples (data). The supervised learning algorithm analyzes the training data (set of training examples) and produces a correct outcome from labeled data. Supervised learning classified into two categories of algorithms:

[2]This section has been presented in the author's specialization project.

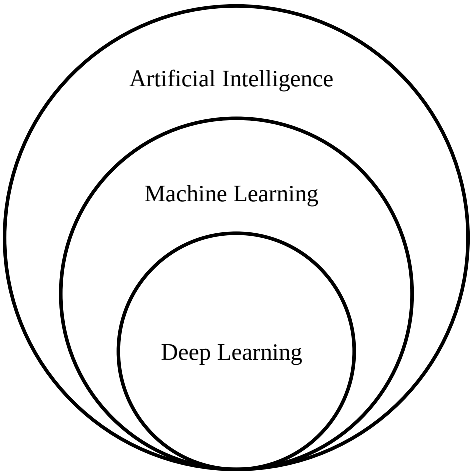


Figure 2.1: The Venn diagram of artificial intelligence, machine learning, and deep learning.

- **Classification or Logistic Regression**: A classification problem is when the output variable is a category, such as ”red” or ”blue” or ”disease” and ”no disease.”
- **Linear Regression**: A regression problem is when the output variable is a real value, such as ”dollars” or ”weight.”

Unsupervised Learning

Unsupervised learning is the training of machine using information that is neither classified nor labeled and allowing the algorithm to act on that information without guidance. Here the task of machine is to group unsorted information according to similarities, patterns, and differences without any prior training of data.

Unlike supervised learning, no teacher is provided, which means no training will be given to the machine. Therefore the machine is restricted to find the hidden structure in unlabeled data by itself. Unsupervised learning classified into two categories of algorithms:

- **Clustering**: A clustering problem is where someone wants to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
- **Association**: An association rule learning problem is where someone wants to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

2.2.1 Artificial Neural Networks

Artificial Neural Networks or ANN is an information processing paradigm that is inspired by the way the biological nervous system such as brain process information. It is composed of many highly interconnected processing elements (neurons) working in unison to solve a specific problem. Biological neurons (Fig. 2.2) or simply neurons are the fundamental units of the brain and nervous system. The cells are responsible for receiving sensory input from the external world via dendrites, process it, and gives the output through Axons.

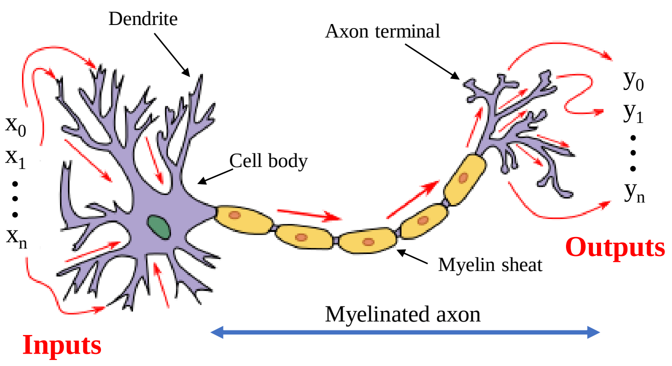


Figure 2.2: A biological neuron. Modified from (Chauhan, 2019).

Cell body (Soma): The body of the neuron cell contains the nucleus and carries out biochemical transformation necessary to the life of neurons.

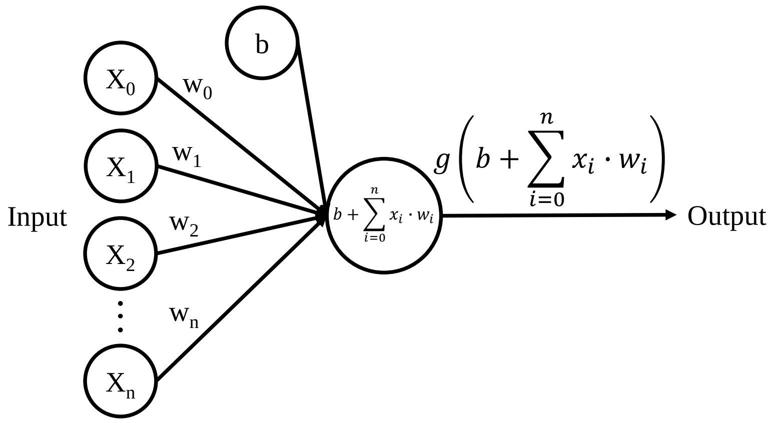
Dendrites: Each neuron has fine, hair-like tubular structures (extensions) around it. They branch out into a tree around the cell body. They accept incoming signals.

Axon: It is a long, thin, tubular structure that works like a transmission line.

Synapse: Neurons are connected in a complex spatial arrangement. When axon reaches its final destination, it branches again called terminal arborization. At the end of the axon are highly complex and specialized structures called synapses. The connection between two neurons takes place at these synapses.

Dendrites receive input through the synapses of other neurons. The soma processes these incoming signals over time and converts that processed value into an output sent out to other neurons through the axon and the synapses. A single neuron is also often called a **node**. The following Fig. 2.3 represents the general model of ANN, which is inspired by a biological neuron. It is also called **Perceptron** (Rosenblatt, 1958). Perceptron is a single layer of neural networks. It gives a single output.

In Fig. 2.3, for one single observation, $x_0, x_1, x_2, \ldots, x_n$ represents various inputs (independent variables) to the network. Each of these inputs is multiplied by a connection weight or synapse. The weights are represented as $w_0, w_1, w_2, \ldots, w_n$. Weight shows the strength of a particular node. These weights are need to be initialized. The common practice is to randomly initialize the weights to prevent each neuron doing the same calculation. Then, b is a bias value. A bias value allows the activation function to shift up or down, just like a constant c does in $y = mx + c$. In the simplest case, these products are summed, fed to a transfer function (activation function) to generate a result, and this result



b
X_0
w_0
X_1
w_1
Input
X_2
w_2
w_n
X_n
$b + \sum_{i=0}^{n} x_i \cdot w_i$
$g\left(b + \sum_{i=0}^{n} x_i \cdot w_i\right)$
Output

Figure 2.3: A perceptron.

is sent as output. This can be expressed mathematically as,

$$b + x_0 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n = b + \sum_{i=0}^{n} x_i \cdot w_i \quad . \tag{2.28}$$

After that, an activation function is applied, expressed as $g\left(b + \sum_{i=0}^{n} x_i \cdot w_i\right)$, where $g(\bullet)$ is the activation function.

Activation Function

The Activation function is vital for an ANN to learn and make sense of something complicated. Their main purpose is to convert an input signal of a node in an ANN to an output signal. This output signal is used as input to the next layer in the stack.

Activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The motive is to introduce non-linearity into the output of a neuron.

If the activation function is not applied, then the output signal would be a simple linear function (one-degree polynomial). A linear function is easy to solve, but it is limited in its complexity and has less power. Without the activation function, the model cannot learn and model complex data such as images, videos, audio, speech, etc. Now, the question arises, why is non-linearity needed?

Non-linear functions are those which have a degree more than one and they have a curvature. A neural network needs to learn and represent almost anything and any arbitrary complex function that maps an input to output. Neural Network is considered “Universal Function Approximators”. It means they can learn and compute any function at all. There are several types of activation functions:

1. **Linear Activation Function**
   A linear function takes the form:

$$f(x) = Cx \quad . \tag{2.29}$$

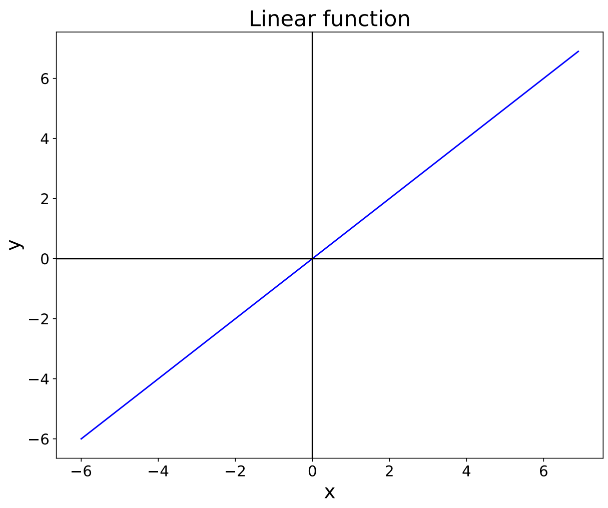


Figure 2.4: A linear function.

   The linear function, Eq. (2.29), takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input. This activation function is only used in the output layer of regression problem, because it returns the same value as the previous layer, which is the prediction value. Fig. 2.4 shows the plot of linear function.

2. **Sigmoid Activation Function — (Logistic function)**
   A sigmoid function (Fig. 2.5) is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve which ranges between 0 and 1, therefore it is used for models where someone need to predict the probability as an output. The sigmoid function, Eq. (2.30), is differentiable, means one can find the slope of the curve at any two points.

$$f(x) = \frac{1}{1 + e^{-x}} \quad . \tag{2.30}$$

   The drawback of the sigmoid activation function is that it can cause the neural network to get stuck at training time if strong negative input is provided.

3. **Hyperbolic Tangent Function — (tanh)**
   Hyperbolic tangent function, Eq. (2.31), is similar to sigmoid but better in performance. It is nonlinear in nature, so it's great to use in stacked layers. It can be seen from Fig. 2.6 the function ranges between (-1,1).

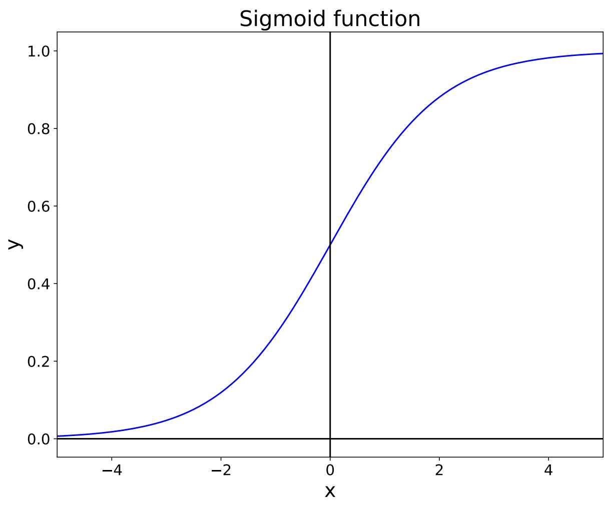


Figure 2.5: A sigmoid function.

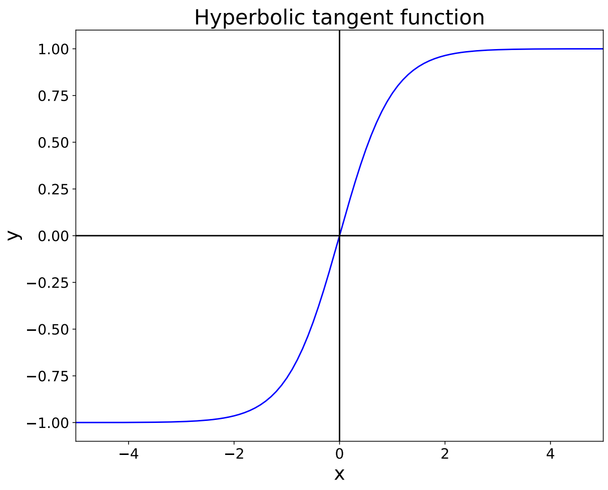


Figure 2.6: A hyperbolic tangent function.

$$f(x) = \tanh x \quad . \tag{2.31}$$

The main advantage of this function is that strong negative inputs will be mapped to negative output and only zero-valued inputs are mapped to near-zero outputs. So, it's less likely to get stuck during training.

4. **Rectified Linear Units — (ReLU)**
ReLU, Eq. (2.32), is the most used activation function in ANN which ranges $[0, \infty)$. Fig. 2.7 below shows the plot of ReLU function.

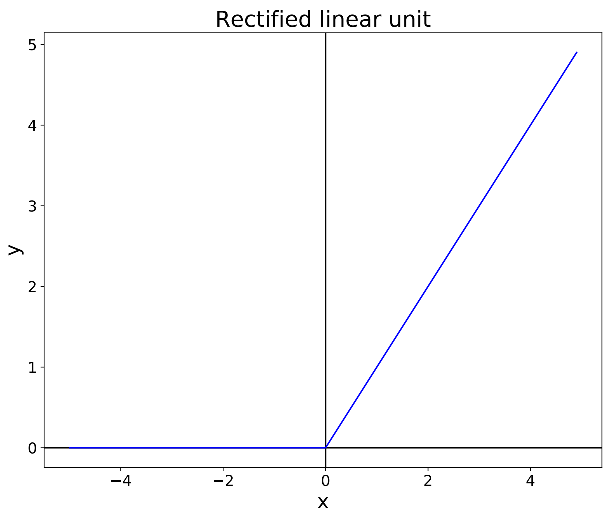


Figure 2.7: A ReLU function.

$$f(x) = \begin{cases} 0 \text{ for } x < 0 & , \\ x \text{ for } x \geq 0 & . \end{cases} \tag{2.32}$$

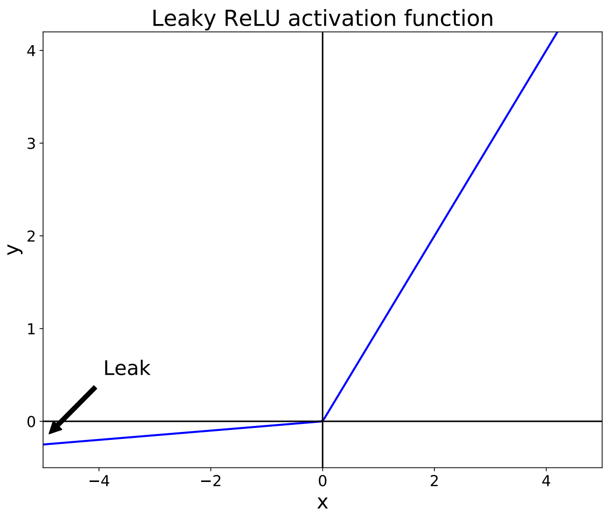


Figure 2.8: A Leaky ReLU function.

$$f(x) = \begin{cases} 0.01 \text{ for } x < 0 & , \\ x \text{ for } x \geq 0 & . \end{cases} \tag{2.33}$$

It gives an output 'x' if x is positive and 0 otherwise. ReLU is non-linear in nature and a combination of ReLU is also non-linear. In fact, it is a good approximator and any function can be approximated with a combination of ReLU. However, it should only be applied to hidden layers of a neural network (Goodfellow et al., 2016).

One problem with ReLU is some gradients are fragile during training and can die. It causes a weight update, which will make it never activate on any data point again. Basically, ReLU could result in dead neurons.

To fix the problem of dying neurons, **Leaky ReLU** was introduced (Fig. 2.8). So, Leaky ReLU introduces a small slope to keep the updates alive. Leaky ReLU, Eq. (2.33), ranges $(-\infty, \infty)$. Leak helps to increase the range of the ReLu function.

How do Neural Networks work?

To better understand how neural networks work, consider an example of the price of a property. In this example, four different factors affect the price of a property, which are area, bedrooms, distance to the city, and age. Fig. 2.9 illustrates this problem in a simple neural network.

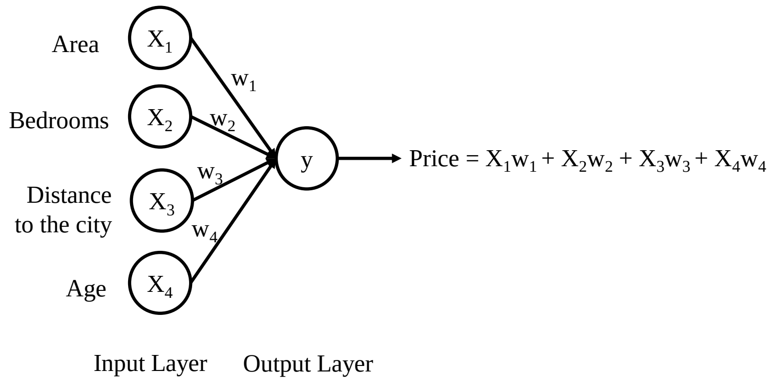


Figure 2.9: A simple neural network.

The input values go through the weighted synapses straight over to the output layer. All four will be analyzed, an activation function will be applied, and the results will be produced.

This is simple enough, but there is a way to amplify the Neural Network's power and increase its accuracy by the addition of a **hidden layer** that sits between the input and output layers. Illustration provided in Fig. 2.10.

In Fig. 2.10, all four variables are connected to neurons via a synapse. However, not all of the synapses are weighted. They will either have a zero value or non-zero value. Here, the non-zero weight value means that the input neuron is important to consider. Otherwise, the zero weight value is not considered. Fig. 2.10 illustrates only the non-zero weight

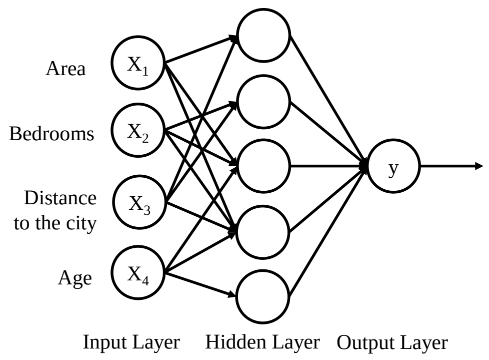


Figure 2.10: A neural network with a hidden layer.

values, and the zero weight values are not. For example, ”Area” and ”Distance to the city” are non-zero for the first neuron in the hidden layer, which means they are weighted and matter to the first neuron. The other two variables, ”Bedrooms” and ”Age” aren’t weighted and so are not considered by the first neuron.

One may wonder why that first neuron is only considering two of the four variables. In this case, it is common on the property market that larger homes become cheaper; the further they are from the city. That’s a basic fact. So, what this neuron may be doing is looking specifically for properties that are large but are not so far from the city.

Now, this is where the power of neural networks comes from. There are many of these neurons, each doing similar calculations with different combinations of these variables. Once this criterion has been met, the neuron applies the activation function and do its calculations. The next neuron down may have weighted synapses of ”Distance to the city” and ”Bedrooms.” This way, the neurons work and interact flexibly, allowing it to look for specific things and therefore do a comprehensive search for whatever it is trained for. This procedure is also known as **forward propagation**.

How Neural Networks learn?

Looking at an analogy may be useful in understanding the mechanisms of a neural network. Learning in a neural network is closely related to how people learn in our regular lives and activities — people perform an action and are either accepted or corrected by a trainer or coach to understand how to get better at a certain task. Similarly, neural networks require a trainer in order to describe what should have been produced as a response to the input. Based on the difference between the actual value and the predicted value, an error value, also called **cost function** is computed and sent back through the system. There are several ways to find a cost function; one of them is the mean squared error (MSE). **MSE** simply squares the difference between every network output and its true label, then takes the average. Mathematically, this cost function can be expressed as:

$$J(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \quad , \tag{2.34}$$

where J is the cost function (also known as the **loss function**), N is the number of trained data, **y** is a vector of true labels, and $\hat{\mathbf{y}}$ is a vector of network predictions.

For each layer of the network, the cost function is analyzed and used to adjust the threshold and weights for the next input. The goal is to minimize the cost function. Lower cost function means the predicted value is closer to the actual value. In this way, the error keeps becoming marginally lesser in each run as the network learns how to analyze values.

The resulting data are fed back through the entire neural network. The weighted synapses connecting input variables to the neuron are the only thing that can be controlled. As long as there exists a disparity between the actual value and the predicted value, those weights have to be adjusted. Once the weights are tweaked, and the neural network is trained again, a new cost function will be produced, hopefully, smaller than the last. This process is repeated until the cost function down to as small as possible.

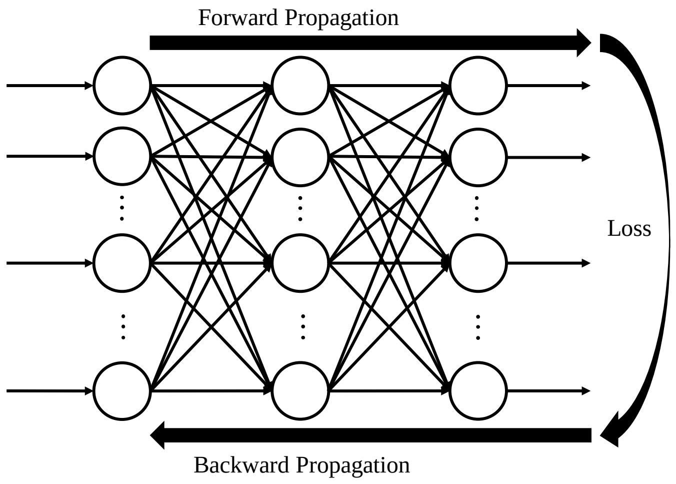


Figure 2.11: ”Learning” process in neural networks.

The procedure described above is known as **backpropagation** (Fig. 2.11) and is applied continuously through a network until the error value is kept at a minimum. Backpropagation takes the error associated with a wrong guess by a neural network and uses it to adjust the neural network's parameters in the direction of less error. But now, a question may arise, how does it know the direction of less error? The problem is usually tackled with optimization methods to find minima. One method frequently used is gradient descent.

Gradient Descent

Generally, to find the minimum of a function, the derivative is set to zero and solve for the parameters. However, it is almost impossible to obtain a closed-form solution for a cost function because it involves so many parameters. Instead, the minimum is iteratively searched using a method called gradient descent.

As a visual analogy, imagine a man standing on a mountain and trying to find the way down. At every step, he will walk into the steepest direction, since this direction is the most promising to lead him towards the bottom.

Gradient descent operates similarly when trying to find the minimum of a function: It starts at a random location in parameter space and then iteratively reduces the error J until it reaches a local minimum. At each step of the iteration, it determines the direction of steepest descent and takes a step along that direction. This process is depicted in the one-dimensional case in Fig. 2.12.

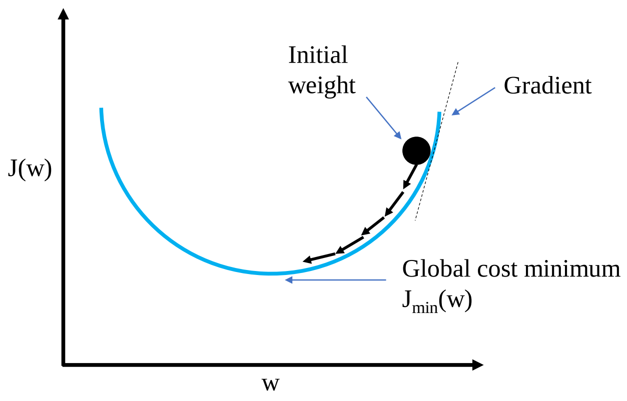


Figure 2.12: Gradient descent illustration.

As seen in Fig. 2.12, there exists a value of parameters w, which has a minimum value of J_{min}. The gradient of the cost function is calculated as a partial derivative of cost function J with respect to each model parameter w_i. One should also define the learning rate, α, which determines how quickly it moves toward the minimum. If α is too large, it can overshoot and not moving toward the minimum. If α is too small, i.e., small steps of learning, the overall time is taken by the model to find the minimum can be too long.

There are three ways of using gradient descent; they are batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.

In **batch gradient descent**, all of the training instances are used to update the model parameters in each iteration. In this way, a vast number of incorrect weights are eliminated. For example, if there are 3 million samples, the gradient descent has to loop through 3 million times. So basically, 3 million calculation per iteration has to be made. The calculation could take high computational power and long running time.

Gradient descent works fine when there is a convex curve just like in Fig. 2.12. But, if the curve has local minimum and global minimum, the gradient descent can be trapped in the local minimum (Fig. 2.13).

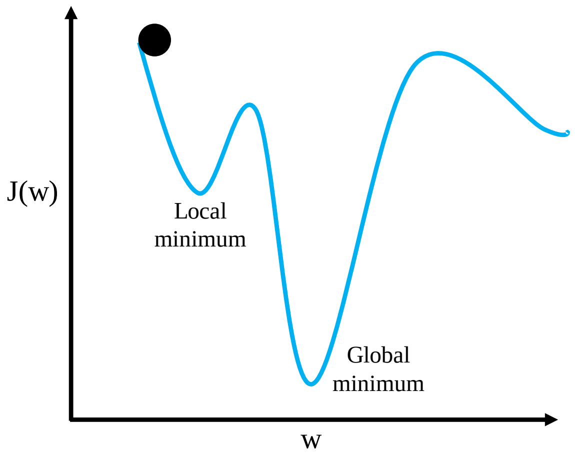


Figure 2.13: Local minimum problem.

The word ‘*stochastic*‘ means a system or a process that is linked with a random probability. Hence, in **stochastic gradient descent (SGD)**, a row of data is selected randomly instead of the whole data set for each iteration.

SGD helps avoid the problem of the local minimum. It is much faster than batch gradient descent because it is running each row of data at a time, and it doesn’t have to load the whole data in memory for doing computation. One thing to be noted is that, as SGD is generally noisier than typical gradient descent, it usually took a higher number of iterations to reach the minimum, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minimum than typical gradient descent, it is still computationally much less expensive than typical gradient descent.

Another way is with the **mini-batch gradient descent**. Instead of using all examples, mini-batch gradient descent divides the training set into a smaller size called batch denoted by m. Thus a mini-batch m is used to update the model parameters in each iteration. Mini-batch gradient descent is typically the algorithm of choice when training a neural network, and the term SGD usually is also employed when mini-batches are used. SGD is technically a mini-batch gradient descent with a mini-batch size m of 1.

The gradient descent can be further optimized with several algorithms to speed up its running time and increase its accuracy.

Gradient Descent Optimization Algorithms

In this section, there will be a discussion about some algorithms that are widely used by the deep learning community.

1. **Momentum**
   SGD has trouble navigating ravines, i.e., areas where the surface curves much more steeply in one dimension than in another (Sutton, 1986), which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum, as in Fig. 2.14.

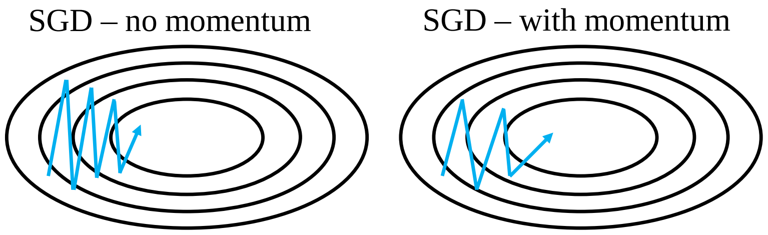
   

   Figure 2.14: The difference of SGD with and without momentum. Modified from (Ruder, 2016).

   Momentum (Qian, 1999) is a method that helps accelerate SGD in the relevant direction and dampens oscillations, as can be seen in Fig. 2.14. Momentum, γ, is an added term in the objective function, which is a value between 0 and 1 that increases the size of the steps taken towards the minimum by trying to jump from a local minimum.

   Essentially, when using momentum, a ball is pushed down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. ($\gamma < 1$). The same thing happens to the parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. The results are faster convergence and reduced oscillation.

2. **RMSprop**
   RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class. Retrieved from (Ng, 2013b).

   RMSprop is one of the algorithms that adaptively adjusts the learning rate. It adapts the learning rate to the parameters, performing smaller updates (i.e., low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e., high learning rates) for parameters related to infrequent features. For this reason, it is well-suited for dealing with sparse data. RMSprop divides the learning rate by an exponentially decaying average of squared gradients.

3. **Adam**
   Adaptive Moment Estimation (Adam) (Kingma and Ba, 2014) is another method

that combines RMSprop and momentum. It computes adaptive learning rates for each parameter and keeps an exponentially decaying average of past gradients, similar to momentum. While momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which prefers flat minima in the error surface (Heusel et al., 2017).

Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast. The learning speed of the model is quite fast and efficient. It also rectifies every problem faced in other optimization techniques such as vanishing learning rate, slow convergence, or high variance in the parameter updates, leading to the fluctuating loss function. Adam is the most popular optimizer used for neural networks nowadays.

2.2.2 Practical Aspects of Neural Networks

In this section, the practical aspects of building neural networks architecture and common problems faced in the process are discussed.

Data Splitting

Before feeding the data into neural networks, the dataset is usually split into three categories, training dataset, validation dataset, and test dataset.

- **Training Dataset**: The actual dataset that is used to train the model. The model *sees* and *learns* from this data.
- **Validation Dataset**: The validation set is used to evaluate a given model regularly. The purpose of the validation dataset is to fine-tune the model hyperparameters. Hyperparameters are parameters whose values are set before the learning process begins, e.g., the learning rate α. Machine learning engineers evaluate the validation set results and then update the hyperparameters. Hence, the model occasionally *sees* this data, but never does it *learn* from this. So, the validation set in a way affects a model, but indirectly.
- **Test Dataset**: The test dataset provides the gold standard used to evaluate the model. It is only used once a model is completely trained (using the train and validation sets). The model never *sees* and *learns* from this data.

Data Scaling

Data scaling is a technique to standardize the independent features present in the data in a fixed range. It is performed during the data pre-processing to handle highly varying magnitudes or values or units. If data scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values.

There are many methods to scale data, but these two methods are the most important:

- **Standardization**: This technique removes the mean and scales the data to unit variance. It will scale the data such that the distribution is now centered around 0, with a standard deviation of 1. The formula of Standardization is expressed in Eq. (2.35).

$$x_{\text{new}} = \frac{x_i - \mu}{\sigma} \quad , \tag{2.35}$$

  where μ is the mean (average) and σ is the standard deviation from the mean.

- **Min-Max Normalization**: This technique re-scales an observation value with distribution value between 0 and 1 (or -1 to 1 if there are negative values). The formula of Min-Max Normalization is expressed in Eq. (2.36).

$$x_{\text{new}} = \frac{x_i - \min(x)}{\max(x) - \min(x)} \quad . \tag{2.36}$$

Underfitting and Overfitting

Underfitting, or high bias, is when the model maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, overfitting, or high variance, is caused by a model that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

Consider the problem of predicting y from $x \in R$. The leftmost figure in Fig. 2.15 shows the result of fitting a line to a dataset. Since the data doesn't lie in a straight line, the fit is not very good.

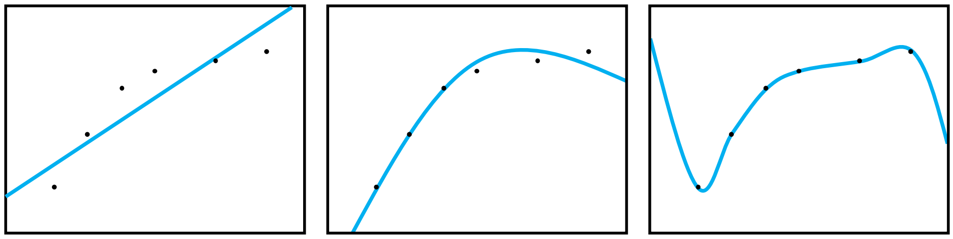


Figure 2.15: Underfitting, appropriate fitting, and overfitting. Modified from (Ng, 2013a).

Instead, if an extra feature x^2 is added, then a slightly better fit to the data is obtained (middle figure in Fig. 2.15). Naively, it might seem that the more features added, the better. However, there is also a danger in adding too many features: The rightmost figure in Fig. 2.15 is the result of fitting a 5^{th} order polynomial (x^5). It can be seen that even though the fitted curve passes through the data perfectly, it will not generalize well for new data. To conclude, the figure on the left in Fig. 2.15 shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right in Fig. 2.15 is an example of **overfitting**.

This terminology is applied to both linear and logistic regression. There are two options to address the issue of underfitting:

1. **Increase the number of features**: Find other data that might help the model to predict better results.
2. **Add polynomial features**: Adding higher order polynomial can address the underfitting problem.

There are four options to address the overfitting issue:

1. **Reduce the number of features**: Manually select which features to keep. Remember that the discarded features may have some important information, so the choice has to be made carefully.
2. **Regularization**: Keep all the features, but reduce the magnitude of weights w. Regularization works well when there exists a lot of slightly useful features. This is a form of regression that constrains/regularizes or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model to avoid overfitting.
3. **Early stopping**: When training a learning algorithm using gradient descent, how well the model performs on each iteration can be measured. Up to a certain number of iterations, each iteration improves the model. After that point, however, the model's ability to generalize can weaken as it begins to over-fit the training data. Fig. 2.16 below illustrates how early stopping works.

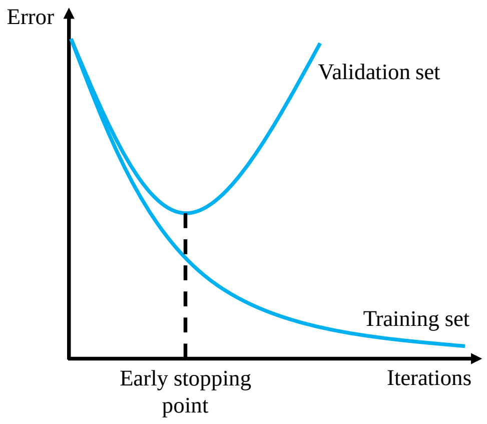


Figure 2.16: Early stopping.

4. **Dropout**: Dropout is a technique that randomly drops the nodes (along with their connections) from the neural network during training (Srivastava et al., 2014). Because the outputs of a layer under dropout are randomly subsampled, it reduces the capacity or thinning the network during training. As such, a wider network, e.g., more nodes, may be required when using dropout.

Hyperparameter Optimization

In machine learning, hyperparameter optimization or tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm. Recall that a hyperparameter is a parameter whose value is set before the learning process begins, e.g., the learning rate α, and cannot be learned by the training process. By contrast, the values of other parameters (typically node weights) are learned.

Hyperparameter optimization finds a tuple of hyperparameters that yields an optimal model that minimizes a predefined loss function on given independent data (Claesen and Moor, 2015). There are several approaches to optimize the hyperparameter:

1. **Grid Search**
   The traditional way of performing hyperparameter optimization has been grid search, or a parameter sweep, which is simply an exhaustive searching through a manually specified subset of the hyperparameters space of a learning algorithm. Every possible combination of hyperparameters in the specified subset of the hyperparameters is then tested, and the loss is recorded. The combination of hyperparameters that gives the minimum loss function is then picked.

2. **Random Search**
   Random Search replaces the exhaustive enumeration of all combinations by selecting them randomly. This can be simply applied to the discrete setting described above, but also generalizes to continuous and mixed spaces. It can outperform Grid search, especially when only a small number of hyperparameters affect the final performance of the machine learning algorithm (Bergstra and Bengio, 2012).

   However, these two methods are relatively inefficient because they do not choose the next hyperparameters to evaluate based on previous results. Grid and random searches are utterly uninformed by past evaluations, and as a result, often spend a significant amount of time evaluating ”bad” hyperparameters.

3. **Bayesian Optimization**
   Bayesian optimization is a global optimization method for noisy black-box functions (Mockus, 2009). Applied to hyperparameter optimization, Bayesian optimization builds a probabilistic model of the function mapping from hyperparameter values to the objective evaluated on a validation set.

   Bayesian approaches, in contrast to random or grid search, keep track of past evaluation results which they use to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function:

   $$P(\text{score} \mid \text{hyperparameters}) \quad . \tag{2.37}$$

   This model is called a ”surrogate” for the objective function and is represented as $p(y \mid x)$ (Shahriari et al., 2016). The surrogate is much easier to optimize than the objective function. Bayesian methods find the next set of hyperparameters to evaluate the actual objective function by selecting hyperparameters that perform best on the surrogate function. In other words:

(a) Build a surrogate probability model of the objective function.
(b) Find the hyperparameters that perform best on the surrogate.
(c) Apply these hyperparameters to the true objective function.
(d) Update the surrogate model incorporating the new results.
(e) Repeat steps (b)–(d) until max iterations or time is reached.

At a high-level, Bayesian optimization methods are efficient because they choose the next hyperparameters in an informed manner. The basic idea is to spend a little more time selecting the next hyperparameters to make fewer calls to the objective function. In practice, the time spent selecting the next hyperparameters is inconsequential compared to the time spent in the objective function. By evaluating hyperparameters that appear more promising from past results, Bayesian methods can find better model configurations than random search in fewer iterations.

A good visual description of what is occurring in Bayesian optimization is shown in the figures below. Fig. 2.17 shows an initial estimate of the surrogate model — in black with associated uncertainty in gray — after two evaluations. Clearly, the surrogate model is a poor approximation of the actual objective function in red.

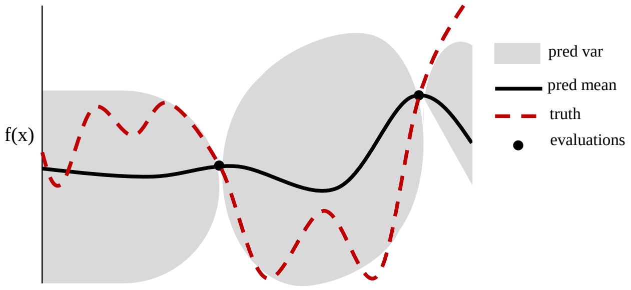


Figure 2.17: Surrogate function after 2 evaluations. Modified from (Snoek et al., 2012).

Fig. 2.18 shows the surrogate function after 8 evaluations. Now the surrogate almost exactly matches the true function. Therefore, if the algorithm selects the hyperparameters that maximize the surrogate, they will likely yield very good results on the true evaluation function.

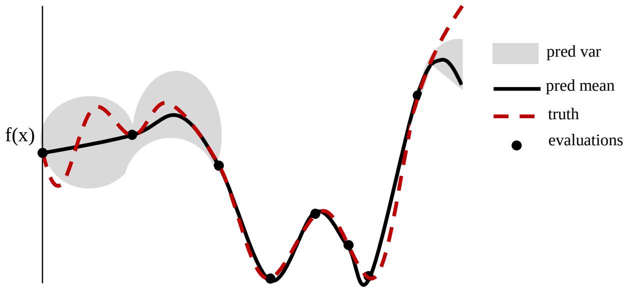


Figure 2.18: Surrogate function after 8 evaluations. Modified from (Snoek et al., 2012).

There are five aspects of Bayesian hyperparameter optimization:

(a) A domain of hyperparameters for the search space.

(b) An objective function that inputs the hyperparameters and outputs a score that is to be minimized or maximized.

(c) The surrogate function.

(d) A selection function for evaluating the next hyperparameters from the surrogate model.

(e) A history of score and hyperparameters pairs to update the surrogate model.

A **domain** consists of probability distributions. The probability distributions place greater probability in regions where the expected true best hyperparameters lie. These are informed by prior practice/knowledge, e.g., the learning rate domain is usually a log-normal distribution over several orders of magnitude.

The **objective function** that hyperparameters usually want to minimize is the mean squared error, like Eq. (2.34). It is expensive to compute this objective function, even though it looks simple. If it can be quickly calculated, every possible combination of hyperparameters could be calculated (like in grid search). However, it may take hours or even days to evaluate the objective function. Therefore, a surrogate model of the objective function is needed.

The **surrogate function**, also called the response surface, is the probability representation of the objective function built using previous evaluations. Sometimes, it is called a response surface because it is a high-dimensional mapping of hyperparameters to the probability of a score on the objective function. One form of the surrogate function is the Tree-structured Parzen Estimator or TPE (Bergstra et al., 2011). The selection function has to be discussed first to construct the TPE.

The **selection function** is the criteria to select the next hyperparameters from the surrogate function. The most common selection function is Expected Improvement,

$$\mathrm{EI}_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy \quad , \tag{2.38}$$

where y^* is a threshold value of the objective function, x is proposed set of hyperparameters, y is the objective function value using hyperparameters x, and $p(y|x)$ is the surrogate probability model expressing the probability of y given x. The aim is to maximize the Expected Improvement with respect to x.

The Tree-structured Parzen Estimator builds a model by applying Bayes rule of

$$p(y|x) = \frac{p(x|y) * p(y)}{p(x)} \quad , \tag{2.39}$$

where $p(x|y)$ is the probability of the hyperparameters given the score on the objective function, also expressed as:

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \quad , \\ g(x) & \text{if } y \geq y^* \quad , \end{cases} \tag{2.40}$$

where $y < y^*$ represents a lower value of the objective function than the threshold. The explanation of this equation is that two different distributions for the hyperparameters are made: one where the value of the objective function is less than the threshold, $l(x)$, and one where the value of the objective function is greater than the threshold, $g(x)$.

Draw values of x from $l(x)$ are better because this distribution is based only on x that produced lower scores than the threshold. It is also corroborated with Bayes rule, as the Expected Improvement equation becomes

$$\mathrm{EI}_{y^*}(x) = \frac{\gamma y^* l(x) - l(x) \int_{-\infty}^{y^*} p(y)dy}{\gamma l(x) + (1-\gamma) g(x)} \propto \left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1} \quad , \tag{2.41}$$

which says that the Expected Improvement is proportional to the ratio $\frac{l(x)}{g(x)}$. Therefore, drawing values of x from $l(x)$ will maximize this ratio, consequently maximize the Expected Improvement.

Every time the algorithm proposes a new set of hyperparameters, it evaluates them with the actual objective function and records the result in a pair of scores and hyperparameters. A **history** consists of these records. The algorithm makes $l(x)$ and $g(x)$ using the history to develop a probability model of the objective function that improves every iteration.

In summary, an initial estimate is made for the surrogate function and updated as more evidence is gathered. Eventually, enough evaluations of the surrogate function will accurately reflect the objective function. The hyperparameters that maximize the Expected Improvement would also maximize the objective function.

2.3 Genetic Algorithm

A genetic algorithm (GA) is a type of population-based stochastic searching approach that mimics the natural selection and survival of the fittest in the biological world (Holland, 1975). This algorithm first introduced by John Holland in 1960 based on the concept of Darwin's theory of evolution. The genetic algorithm does not use derivative information to find the optimal solutions, and there it falls in the category of derivative-free optimization. This algorithm is better suited if the objective function f might be non-smooth, or time-consuming to evaluate, or in some way noisy, so that methods that rely on derivatives or approximate them via finite differences are of little use.

The evolution usually starts from a population of randomly generated individuals. They then produce offspring which inherit the characteristics of the parents and will be added to the next generation. The better the fitness of the parents, the better fitness of their offspring, the better chance of surviving. This process keeps repeating until the fittest individual is found. This notion is then applied for an optimization or search problem.

The genetic algorithm generally operates in a series of framework as shown in Fig. 2.19.

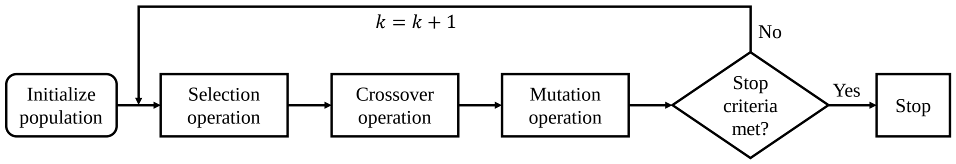


Figure 2.19: The flowchart of the genetic algorithm optimization scheme. Modified from (Chuang et al., 2015).

Initial Population

The process begins with a set of individuals, which is called a **population**. Each individual is the proposed solution to the defined problem. Every individual is a set of unique parameters or variables known as **gene**. These genes are then joined into a string to form an individual, or also called **chromosome**. Fig. 2.20 illustrates the terms for better understanding.

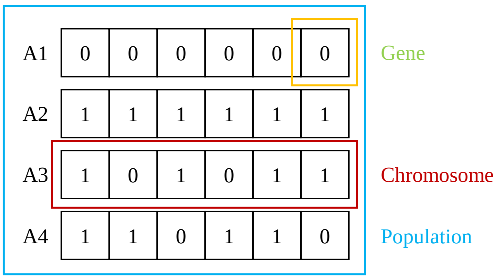


Figure 2.20: Population, gene, and chromosome in genetic algorithm.

The size of the population varies for every problem but usually includes several hundred or thousands of possible solutions. The initial population is often generated randomly, allowing for a wide range of possible solutions (the search space). Occasionally, the solutions can also be initialized in areas where optimal solutions are likely to be found.

Selection Operation

Some parts of the existing population are selected to breed a new generation. The selection is based on the fitness of the individual. A fitness function will determine the fitness of every individual, and the fitter individuals are more likely to be selected. In an optimization problem, the objective function is the fitness function. For example, if the optimization problem is to maximize NPV, then the objective function is NPV. The individual (solution) that yields higher NPV has higher fitness. After knowing the fitness of every individual, a particular selection method is then used. Two of the selection methods are roulette wheel selection and elitism selection.

In the roulette wheel selection, better fitness is more likely to be selected for the breeding of the next generation. Imagine this process as spinning a roulette with the same amount of pockets as the number of individuals in the existing population, with fitness determining its pocket size. The probability of choosing individual i, p_i, is equal to the fitness of i, f_i, divided by the total fitness of that population, mathematically expressed as

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \quad , \tag{2.42}$$

where N is the size of the current population.

In the elitism selection, a small portion of the best individuals from the current population is carried over to the next generation without any changes. This often performed to keep the best individuals alive over the generations instead of breeding them.

Crossover Operation

Crossover (also called recombination or breeding) is the next step to generate the next population from the selected individuals. A pair of ”parent” individuals are chosen from the pool of chosen individuals previously to produce a new individual. A crossover operation is performed afterward. This operation can be performed for bit arrays or vector of real numbers. For a bit array, single-point or k-point crossover can be implemented.

In a single-point crossover, a crossover point is chosen at a random point on both parents’ chromosomes. New offspring are created by exchanging the genes of parents among themselves until the crossover point is reached. This strategy can be generalized to k-point crossover for any positive integer k by picking k crossover points. This process is illustrated in Fig. 2.21.

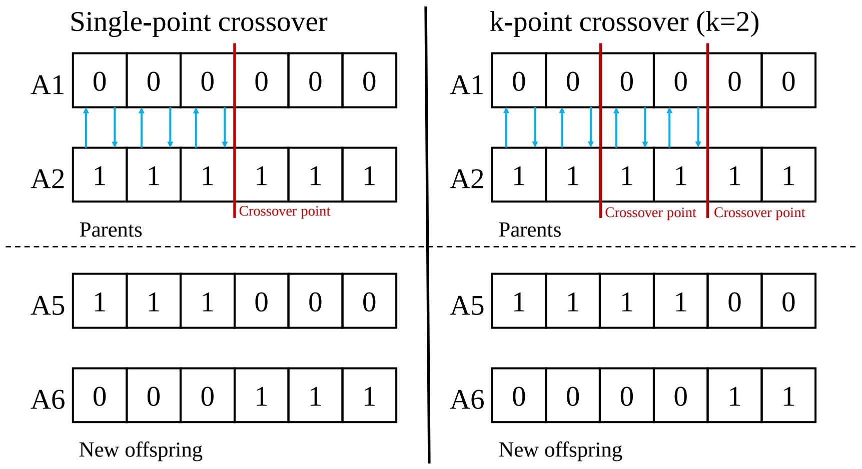


Figure 2.21: Single-point crossover and k-point crossover.

There are also several crossover techniques for a vector of real numbers. Instead of swapping the genes, the geometric mean, or the arithmetic mean is calculated from both parent’s chromosomes. The resulting offspring is now made of the chosen mean. Illustration of these techniques can be seen in Fig. 2.22. Other popular crossover techniques are Blend crossover, or BLX-α (Takahashi and Kita, 2001), and Simulated Binary Crossover, or SBX (Deb and Beyer, 2001).

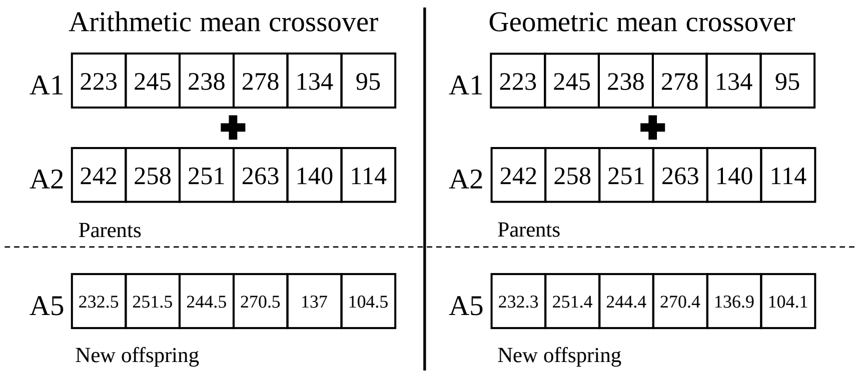


Figure 2.22: Arithmetic-mean and geometric-mean crossover.

Mutation Operation

The mutation alters one or more genes in a chromosome from its initial value. A mutation is added to maintain diversity within the population, prevent premature convergence, and anticipate being trapped in the local optima, if it ever trapped. The chromosome may also be entirely different than the previous chromosome; hence GA may come to a better solution by mutating. This mutation happens according to a predefined probability. However, this probability should be set low; otherwise, it will turn into a traditional random search.

For a bit array, mutation can flip the bit from 1 to 0 and vice versa. User can define how many genes are going to be flipped and how likely will it happen. An illustration of this process can be seen in Fig. 2.23.

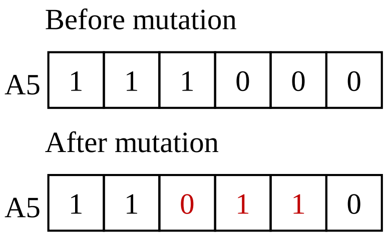


Figure 2.23: Before and after mutation for binary data.

For a vector of real numbers, there are also plenty of options. It depends on the search problem. For instance, one could sum or subtract a random number between a fixed interval for some of the genes, as can be seen in Fig. 2.24. For larger search spaces, one could also choose a larger interval and diminish it from generation to generation.

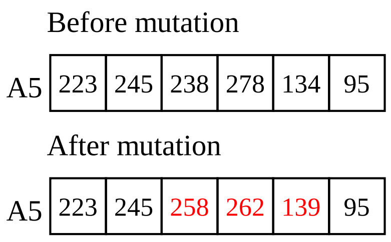


Figure 2.24: Before and after mutation for real numbers.

Termination Operation

This generational process will be repeated until the termination condition has been reached. These are, but not limited to, the termination condition of the genetic algorithm:

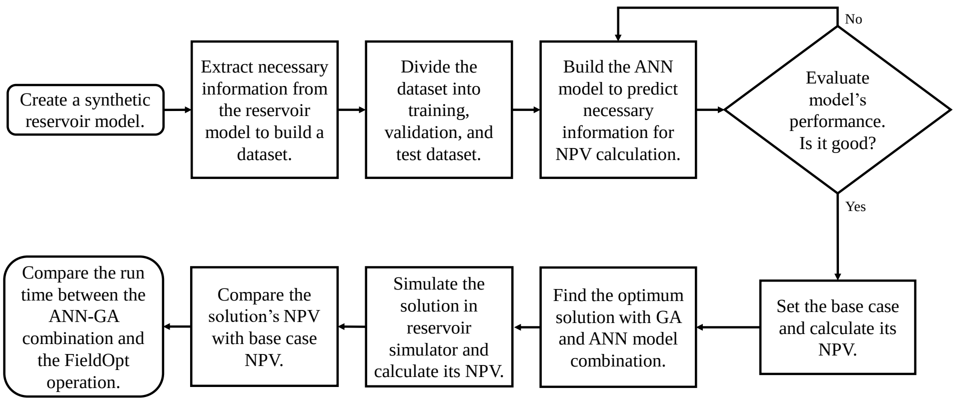
- a solution is found within the minimum criteria;
- allocated resources (money/computational time) reached;
- maximum number of generations reached;
- the best solution (the highest fitness) of every generation approaching or reaching a plateau such that successive iteration only produce diminishing results or no longer produce better results;
- manual termination;
- combinations of the above.

Chapter 3

Methodology

This chapter describes the workflow of this study and discusses the technical details involved in building the ANN model and genetic algorithm optimization.

3.1 Workflow



Create a synthetic reservoir model.
Extract necessary information from the reservoir model to build a dataset.
Divide the dataset into training, validation, and test dataset.
Build the ANN model to predict necessary information for NPV calculation.
Evaluate model's performance. Is it good?
No
Yes
Set the base case and calculate its NPV.
Find the optimum solution with GA and ANN model combination.
Simulate the solution in reservoir simulator and calculate its NPV.
Compare the solution's NPV with base case NPV.
Compare the run time between the ANN-GA combination and the FieldOpt operation.

Figure 3.1: The main workflow of the master's thesis.

The main workflow of the master's thesis is shown in Fig. 3.1. The first step was to create a synthetic reservoir model using OPM Flow. After that, the necessary information was extracted from the reservoir model to build a dataset. This dataset was then divided into training, validation, and test dataset. The next step was to build the ANN model to predict cumulative oil production, cumulative water production, and cumulative water injection based on sets of well control values. This model's capability to predict was then evaluated with a performance metric. If it were not good, additional training data would be fed into the neural network. After the model's performance was considered good, a base case was set, and its NPV was calculated. Afterward, the ANN model and GA worked together to find the optimum well control values. Next, the optimum well control values obtained from the optimizer were simulated in the reservoir simulator, and its NPV was calculated. Subsequently, the NPV from the optimizer result and the base case NPV were compared, and the result was analyzed and discussed. Finally, the ANN-GA run time was compared with the FieldOpt operation run time. The following sections will explain the details of each step.

3.2 Software

OPM Flow

The Open Porous Media (OPM) Flow is a reservoir simulator that aims to represent reservoir geology, fluid behavior, and description of wells and production facilities as in commercial simulator (Rasmussen et al., 2019). This simulator supports industry-standard input and output formats. In this project, the OPM Flow was used to simulate the case of production and injection. The results generated from the simulation were then used as the datasets for the ANN model.

Python

Python is an interpreted, high-level, general-purpose programming language (Python, 2019). Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Python's interpreters are available for many operating systems. Python is also an open-source program and has a global community of programmers that discuss their code and implementations. Python was used as the compiler and programming language to set up the ANN model in this project. The python version used in this project is Python 3.7.1. The `Pandas` module is used to load the dataset, `keras` module is used to build the ANN model, and `skopt` module is used to optimize the hyperparameters with Bayesian optimization.

Jupyter Notebook

According to Jupyter website, ”The Jupyter Notebook is an open-source web application that allows people to create and share documents that contain live code, equations, visualizations, and narrative text” (Project Jupyter, 2019). Fernando Pérez announced the Jupyter Notebook in 2014 as a spin-off project from IPython called Project Jupyter. Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. This software allows the user to code in the browser and runs the code section by section. Jupyter Notebook was used as the platform for coding the ANN model and the genetic algorithm optimizer.

FieldOpt

FieldOpt is software for field development optimization. FieldOpt is an open-source, extensible, and tailor-made programming framework (Baumann et al., 2020). Its primary purpose is quick prototyping and testing of optimization strategy to solve defined field development problems. FieldOpt is written in C++ and presents an efficient integration of reservoir simulation with mathematical optimization procedures. This software can be used to optimize well control, well completion design, and well placement parameters. Currently, there are five optimization algorithms implemented in FieldOpt: asynchronous parallel pattern search (Kolda, 2005), compass search (Kolda et al., 2003), efficient global optimization (Jones et al., 1998), genetic algorithm (Chuang et al., 2015), and particle swarm optimization (Nwankwor et al., 2013).

3.3 Reservoir Model

The reservoir model is a simple-hypothetical reservoir with 60 x 60 x 2 grid cells. There are only dead-oil and water phase present in this reservoir. This reservoir has heterogeneous porosity and permeability, ranging from 0.97% until 42.47% for the porosity and from 1 mD to 1000 mD for the permeability. There are four vertical injector wells (I1, I2, I3, I4) and two deviated producer wells (P2 and P3). Water is injected from the injection wells, and oil is produced from the producer wells. The reservoir is producing for 4198 days (12 years) starting from 2 August 2015 until 29 January 2027. The visualization of the reservoir model is presented in Fig. 3.3. Fluid properties and rock properties are summarized in Table 3.1 and Table 3.2, respectively. In addition, the inputted relative permeability curve is also shown in Fig. 3.2.

Table 3.1: Initial reservoir fluid properties

Parameters	Value	Unit
Oil Density	786.50	kg/m^3
Water Density	1037.84	kg/m^3
Reservoir pressure	170	bara
STOOIP	5.19	MM sm^3
Aquifer volume	1.04	MM sm^3

Table 3.2: Reservoir rock properties

Parameters	Value	Unit
Average permeability	247.7	mD
Average porosity	21.53	%
Reservoir thickness	48	m
Rock compressibility	$4.4 \cdot 10^{-5}$	1/bar
Initial water saturation	0.1	-
Irreducible water saturation	0.1	-
Residual oil saturation	0.2	-

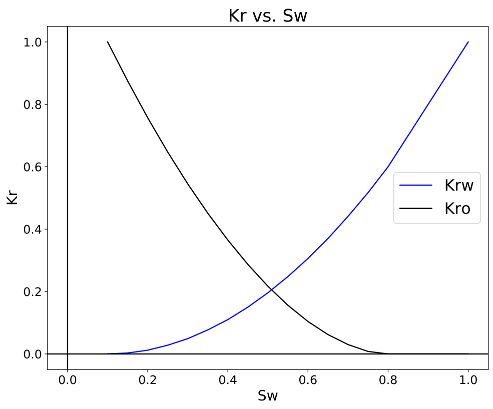
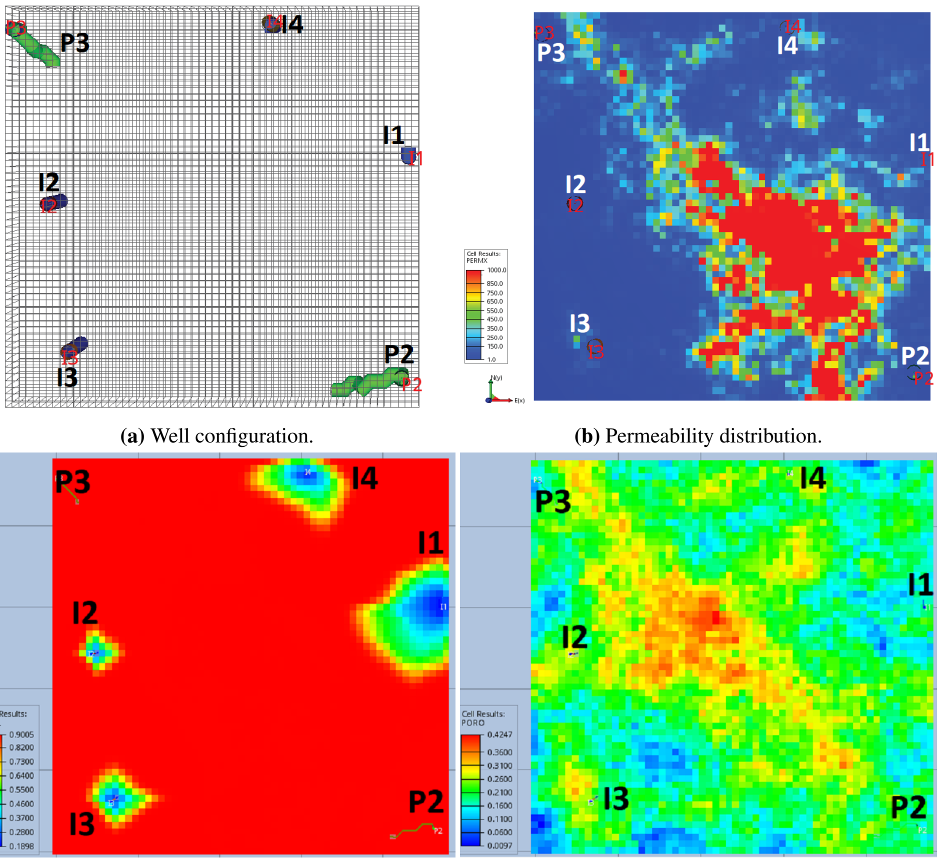


Figure 3.2: The relative permeability curve.



(a) Well configuration.

(b) Permeability distribution.

(c) Initial oil saturation distribution.

(d) Porosity distribution.

Figure 3.3: Visualization of the reservoir model.

3.4 Generating the Dataset and Data Splitting

The well control value for this reservoir simulation is well bottom-hole pressure (BHP). As BHP's value is set to constant for some time, the affected rates will be calculated from the reservoir simulator. In this reservoir simulation, BHP of injector and producer wells are input by the user, and the reservoir simulator calculates water injection rates, water production rates, and oil production rates.

In one simulation case, BHP of injector and producer wells are varied every 2 years. The reservoir is producing for almost 12 years, which means BHPs are changing 6 times during the production. As there are 4 producer wells and 2 injector wells, there will be a total of 36 different values of BHP. BHP is defined by randomly choosing a pressure value within a range of [220,280] bara for injector wells and [90,140] bara for producer wells. The cumulative oil production, cumulative water injection, and cumulative water production data are then generated by the simulator and used for building the dataset. One timestep, Δt, is defined for every 4 months. As a result, one simulation case generates 47 data points. Example of one simulation case is tabulated in Table 3.3.

Table 3.3: Example of one simulation case (N_p = cumulative oil production, W_i = cumulative water injection, W_p = cumulative water production).

Timestep	**BHP (bara)**						**x 10^6 (sm^3)**		
	I1	**I2**	**I3**	**I4**	**P2**	**P3**	N_p	W_i	W_p
1 ⋮ 8	224	270	257	241	116	97	0.30 ⋮ 1.35	0.432 ⋮ 2.49	0 ⋮ 0.12
9 ⋮ 16	259	261	240	250	124	98	1.47 ⋮ 2.16	2.81 ⋮ 5.06	0.2 ⋮ 1.18
17 ⋮ 24	234	220	270	249	118	108	2.23 ⋮ 2.64	5.36 ⋮ 7.55	1.36 ⋮ 2.83
25 ⋮ 32	226	264	277	220	101	111	2.69 ⋮ 2.97	7.87 ⋮ 10.2	3.08 ⋮ 4.88
33 ⋮ 40	273	263	274	266	112	114	3.01 ⋮ 3.21	10.6 ⋮ 13.3	5.2 ⋮ 7.56
41 ⋮ 47	230	229	231	231	147	144	3.22 ⋮ 3.29	13.6 ⋮ 14.9	7.74 ⋮ 8.94

The ANN model took the inputs (BHP data) to predict the outputs (cumulative oil production, cumulative water injection, and cumulative water production). To make a good prediction, the model needs to learn for multiple scenarios. Thus, lots of different cases were simulated. There are 45 different cases simulated in this dataset, with 43 of them used as the training dataset, 1 of them for the validation dataset, and 1 of them for the test dataset. Recall section 2.2.2 for the definition of each dataset. In total, there are 2115 data points in all dataset.

The model is expected to learn the correlation of various BHP with cumulative oil production, cumulative water injection, and cumulative water production in the training dataset. The model will be improved continuously based on its performance in the validation dataset. Finally, the model will be tested in the test dataset. This dataset will dictate whether the model can generalize to other data or not.

3.5 Building the ANN Model

This section describes the steps needed to build the ANN model.

Load the Dataset

The simulation results are tabulated in `.csv` file and then loaded into Jupyter Notebook. The `Pandas` module is used to load the dataset.

Define the Architecture and Practical Aspects of the Neural Networks

The ANN model is built using the `keras` module and based on these following setup:

- **Data Scaling**: Min-Max normalization technique is used to scale the data.
- **Activation function**: The hidden layer uses ReLU activation function. The output layer uses a linear activation function because the model will predict a real value, which classified as a linear regression problem.
- **Loss function**: In this model, the MSE is used to calculate the loss function.
- **Gradient descent and its optimization algorithm**: Mini-batch gradient descent with a batch size of 32 is used. For the gradient descent optimizer, Adam optimizer is chosen.
- **Overfitting prevention**: Early stopping technique is used to prevent the model overfits to the data. A ”patience” of 20 is set in the code, which means that if the validation error is not decreasing after 20 consecutive iterations, the algorithm will stop training. In addition, the dropout technique is also used to prevent overfitting. For example, if a dropout value of 0.1 is chosen, then 10% of the nodes in every cell state will be dropped out randomly.

- **Hyperparameters and its optimization algorithm**: There are four hyperparameters in this ANN model: learning rate α, number of layers, number of neurons, and dropout.

  These four hyperparameters will be optimized with Bayesian optimization to find the best combination of hyperparameters that gives the minimum error in validation dataset. The Bayesian optimizer will iterate until 80 evaluations. This was done with the help of `skopt` module in Python. The starting point and the search range of each hyperparameter is defined as follows:

  – *Learning rate α*. Starting point: 0.01. Search space: Real values between 0.0001 until 0.01.

  – *Number of layers*. Starting point: 1. Search space: Integer values between 1 until 5.

  – *Number of neurons*. Starting point: 60. Search space: Integer values between 50 until 500.

  – *Dropout*. Starting point: 0.1. Search space: Real values between 0.01 until 0.1.

  The process of Bayesian optimization is illustrated in Fig. 3.4.

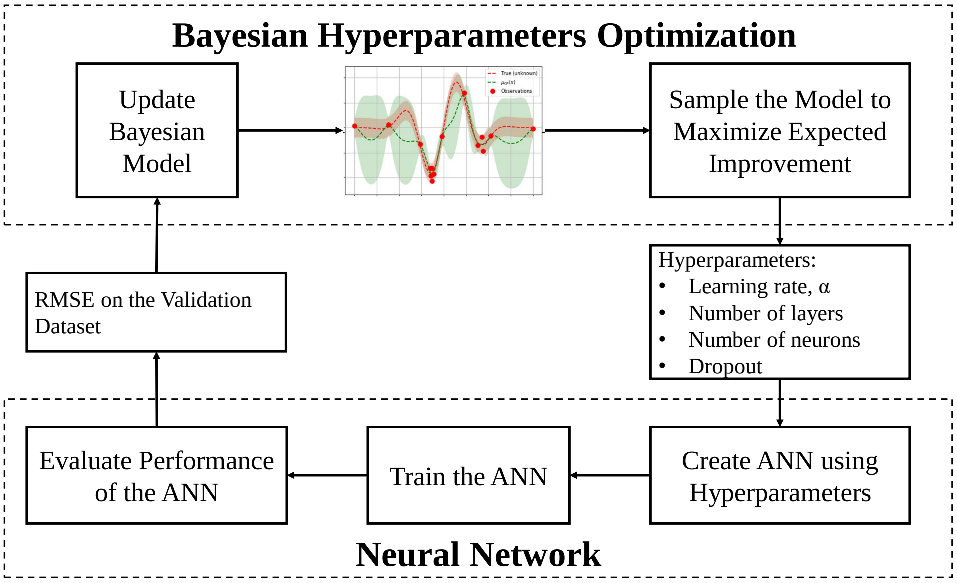

Bayesian Hyperparameters Optimization
Update Bayesian Model
True (unknown)
Observations
Sample the Model to Maximize Expected Improvement
Hyperparameters:
• Learning rate, α
• Number of layers
• Number of neurons
• Dropout
RMSE on the Validation Dataset
Evaluate Performance of the ANN
Train the ANN
Create ANN using Hyperparameters
Neural Network

Figure 3.4: The Bayesian optimization process.

Evaluate the Model Performance in the Test Dataset

The NRMSE (Normalized Root Mean Squared Error) metric is used to evaluate the model performance. NRMSE is expressed mathematically as

$$\mathrm{NRMSE} = \frac{\sqrt{\frac{1}{N}\sum\limits_{i=1}^{N}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}}{\bar{\mathbf{y}}_i} \cdot 100\% \quad , \tag{3.1}$$

where $\mathbf{y}$ is the actual data, $\hat{\mathbf{y}}$ is the model prediction, $\bar{\mathbf{y}}_i$ is the actual data mean, and N is the amount of data points. This metric shows how close is the model prediction to the real value in percentage. The lesser the NRMSE is, the better the model performance. The effect of each error on NRMSE is proportional to the squared error; thus, larger errors have a disproportionately large effect on NRMSE. Consequently, NRMSE is sensitive to outliers, which means the NRMSE should be more useful when large errors are particularly undesirable. This metric is well-suited for this study, as large errors can be translated into millions of dollars of NPV difference.

3.6 Buiding the Genetic Algorithm Optimization

This section describes the steps involved in building the genetic algorithm optimization.

Objective Function

The main goal of field development optimization problem is often finding a set of reservoir parameters that represented by a vector of variable values $\mathbf{x}$ that yields the optimal (here: maximum) objective function value $f(\mathbf{x})$:

$$\arg\max_{\mathbf{x}} f(\mathbf{x}) = \mathbf{x} \quad . \tag{3.2}$$

The objective function $f(\mathbf{x})$ defined in this study is the net present value (NPV). This function is formulated as

$$f(\mathbf{x}) = \mathrm{NPV}(\mathbf{x}) = C_{Np} - C_{Wi} - C_{Wp} \quad . \tag{3.3a}$$

$$C_{Np} = \sum_{j=1}^{N_j} \frac{6.29\, P_{\text{oil}}\,(\mathbf{N}_{p,j} - \mathbf{N}_{p,j-1})}{(1+d)^{j-1}} \quad , \quad \text{where} \quad \mathbf{N}_{p,0} = 0 \quad . \tag{3.3b}$$

$$C_{Wi} = \sum_{j=1}^{N_j} \frac{6.29\, P_{\text{water,inj}}\,(\mathbf{W}_{i,j} - \mathbf{W}_{i,j-1})}{(1+d)^{j-1}} \quad , \quad \text{where} \quad \mathbf{W}_{i,0} = 0 \quad . \tag{3.3c}$$

$$C_{Wp} = \sum_{j=1}^{N_j} \frac{6.29\, P_{\text{water,prod}}\,(\mathbf{W}_{p,j} - \mathbf{W}_{p,j-1})}{(1+d)^{j-1}} \quad , \quad \text{where} \quad \mathbf{W}_{p,0} = 0 \quad . \tag{3.3d}$$

See List of Symbols for the explanation of the symbols. P_{oil}, $P_{\text{water,inj}}$, and $P_{\text{water,prod}}$ are set constant to 50 \$/bbl, 2 \$/bbl, and 6 \$/bbl respectively. The discount rate d is set to

2% (quarterly). The BHPs in **x** are constrained to [220,280] bara for injector wells and [90,140] bara for producer wells.

The NPV is calculated by using the prediction from the ANN model. Keeping the NPV equation out of the ANN model is, in a way, sort of a hybrid modelling scheme. This is attractive because it reduces the complexity of the ANN.

Population Size

Population size is the number of chromosomes in one population. In this study, different population sizes are tested in a range of $[n_{\text{vars}}, 2 \cdot n_{\text{vars}}]$. There are 36 variables in one chromosome (36 values of BHP), thus making the range to [36, 72]. The population size of 36, 48, and 72 are chosen to be tested in this study.

Selection, Crossover, and Mutation Operation

Both roulette wheel selection and elitism selection are used in the selection operation. One chromosome with the highest fitness is carried over (elitism selection), and the remaining chromosomes are selected with roulette wheel selection to produce the next generation. Then, the chosen chromosomes with roulette wheel selection have a chance of doing a crossover, and after that, have a chance of mutating. In this study, three different probabilities of crossover and three different probabilities of mutation are tested: 25%, 50%, and 75% chance of doing crossover and 5%, 10%, and 15% chance of mutating.

Crossover technique used in this algorithm is by calculating the arithmetic mean of both parent's chromosomes. The mutation technique used in this algorithm is by adding a random number in a range of [-30,30] in every gene. The mutation still honors the producer constraint of [220,280] bara, and injector constraint of [90,140] bara, i.e., the lower boundary is chosen if the new gene is below, the lower boundary and the upper boundary is selected if the new gene is over the upper boundary.

Termination Operation

This generational process will be repeated until the maximum number of generations reached. There is no maximum NPV criteria to be met and no maximum allocated time. However, maximum NPV reached and elapsed time will be taken into consideration when choosing the best genetic algorithm setup.

Chapter 4

Results and Discussion

This chapter presents the results of building the ANN model and genetic algorithm optimization. Those results are also analyzed and discussed in this chapter.

4.1 The ANN Model

A specific ANN model was developed to predict a specific output. There were three outputs that this model intended to predict, which were cumulative oil production (N_p), cumulative water injection (W_i), and cumulative water production (W_p). In total, there were three different ANN models developed.

4.1.1 The Cumulative Oil Production Prediction Model

The convergence plot in Fig. 4.1 shows that the surrogate function already reached its minimum value for 80 evaluations at the twelfth evaluation. It seems that the maximum evaluation needed can be reduced to save more time. However, this fast convergence may only be applied to this model. Therefore, the maximum evaluation of 80 will still be used for the next prediction model. Table 4.1 shows the combination of hyperparameters that minimize the validation loss after 80 evaluations of Bayesian optimization.

Table 4.1: Optimum hyperparameters configuration for N_p model.

Hyperparameters	**Optimum Value**
Learning rate α	0.0001
Number of layers	5
Number of neurons	477
Dropout	0.01

As seen from Table 4.1, a high number of layers, number of neurons, and low dropout

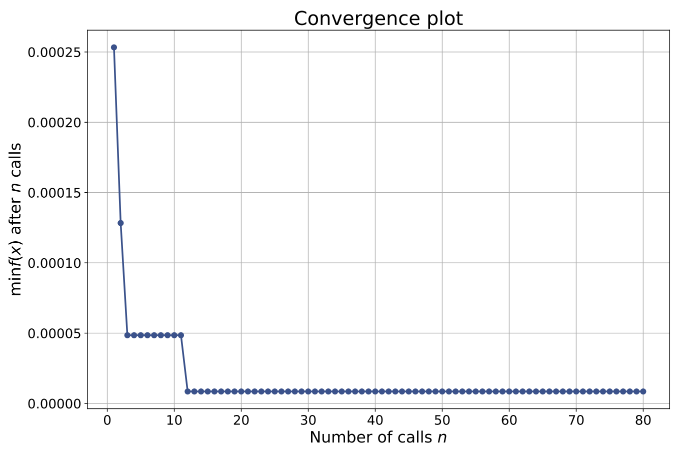


Figure 4.1: The convergence plot of the surrogate function in Bayesian optimization for cumulative oil production prediction model.

were needed to minimize the validation error. The validation NRMSE of this model is 0.56%. The addition of hidden layers and neurons increase the model complexity, and low dropout kept the complexity high. Low validation error means that there is only a small difference between the actual data and the predicted data; hence the model can generalize to data that it never learns. So, this ANN model is a complex model that able to predict data it never learns satisfactorily. However, this claim has to be proven in the test dataset. The model's prediction in the test dataset is shown in Fig. 4.2.

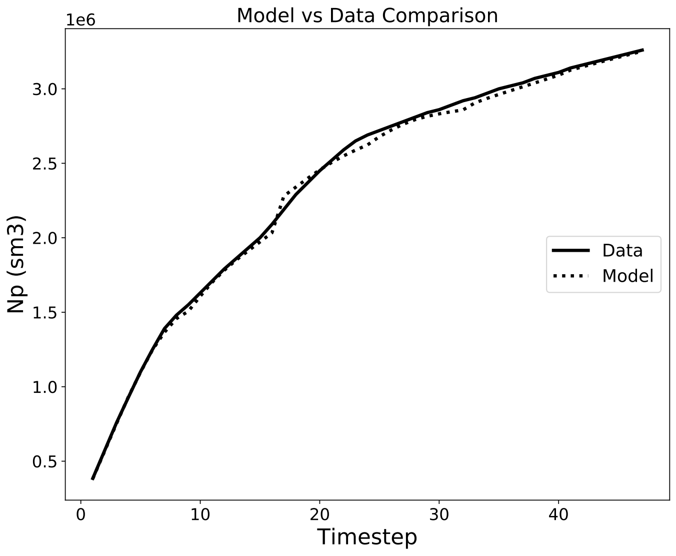


Figure 4.2: The comparison of N_p prediction and actual N_p in test dataset.

It can be seen visually from Fig. 4.2 that the model's prediction is very close to the actual data. Quantitatively, this model has an NRMSE of 1.35%, which means that this model's prediction is very close to the actual value. The value of NRMSE in the test dataset corresponds well with the value of NRMSE in the validation dataset. Small NRMSE in the validation dataset translates to small NMRSE in the test dataset. In conclusion, this model can accurately predict the cumulative oil production.

4.1.2 The Cumulative Water Injection Prediction

The convergence plot in Fig. 4.3 shows that the surrogate function already reached its minimum value for 80 evaluations at the seventeenth evaluation. For the second time, the optimization converges faster than expected. The maximum evaluation of 80 seems excessive for this model. But again, this fast convergence may not be applied to another model. Thus, the maximum evaluation of 80 will still be used for the next prediction model. The optimum combination of hyperparameters in this model is summarized in Table 4.2.

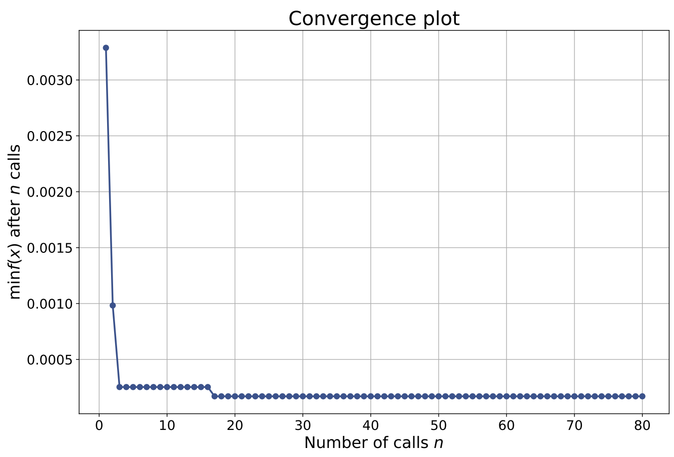


Figure 4.3: The convergence plot of the surrogate function in Bayesian optimization for cumulative water injection prediction model.

Table 4.2: Optimum hyperparameters configuration for W_i model.

Hyperparameters	**Optimum Value**
Learning rate α	0.009
Number of layers	5
Number of neurons	85
Dropout	0.01

Table 4.2 shows that the model to predict W_i has similar hyperparameters with the model to predict N_p except the number of neurons. The W_i prediction model has a smaller

number of neurons than the N_p prediction model. This could mean that it is relatively easier to predict W_i; thus, the model needs a lesser amount of neurons. This model yields an NRMSE of 4.03% in the validation dataset. Fig. 4.4 shows the model's prediction in the test dataset.

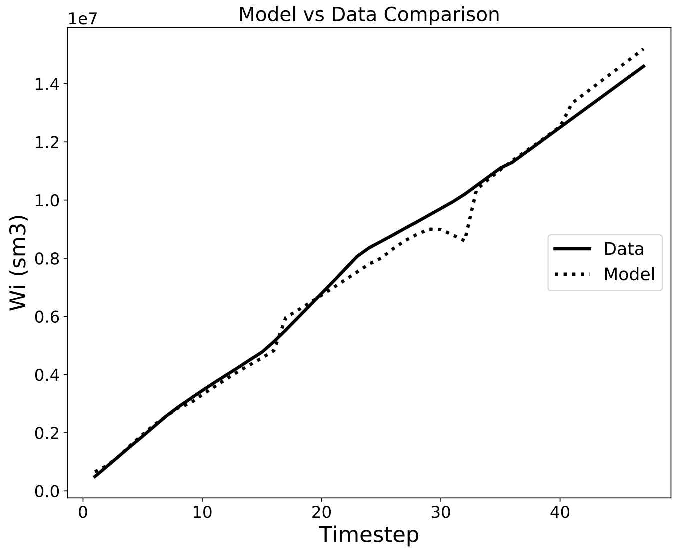


Figure 4.4: The comparison of W_i prediction and actual W_i in the test dataset.

Fig. 4.4 shows some mismatches between the data and the model. This model is not as good as the N_p prediction model. However, this model yields an NRMSE of 5.68%, which is still a low error. Thus, this model is also deemed fit to predict the cumulative water injection.

4.1.3 The Cumulative Water Production Prediction

Here, the convergence plot in Fig. 4.5 shows that the surrogate function already reached its minimum value for 80 evaluations at the nineteenth evaluation. For the third time, the optimization converges before 20 evaluations. Based on these three models, it can be concluded that 20 evaluations are enough to get an optimum combination of hyperparameters. This finding can be used for future consideration for choosing the maximum evaluation. Table 4.3 shows the optimum hyperparameters combination from Bayesian optimization.

It can be inferred from Table 4.3 that this model has 4 hidden layers and 384 neurons. This means that this model is less complex than the N_p prediction model but more complex than the W_i prediction model. This model's NRMSE in the validation dataset is 8.19%, which is higher than the N_p and W_i prediction model. High NRMSE in the validation dataset would generally mean high NRMSE in the test dataset. Fig. 4.6 shows the comparison between the W_p prediction model and the actual W_p in the test dataset.

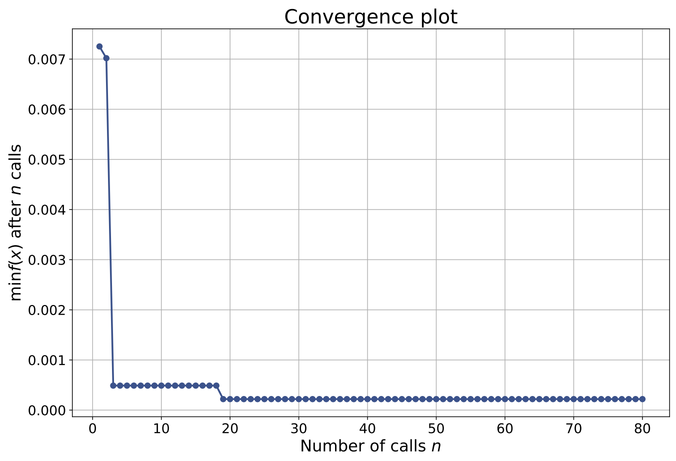


Figure 4.5: The convergence plot of the surrogate function in Bayesian optimization for cumulative water production prediction model.

Table 4.3: Optimum hyperparameters configuration for W_p prediction model.

Hyperparameters	**Optimum Value**
Learning rate α	0.003
Number of layers	4
Number of neurons	384
Dropout	0.01

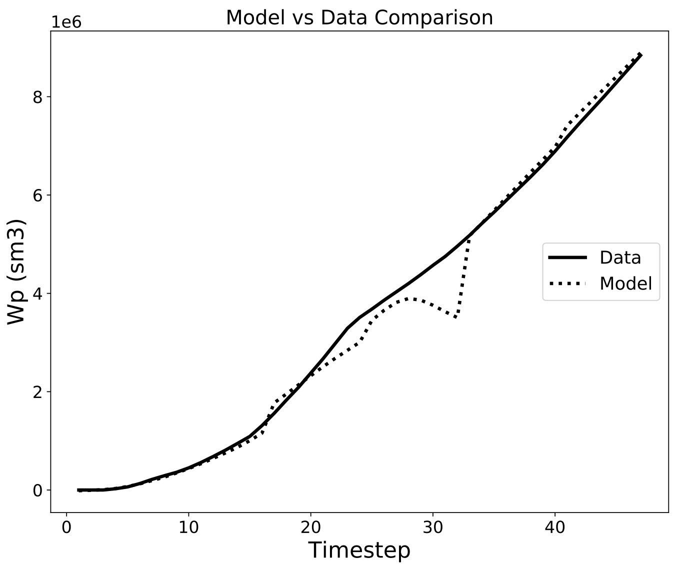


Figure 4.6: The comparison of W_p prediction and actual W_p in the test dataset.

It can be seen from Fig. 4.6 that this model performs worse than the N_p and W_i prediction model. The model predicts very well most of the time but suffers in timestep 20 until 34. This might be because the BHP variation in this range of timestep differs greatly from the BHPs in the training dataset; thus, the model did not learn/know what the right outcome is. The NRMSE for this model in the test dataset is 9.63%, which is also higher than the NRMSE of N_p and W_i prediction model in the test dataset. However, the NRMSE tolerance for this study is 10%, hence this model is used to predict the cumulative water production.

4.1.4 NPV Difference Between Model Prediction and Actual Data

To find the accuracy of those three model combined, one could calculate the NPV from the model prediction and then compare them to the NPV from the actual data using Eq. (3.3). The results are

- actual data's NPV: **5.057 x 10^8 USD**;
- model prediction's NPV: **4.962 x 10^8 USD**.

The NRMSE between the two NPVs is

$$\text{NRMSE} = \frac{\sqrt{\frac{1}{1}\sum_{i=1}^{1}(5.057\cdot 10^8 - 4.962\cdot 10^8)^2}}{\frac{(5.057\cdot 10^8 - 4.962\cdot 10^8)}{2}} \cdot 100\% = \mathbf{1.89\%} \quad .$$

This small difference gives a strong confidence in the model's prediction.

4.2 The Genetic Algorithm Optimization

The ANN model was applied in the genetic algorithm optimization to predict the N_p, W_i, and W_p instead of using the reservoir simulator. Before doing the optimization, the base case was set, and several sensitivity studies were performed to find the optimum configuration for the genetic algorithm optimization.

4.2.1 The Base Case

In the base case, BHP was picked randomly within a range of [220,280] bara for producer wells and [90,140] bara for injector wells (see Table 4.4) because there was no available information about the possible optimal solution. The base case was also used as the initial population for the optimization run. The base case NPV was then calculated using Eq. (3.3), resulting in an NPV of **5.118 x 10^8 USD**.

4.2.2 Population Size

The population size of 36, 48, and 72 were tested to determine which population size is cost-efficient for the optimization run. Generally, increasing the population size will increase the chance of getting a better solution in the next generation. However, this will

Table 4.4: The base case (N_p = cumulative oil production, W_i = cumulative water injection, W_p = cumulative water production).

Timestep	BHP (bara)						x 10^6 (sm^3)		
	I1	I2	I3	I4	P2	P3	N_p	W_i	W_p
1 ⋮ 8	258	220	222	224	113	140	0.27 ⋮ 1.18	0.39 ⋮ 2.32	0 ⋮ 0.27
9 ⋮ 16	241	244	255	241	90	103	1.32 ⋮ 2.06	2.66 ⋮ 5.12	0.38 ⋮ 1.46
17 ⋮ 24	251	273	235	280	133	135	2.13 ⋮ 2.56	5.46 ⋮ 7.73	1.62 ⋮ 3.09
25 ⋮ 32	241	280	240	280	125	142	2.61 ⋮ 2.90	8.05 ⋮ 10.4	3.33 ⋮ 5.12
33 ⋮ 40	257	266	266	237	130	145	2.93 ⋮ 3.12	10.7 ⋮ 12.8	5.36 ⋮ 7.2
41 ⋮ 47	249	271	254	263	101	142	3.14 ⋮ 3.28	13.2 ⋮ 15.5	7.54 ⋮ 9.6

also increase its computational time. Hence, the optimization results and the elapsed time of each population size were recorded.

The base case was used as the initial population. Then, each case was running until 200 generations. The crossover probability and mutation probability are set to 50% and 15%, respectively. Finally, because GA is a stochastic searching approach, there were five optimization runs performed and the mean of these runs was reported.

For the three population sizes, Fig. 4.7 shows one of the optimization runs and Table 4.5 summarizes the average elapsed time and the average maximum NPV achieved.

Table 4.5: The average maximum NPV and elapsed time for every population size.

Population Size	Average Maximum NPV	Average Elapsed Time
36	6.064 x 10^8 USD	36 minutes and 46 seconds
48	6.072 x 10^8 USD	1 hours 3 minutes and 18 seconds
72	6.312 x 10^8 USD	2 hours 17 minutes and 19 seconds

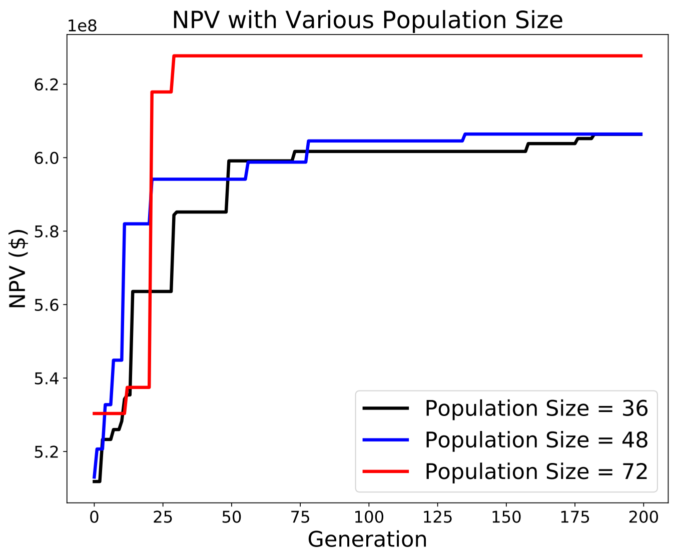


Figure 4.7: One of the optimization runs with different population sizes.

From Fig. 4.7, it can be noticed that population size of 72 yields a better NPV at the very first generation. This is the benefit of having a large population size, which will increase the likelihood of getting a better solution. However, in the end, the percentage increase of maximum NPV from population size of 36 to 72 is only 4% (from 6.064 x 10^8 USD to 6.312 x 10^8 USD), certainly not enough to outweigh its 273% increase of elapsed time (from 36 minutes and 46 seconds to 2 hours 17 minutes and 19 seconds). This is also the case for comparison between population size of 36 and 48, with an increase of only 0.13% in NPV (from 6.064 x 10^8 USD to 6.072 x 10^8 USD) but a 72% increase of elapsed time (from 36 minutes and 46 seconds to 1 hours 3 minutes and 18 seconds).

The BHP solutions of the optimization run were also similar. It seems that optimizations are converging to the same solution. Therefore, by assessing the NPV, elapsed time, and the BHP solution, the most cost-efficient population size is 36.

4.2.3 Crossover and Mutation Probability

To determine the efficient configuration of crossover and mutation probability, various configurations were tested. Crossover probability of 25%, 50%, 75% and mutation probability of 5%, 10%, 15% were tested, a total of 9 possible combination of crossover and mutation probability. Each case was running with the same initial population and the same maximum generation (200), just different crossover and mutation probability. Five optimization runs were performed for each case and the mean of these runs were reported. One of the optimization runs is shown in Fig. 4.8 and Table 4.6 tabulates the average maximum NPV achieved ranked from highest to lowest.

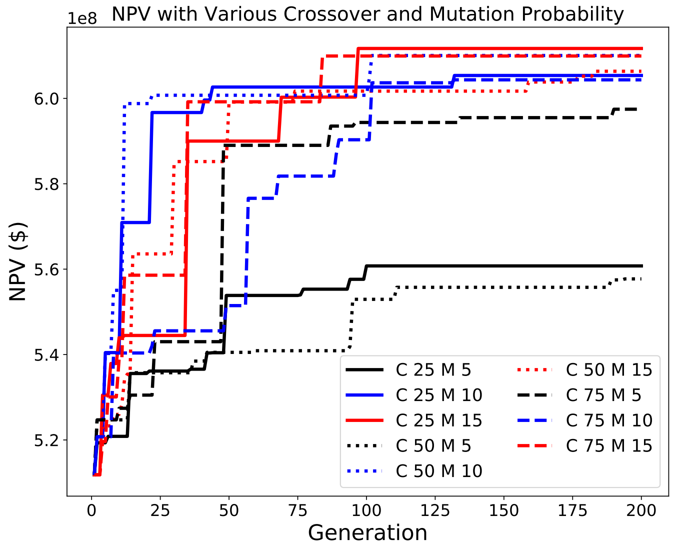


Figure 4.8: One of the optimization runs with various crossover and mutation probability (C for crossover probability and M for mutation probability).

Table 4.6: The average maximum NPV for every crossover and mutation probability ranked from highest to lowest.

Crossover Probability (%)	**Mutation Probability (%)**	**Avg. Max. NPV x 10^8 (USD)**
25	15	6.245
50	10	6.132
75	15	6.099
25	10	6.064
50	15	6.053
75	10	6.045
75	5	5.965
25	5	5.611
50	5	5.558

The elapsed time of every configuration is similar, about 36 minutes. This is because of the same population size in every case. The BHP solutions are also similar, indicating that they are converging to the same solution. From Table 4.6, it can be seen that generally lower mutation probability performs worse than higher mutation probability. However, it might not be the case for even higher mutation probability (>20%), because it can be destructive to the whole population, just like in nature. An optimum chance is needed in mutation, not too low to cause uniformity in the population, but not too high to disrupt the population.

By contrast, lower crossover probability generally performs better than higher crossover probability. Again, high crossover probability may introduce too many new variables into the population, which may divert the search to a longer or worse direction. A crossover probability and mutation probability of 25% and 15% seem to be the ”sweet spot” for this optimization run.

4.2.4 The Best Case

After deciding upon the best optimization configuration, which was the population size of 36, crossover probability of 25%, and mutation probability of 15%, the optimization was run until 3000 generations to find the maximum NPV. Only one optimization run was performed instead of the usual five because the time needed to run 5 x 3000 generations would be very long. Besides, the goal is to find the maximum NPV, which more likely to be achieved by increasing the maximum generation, not by doing the same simulation over again. Fig. 4.9 depicts the objective function development over 3000 generations.

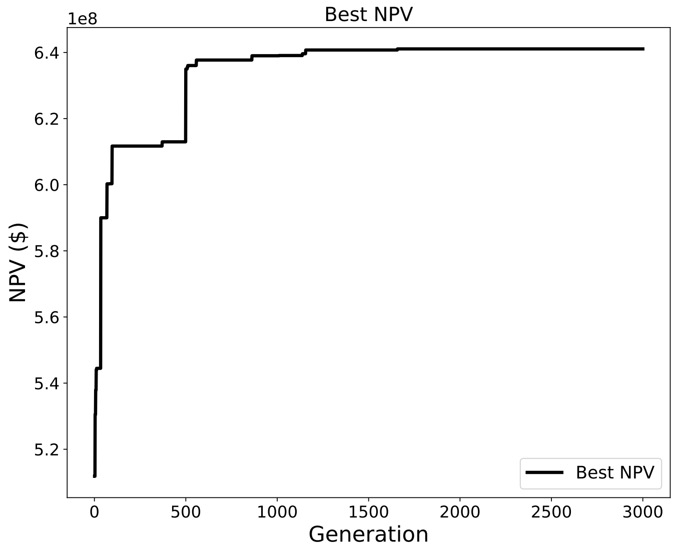


Figure 4.9: The optimization run with a population size of 36, crossover probability of 25%, mutation probability of 15%, and 3000 maximum generation.

The time elapsed to run this optimization was **9 hours 30 minutes and 31 seconds**. Fig. 4.9 shows that approximately after 500 generations, producing more generations only gives a little addition to the NPV, as the law of diminishing returns comes into play. The optimization stops finding a better solution after 1658 generations. One would probably only need 500 generations to produce satisfactory results. Nevertheless, after 3000 generations, the maximum NPV achieved from model prediction is **6.411 x 10^8 USD**.

However, this result should be confirmed in the reservoir simulation. The BHPs configuration that yields the maximum NPV was simulated and the results are tabulated in Table 4.7.

Table 4.7: The best case (N_p = cumulative oil production, W_i = cumulative water injection, W_p = cumulative water production).

Timestep	**BHP (bara)**						**x 10^6 (sm^3)**		
	I1	**I2**	**I3**	**I4**	**P2**	**P3**	N_p	W_i	W_p
1 ⋮ 8	220.0	280.0	279.6	223.2	140.0	140.0	0.19 ⋮ 1.00	0.33 ⋮ 1.84	0 ⋮ 0.02
9 ⋮ 16	220.0	279.4	279.9	220.0	140.0	140.0	1.11 ⋮ 1.76	2.05 ⋮ 3.56	0.03 ⋮ 0.37
17 ⋮ 24	220.0	280.0	279.9	220.0	140.0	140.0	1.84 ⋮ 2.29	3.78 ⋮ 5.33	0.45 ⋮ 1.18
25 ⋮ 32	220.0	280.0	279.4	220.0	140.0	140.0	2.34 ⋮ 2.68	5.56 ⋮ 7.16	1.31 ⋮ 2.31
33 ⋮ 40	220.0	279.9	259.6	220.0	140.0	140.0	2.72 ⋮ 2.95	7.38 ⋮ 8.96	2.46 ⋮ 3.63
41 ⋮ 47	220.0	220.0	220.1	220.0	140.0	139.9	2.97 ⋮ 3.09	9.16 ⋮ 10.4	3.79 ⋮ 4.8

The BHPs in Table. 4.7 are at the bounds for most wells. It seems that all producers are operating at the maximum allowed BHP to minimize the water production and avoid early water breakthrough. It means that higher water production will severely impact the NPV. Then, for most of the time, injector I2 and I3 inject water at the maximum allowed BHP while injector I1 and I4 inject water at the minimum allowed BHP. This phenomenon may be explained by the location of the injector (see Fig. 3.3). I2 and I3 are relatively farther to the producers than I1 and I4, thus can inject more water to push more oil into the producers while also avoid early water breakthrough. The permeability near I2 and I3 is also lower than I1 and I4, so it needs to pump more water to sweep the oil and reach the producers. At the final stage of the production, all injector's BHPs are at the minimum allowed BHP. This might suggest that lots of oil are already swept and produced near the end of production, thus eliminating the need for injecting more water into the reservoir.

The N_p, W_i, and W_p data in Table 4.7 were then used to calculate the NPV, resulting in an

NPV of **6.255 x 10^8 USD**, or a **0.156 x 10^8 USD** difference from the model prediction's NPV. The NRMSE between the model prediction's NPV and actual data's NPV is only **2.46%**, which is a small error. This error is not far from the error in test dataset (**1.89%**), indicating that a small error in test dataset may correspond well with other multiple predictions in the optimization run.

The optimum NPV of **6.255 x 10^8 USD** is a **22.2%** increase from the base case's NPV, or a **1.137 x 10^8 USD** increase in absolute value. This NPV increase proves the genetic algorithm's ability as a useful optimization algorithm for field development optimization problems. This NPV increase was also helped by a reliable ANN model to evaluate the objective function.

After comparing the NPV results, one could also compare the elapsed time. Here, FieldOpt software was used as a benchmark for this study. FieldOpt is a software capable of optimizing the well control values using a genetic algorithm but using the reservoir simulator for every model evaluation.

FieldOpt Optimization Run

For a fair comparison, the same optimization configuration was used in the FieldOpt optimization run (population size: 36, crossover probability: 25%, mutation probability: 15%). However, the maximum generation was only set to 500, as significantly longer running time was expected.

The time elapsed for the FieldOpt optimization run was **1 day 20 hours and 48 minutes** for a maximum generation of 500. On the other hand, an ANN-GA optimization run only took **1 hour 35 minutes** for a maximum generation of 500. It's about **43 hours difference** or **96% decrease** in running time. The significantly longer running time is because one reservoir simulation run is needed to make every population in a generation. One simulation run would need about 7 seconds to complete. For a population size of 36 and a maximum generation of 500, FieldOpt would need 36 x 500 x 7 seconds = 126.000 seconds or 35 hours just for the objective function evaluation. It is not the case if the ANN model is used. One model prediction only needs about 0.12 seconds to complete. Thus, for a 36 x 500 = 18.000 evaluations, the model only needs about 2160 seconds or 36 minutes for the objective function evaluation.

The FieldOpt optimization run achieved a maximum NPV of **6.255 x 10^8 USD** with the same solution (BHPs) as the ANN-GA optimization run, hence also the same NPV. Here, FieldOpt was able to achieve the same result as the ANN-GA model with a lesser maximum generation. It might be caused by the diminishing returns seen in the ANN-GA optimization run after 500 generations, which means later generations after the 500th generation didn't find a meaningfully better solution. Another reason is that FieldOpt tied directly to the reservoir simulator, which didn't introduce any error in NPV calculation as the model predictions did. However, given the same results achieved in significantly faster running time, this ANN-GA model seems to be an attractive method for field development optimization problems.

4.3 Sensitivity Analysis

The best case that yields an NPV of **6.255 x 10^8 USD** was calculated using Eq. (3.3) with oil price, water injection cost, water production/treatment cost, and the discount rate are set constant to 50 \$/bbl, 2 \$/bbl, 6 \$/bbl, and 2% (quarterly) respectively. A sensitivity analysis was performed to study the influence of those four components mentioned earlier. The sensitivity analysis was conducted by increasing and decreasing the component values by 20%. It is realistic to assume actual components will vary between these values. Fig. 4.10 shows the spider plot and tornado chart to visualize the sensitivity results.

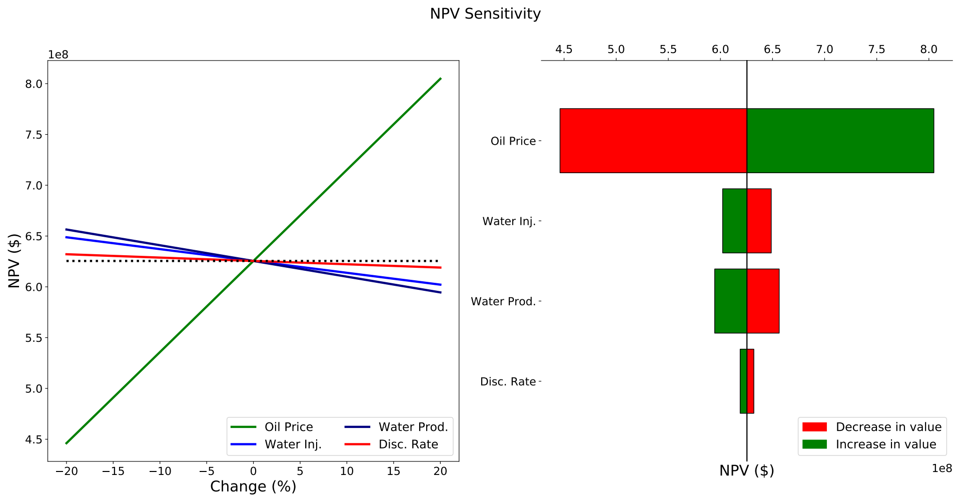


Figure 4.10: NPV sensitivity study for the oil price, water injection price, water production price, and discount rate.

As seen in Fig. 4.10, oil price changes are the most dominant factor in NPV changes. NPV will increase substantially with a 20% increase in oil price and vice versa. Oil price is also the only component that is proportional to the NPV, i.e., an increase in oil price will increase the NPV, vice versa. The other three components are inversely proportional to the NPV. These relationships are plausible because an increase in oil price will increase the revenue, thus increase the NPV, and an increase in cost will decrease the profit, thus decrease the NPV. Moreover, the least dominant factor in NPV changes is the discount rate. The lower discount rate will appreciate the present value and vice versa, albeit rather marginally. In conclusion, changes in oil prices will substantially change NPV, and changes in the discount rate will marginally change NPV.

4.4 Limitations

Even though this proposed model of ANN-GA combination seems promising, there are some limitations to keep in mind for both the ANN model and GA optimization.

Limitations of the ANN Model

- The model is only trained for a specific reservoir condition. If a change is introduced to the reservoir, e.g., the addition of a new well, different well placement, different fluid properties, adding an EOR, etc., the model should be trained all over again, rendering the previous model useless.
- Generating the dataset take some preparations and time. One simulation case only took about 7 seconds in this simple reservoir model. Therefore, it was relatively quick to make a dataset containing tens of simulation results. However, it might take hours or days for a field with hundreds of thousands or millions of grids to generate the dataset.
- Training the model may also take some considerable computational power and time. In this simple reservoir model, it only took about 30 minutes to train one ANN model. Again, it might take hours or days for a field with hundreds of thousands or millions of grids to train. Fortunately, this model only needs to be trained once if it already considered good enough.
- Currently, a reservoir simulation model is the closest thing one could make to estimate the real reservoir behavior. A reservoir simulation result may not reflect in the real field application. Consequently, an ANN model that built upon an inaccurate reservoir model would also produce inaccurate results.

Limitations of the Genetic Algorithm

- Finding an optimal solution to complex, high-dimensional, multimodal problems often requires a very costly fitness function evaluations. For instance, a structural optimization problem may require several hours or several days for one function evaluation. In this case, it may be a wiser decision to forgo the exact evaluation and use approximated fitness instead.
- GA, in many problems, actually appears to converge towards local optima or even arbitrary points rather than the global optimum of the problem. As the drawback of not using a gradient to direct its search, GA does not ”know how” to sacrifice short-term fitness for longer-term fitness. This problem may be mitigated by increasing the rate of mutation, using a different fitness function, or using a selection technique that ensures diversity in the population. This problem also proves the No Free Lunch theorem (Wolpert and Macready, 1997) which states that no model that works best for every problem.
- In any problem, the stop criterion is not clear, since the ”better” solution is only in comparison to other solutions.

Chapter 5

Conclusion and Recommendation

5.1 Conclusion

1. The developed ANN model was proven able to reliably predict cumulative oil production, cumulative water injection, and cumulative water production based on sets of well control values with an NRMSE between the model prediction's NPV and actual data's NPV of 1.89% in the test dataset.

2. The most cost-efficient population size is 36. The highest average maximum NPV was achieved with crossover probability of 25%, and mutation probability of 15%.

3. The genetic algorithm optimization was run for 3000 generations for 9 hours 30 minutes and 31 seconds but achieved convergence after 1658 generations. It found successfully well control values that increase the value of NPV compared to the base case.

4. The BHPs configuration that yields the maximum NPV was simulated and resulting in an NPV of 6.255 x 10^8 USD, a 0.156 x 10^8 USD difference from the model prediction's NPV, or an NRMSE of 2.46%.

5. The optimum NPV of 6.255 x 10^8 USD is a 22.2% increase from the base case NPV of 5.118 x 10^8 USD, or a 1.137 x 10^8 USD increase in absolute value.

6. The FieldOpt optimization was run until 500 generations for 1 day 20 hours and 48 minutes, compared to the ANN-GA run, which took only 1 hour and 35 minutes for the same maximum generation. The ANN-GA run was 43 hours faster than the FieldOpt run, or 96% decrease in running time.

7. The ANN-GA optimization run and the FieldOpt run achieved the same maximum NPV of 6.255 x 10^8 USD with the same BHPs.

8. The sensitivity analysis revealed that oil prices are the most influential factor in NPV changes, and the discount rates are the least influential factor in NPV changes.

9. Even though the ANN-GA model seems promising, there are some limitations to be remembered for both the ANN model and GA optimization.

5.2 Recommendation

1. Given the time constraint for this thesis, this proposed ANN-GA model was only tested in a simple synthetic reservoir model. One can try to test the ANN-GA model in a bigger and more complex reservoir model to test its applicability in the oil and gas industry.
2. Given the limitations of the genetic algorithm, one may also try other optimization algorithms such as particle swarm optimization and compass search to see if there are any advantages than using a genetic algorithm.
3. In this thesis, the ANN-GA model is built in the python programming language, while the FieldOpt software is built in the C++ programming language. One may try to build both the ANN-GA model and FieldOpt in the same programming language or build a compatible code in both languages. It can ease collaboration if there are further advancements in the machine learning model and the optimization techniques.

Bibliography

Aziz, K., Settari, A., 1979. Petroleum reservoir simulation. Applied Science Publishers. URL: `https://books.google.no/books?id=GJ5TAAAAMAAJ`.

Baumann, E.J., Dale, S.I., Bellout, M.C., 2020. FieldOpt: A powerful and effective programming framework tailored for field development optimization. Computers & Geosciences 135, 104379. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0098300419301013`, doi:`10.1016/j.cageo.2019.104379`.

Berg, C.F., 2019. Lecture Notes Reservoir Simulation. NTNU.

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization, in: Proceedings of the 24th International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA. p. 2546–2554.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research 13, 281–305.

Chauhan, N.S., 2019. Introduction to Artificial Neural Networks (ANN). Towards Data Science.

Chuang, Y.C., Chen, C.T., Hwang, C., 2015. A real-coded genetic algorithm with a direction-based crossover operator. Information Sciences 305, 320–348. URL: `https://linkinghub.elsevier.com/retrieve/pii/S002002551500064X`, doi:`10.1016/j.ins.2015.01.026`.

Claesen, M., Moor, B.D., 2015. Hyperparameter search in machine learning. CoRR abs/1502.02127. URL: `http://arxiv.org/abs/1502.02127`, `arXiv:1502.02127`.

Deb, K., Beyer, H.G., 2001. Self-Adaptive Genetic Algorithms with Simulated Binary Crossover. Evolutionary Computation 9, 197–221. URL: `http://www.mitpressjournals.org/doi/10.1162/106365601750190406`, doi:`10.1162/106365601750190406`.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S., 2017. Gans trained by a two time-scale update rule converge to a local nash equilibrium., in: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (Eds.), NIPS, pp. 6626–6637. URL: http://dblp.uni-trier.de/db/conf/nips/nips2017.html#HeuselRUNH17.

Hey, A.J.G. (Ed.), 2009. The fourth paradigm: data-intensive scientific discovery. Microsoft Research, Redmond, Washington.

Holland, J.H., 1975. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. University of Michigan Press, Ann Arbor.

Holmes, J.A., 1983. Enhancements to the strongly coupled, fully implicit well model: wellbore crossflow modeling and collective well control. Soc. Pet. Eng. AIME, Pap.; (United States) SPE 11259.

Jones, D.R., Schonlau, M., Welch, W.J., 1998. Efficient global optimization of expensive black-box functions. Journal of Global Optimization 13, 455–492. URL: http://link.springer.com/10.1023/A:1008306431147, doi:10.1023/A:1008306431147.

Khan, M.I., Islam, M., 2007. Chapter 6 - reservoir engineering and secondary recovery, in: Khan, M.I., Islam, M. (Eds.), The Petroleum Engineering Handbook: Sustainable Operations. Gulf Publishing Company, pp. 189 – 241. URL: http://www.sciencedirect.com/science/article/pii/B9781933762128500131, doi:https://doi.org/10.1016/B978-1-933762-12-8.50013-1.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. URL: http://arxiv.org/abs/1412.6980. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Kolda, T.G., 2005. Revisiting Asynchronous Parallel Pattern Search for Nonlinear Optimization. SIAM Journal on Optimization 16, 563–586. URL: http://epubs.siam.org/doi/10.1137/040603589, doi:10.1137/040603589.

Kolda, T.G., Lewis, R.M., Torczon, V., 2003. Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods. SIAM Review 45, 385–482. URL: http://epubs.siam.org/doi/10.1137/S003614450242889, doi:10.1137/S003614450242889.

Mitchell, T.M., 1997. Machine Learning. McGraw-Hill series in computer science, McGraw-Hill, New York.

Mockus, J., 2009. Bayesian global optimization., in: Floudas, C.A., Pardalos, P.M. (Eds.), Encyclopedia of Optimization. Springer, pp. 183–187. URL: `http://dblp.uni-trier.de/db/reference/opt/opt2009.html#Mockus09`.

Mohaghegh, S.D., 2011. Reservoir simulation and modeling based on artificial intelligence and data mining (AI&DM). Journal of Natural Gas Science and Engineering 3, 697–705. URL: `https://linkinghub.elsevier.com/retrieve/pii/S1875510011001090`, doi:`10.1016/j.jngse.2011.08.003`.

Ng, A., 2013a. The Problem of Overfitting. Coursera.

Ng, A., 2013b. RMSprop. Coursera.

Nwankwor, E., Nagar, A.K., Reid, D.C., 2013. Hybrid differential evolution and particle swarm optimization for optimal well placement. Computational Geosciences 17, 249–268. URL: `http://link.springer.com/10.1007/s10596-012-9328-9`, doi:`10.1007/s10596-012-9328-9`.

Project Jupyter, 2019. Jupyter Notebook. URL: `https://jupyter.org/`.

Python, 2019. Python. URL: `https://www.python.org/`.

Qian, N., 1999. On the momentum term in gradient descent learning algorithms. Neural Networks 12, 145 – 151. URL: `http://www.sciencedirect.com/science/article/pii/S0893608098001166`, doi:`https://doi.org/10.1016/S0893-6080(98)00116-6`.

Rasmussen, A.F., Sandve, T.H., Bao, K., Lauser, A., Hove, J., Skaflestad, B., Klöfkorn, R., Blatt, M., Rustad, A.B., Sævareid, O., Lie, K.A., Thune, A., 2019. The open porous media flow reservoir simulator. CoRR abs/1910.06059. URL: `http://dblp.uni-trier.de/db/journals/corr/corr1910.html#abs-1910-06059`.

Rosenblatt, F., 1958. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review 65, 386–408. URL: `http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519`, doi:`10.1037/h0042519`.

Ruder, S., 2016. An overview of gradient descent optimization algorithms. URL: `http://arxiv.org/abs/1609.04747`.

Sarma, P., Durlofsky, L., Aziz, K., 2005. Efficient closed-loop production optimization under uncertainty. doi:`10.2523/94241-MS`.

Shahriari, B., Swersky, K., Wang, Z., Adams, R.P., de Freitas, N., 2016. Taking the human out of the loop: A review of bayesian optimization. Proceedings of the IEEE 104, 148–175. URL: `http://dblp.uni-trier.de/db/journals/pieee/pieee104.html#ShahriariSWAF16`.

Snoek, J., Larochelle, H., Adams, R.P., 2012. Practical bayesian optimization of machine learning algorithms., in: Bartlett, P.L., Pereira, F.C.N., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (Eds.), NIPS, pp. 2960–2968. URL: `http://dblp.uni-trier.de/db/conf/nips/nips2012.html#SnoekLA12`.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research 15, 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

Sutton, R.S., 1986. Two problems with backpropagation and other steepest-descent learning procedures for networks, in: Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Hillsdale, NJ: Erlbaum.

Takahashi, M., Kita, H., 2001. A crossover operator using independent component analysis for real-coded genetic algorithms, in: Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), pp. 643–649 vol. 1.

Wolpert, D., Macready, W., 1997. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1, 67–82. URL: `http://ieeexplore.ieee.org/document/585893/`, doi:`10.1109/4235.585893`.

List of Symbols

Symbols definitions for the continuous equations

λ_α	*mobility* of phase α defined as $\lambda_\alpha = k_{r,\alpha}/\mu_\alpha$.
μ_α	*viscosity* of phase α.
ϕ	*porosity* defined as the ratio of pore volume to bulk volume.
ϕ_{ref}	*reference porosity*, varying in space but constant in time
ρ_α	*density* of phase α at the reservoir condition. For water phase, $\rho_w = b_w\rho_{S,w}$. For oil phase, $\rho_o = b_o(\rho_{S,o} + r_{go}\rho_{S,g}$. For gas phase, $\rho_g = b_g(\rho_{S,g} + r_{og}\rho_{S,o})$.
$\rho_{S,\alpha}$	*surface density* of phase α at 1 atm.
$\mathbf{g}$	*gravitational acceleration* vector.
$\mathbf{u}_\alpha$	*component velocity* of component α.
$\mathbf{v}_\alpha$	*phase velocity* of phase α.
b_α	*shrinkage/expansion factor* for phase α defined as the ratio of surface volume at standard conditions to reservoir volume: $b_\alpha = V_{\text{surface},\alpha}/V_{\text{reservoir},\alpha}$.
$k_{r,\alpha}$	*relative permeability* for phase α.
m_ϕ	*pore volume multiplier* as function of pressure
p_α	*phase pressure* for phase α
$p_{c,\alpha\beta}$	*capillary pressure* between phases α and β.
q_α	*well outflux density* of pseudo component α.
r_{go}	*ratio of dissolved gas to oil* in the oleic phase. Usually called r_S in other literature.

r_{og}	*ratio of vaporized oil to gas* in the gaseous phase. Usually called r_V in other literature.
s_α	*saturation* of phase α defined as the ratio of phase α volume to pore volume. The summation of the saturation of all phases equals to 1.
K	*permeability* of the porous media.

Symbols definitions for the discrete equations

Δt	*time step length* for the current Euler step.
$\Delta\Phi_{\alpha,ij}$	*potential difference* for phase α for the connection between cells i and j.
$C(i)$	*connection* from cell i.
g	*gravitational acceleration* in the z-direction.
m_T	*transmissibility multiplier* as function of pressure.
T_{ij}	*transmissibility* factor for a connection.
$U(\alpha, ij)$	*upwind cell* for phase α for the connection between cells i and j.
u_α	*surface volume flux* of pseudo component α.
V	*cell volume*.
v_α	*volume flux* of phase α.
z_i	*depth* of center of cell i.

Symbols definitions for the well models

$h_{w,j}$	*pressure difference* within the wellbore between connection j and the well's bottom-hole datum depth.
p_j	*pressure* of the grid block that contains the connection j.
$p_{\text{bhp},w}$	*bottom-hole pressure* of well w.
$T_{\alpha,j}$	*mobility* for phase α at the connection j.
$T_{w,j}$	*connection transmissibility* factor.

Symbols definitions for the NPV

6.29	*conversion factor* from sm^3 to bbl.
x	*vector* of BHP.
C_{Np}	*NPV component* of oil production.
C_{Wi}	*NPV component* of water injection.

C_{Wp}	*NPV component* of water production.
d	*discount factor*.
j	*timestep*, one timestep equals 4 months.
N_j	*number of timesteps*.
$N_{p,0}$	*cumulative oil production* at timestep 0 in sm^3.
$N_{p,j-1}$	*cumulative oil production* at timestep $j-1$ in sm^3.
$N_{p,j}$	*cumulative oil production* at timestep j in sm^3.
$P_{\text{water,inj}}$	*water injection price* in \$/bbl.
$P_{\text{water,prod}}$	*water treatment price* in \$/bbl.
P_{oil}	*oil price* in \$/bbl.
$W_{i,0}$	*cumulative water injection* at timestep 0 in sm^3.
$W_{i,j-1}$	*cumulative water injection* at timestep $j-1$ in sm^3.
$W_{i,j}$	*cumulative water injection* at timestep j in sm^3.
$W_{p,0}$	*cumulative water production* at timestep 0 in sm^3.
$W_{p,j-1}$	*cumulative water production* at timestep $j-1$ in sm^3.
$W_{p,j}$	*cumulative water production* at timestep j in sm^3.
NPV	*net present value* in \$.

Appendix

The code for building the ANN model and the genetic algorithm

```python
import pandas as pd
import math
import tensorflow as tf
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error as mse
import time

#Plotting
import matplotlib.pyplot as plt

# Bayesian optimizer
import skopt
from skopt import gp_minimize, forest_minimize
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_convergence
from skopt.plots import plot_objective, plot_evaluations
from skopt.plots import plot_histogram, plot_objective_2D
from skopt.utils import use_named_args

# ANN
from numpy import array
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.layers import LeakyReLU
from keras import backend as K
from keras.callbacks import History
from keras import optimizers
from keras.layers import Dropout
from keras.models import load_model
from keras.callbacks import EarlyStopping
from keras.callbacks import TensorBoard

# Random seed reproducibility
import os
os.environ['PYTHONHASHSEED'] = '0'
import random as rn

# Warnings
import warnings
warnings.filterwarnings("ignore")

# load dataset
```

```
dataframe = pd.read_csv("Data.csv", sep=';' , header=0)
dataset = dataframe.values
# split into input (X) and output (Y) variables
data = pd.DataFrame(dataset, columns=["timestep", "BHPI1", "BHPI2", "BHPI3
    ", "BHPI4", "BHPP2", "BHPP3",
                                     "FOPT", "FWIT", "FWPT"])
data.head()

def ANN_model(X_train, y_train, learning_rate, dropout, n_units, n_layers)
    :
    """
    Compiles an LSTM model given hyperparameters

    Returns:
    -------
    model: obj
        Keras LSTM model
    """
    # Reset the network graphs for results reproducability in Keras
    N = 17 #Seed number
    np.random.seed(N)
    rn.seed(N)
    session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
                                  inter_op_parallelism_threads=1)
    tf.set_random_seed(N)
    K.clear_session()
    sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
    K.set_session(sess)

    # Define model
    model = Sequential()
    if n_layers == 1:
        # Input layer
        model.add(Dense(n_units, activation='relu',
                        input_dim=X_train.shape[1]))
    else:
        model.add(Dense(n_units, activation='relu',
                        input_dim=X_train.shape[1]))
        for i in range(n_layers-1):
            model.add(Dense(n_units, activation='relu'))
    # Add dropout
    model.add(Dropout(dropout))
    # Add dense layer
    model.add(Dense(y_train.shape[1], activation='linear'))
    #model.add(LeakyReLU(alpha=0.1))
    adam = optimizers.Adam(lr=learning_rate)
    model.compile(optimizer=adam, loss='mse')

    return model

def data_split(data, feature_list, target):
    """
    Splits data into train, validation and test sets
    Parameters
    ----------
    data : Dataframe, shape = [n_points, n_features]
        All feature data
    feature_list : list
        List of features to include into training
    target : list
        List of target variables to include into training
    Returns
    -------
    - Train, validation and test sets, array-like
    - Scaling functions for X and y
    """
    X = data[feature_list]
    y = data[target]

    #train_size = int(0.6*X.shape[0])
    #test_size = int(0.85*X.shape[0])
    train_size = 2021
    test_size = 2068
    n_target = int(len(target)) # Number of target outputs

    # Splitting
    (X_train,
     y_train) = (np.array(X[:train_size]),
                 np.array(y[:train_size]).reshape(-1, n_target))
    (X_val,
     y_val) = (np.array(X[train_size : test_size]),
               np.array(y[train_size : test_size]).reshape(-1, n_target))
    (X_test,
     y_test) = (np.array(X[test_size:]),
                np.array(y[test_size:]).reshape(-1, n_target))
    # Scaling
    scalerX = MinMaxScaler().fit(X_train)
    scalery = MinMaxScaler().fit(y_train)
    X_train = scalerX.transform(X_train)
    y_train = scalery.transform(y_train)
    X_val = scalerX.transform(X_val)
    y_val = scalery.transform(y_val)
    X_test = scalerX.transform(X_test)
    y_test = scalery.transform(y_test)

    return X_train, y_train, X_val, y_val, X_test, y_test, scalerX,
    scalery, train_size, test_size

[X_train, y_train, X_val,
 y_val, X_test, y_test,
 scalerX_fopt, scalery_fopt,
 train_size, test_size] = data_split(data=data,
                                feature_list=["timestep", "BHPI1", "BHPI2"
    , "BHPI3", "BHPI4", "BHPP2", "BHPP3"],
                                target=["FOPT"])

#default hyperparameters
default_parameters = [0.01, 1, 60, 0.1]
num_epochs = 100

dim_learning_rate = Real(low=1e-4, high=1e-2, prior='log-uniform', name='
    learning_rate')
dim_n_layers = Integer(low=1, high=5, name='n_layers')
dim_n_units = Integer(low=50, high=500, name='n_units')
```

```
dim_dropout = Real(low=0.01, high=0.1, prior='log-uniform', name='dropout'
    )

dimensions = [dim_learning_rate, dim_n_layers, dim_n_units, dim_dropout]

path_best_model_fopt = 'best_model_fopt_v3edit_minmax.keras'

best_val_loss = 100.0

@use_named_args(dimensions=dimensions)
def fitness_fopt(learning_rate, n_layers, n_units, dropout):
    """
    Hyper-parameters:
    learning_rate:     Learning-rate for the optimizer.
    n_layers:          Number of dense layers.
    n_units:           Number of nodes in each dense layer.
    dropout:           Percentage of the nodes retained.
    window:            Number of window
    """

    # Print the hyper-parameters.
    print('learning rate: {0:.1e}'.format(learning_rate))
    print('num_dense_layers:', n_layers)
    print('num_dense_nodes:', n_units)
    print('dropout:', dropout)

    model = ANN_model(X_train=X_train,
                      y_train=y_train,
                      learning_rate=learning_rate,
                      dropout=dropout,
                      n_units=n_units,
                      n_layers=n_layers)

    # Use Keras to train the model.
    patience = 20
    es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience
    =patience, restore_best_weights=True)
    history = model.fit(X_train, y_train, verbose=1, epochs=num_epochs,
                        validation_data=(X_val, y_val), callbacks=[es])

    # Get the classification accuracy on the validation-set
    # after the last training-epoch.
    if es.stopped_epoch == 0:
        val_loss = history.history['val_loss'][-1]
    else:
        val_loss = history.history['val_loss'][es.stopped_epoch-patience]

    # Print the classification accuracy.
    print()
    print('val_loss:', val_loss)
    print()

    # Save the model if it improves on the best-found performance.
    # We use the global keyword so we update the variable outside
    # of this function.
    global best_val_loss

    # If the classification accuracy of the saved model is improved ...
    if val_loss < best_val_loss:
        # Save the new model to harddisk.
        model.save(path_best_model_fopt)

        # Update the classification accuracy.
        best_val_loss = val_loss

    # Delete the Keras model with these hyper-parameters from memory.
    del model

    # Clear the Keras session, otherwise it will keep adding new
    # models to the same TensorFlow graph each time we create
    # a model with a different set of hyper-parameters.
    K.clear_session()

    # NOTE: Scikit-optimize does minimization so it tries to
    # find a set of hyper-parameters with the LOWEST fitness-value.
    # Because we are interested in the HIGHEST classification
    # accuracy, we need to negate this number so it can be minimized.
    return val_loss

search_result = gp_minimize(func=fitness_fopt,
                            dimensions=dimensions,
                            acq_func='EI', # Expected Improvement.
                            n_calls=80,
                            x0=default_parameters)

plot_convergence(search_result)
space = search_result.space
space.point_to_dict(search_result.x)
model_fopt = load_model(path_best_model_fopt)

test_preds_fopt = model_fopt.predict(X_test)
train_preds_fopt = model_fopt.predict(X_train)
val_preds_fopt = model_fopt.predict(X_val)
# invert predictions
train_preds_fopt = scalery_fopt.inverse_transform(train_preds_fopt)
y_train_fopt = scalery_fopt.inverse_transform(y_train)
val_preds_fopt = scalery_fopt.inverse_transform(val_preds_fopt)
y_val_fopt = scalery_fopt.inverse_transform(y_val)
test_preds_fopt = scalery_fopt.inverse_transform(test_preds_fopt)
y_test_fopt = scalery_fopt.inverse_transform(y_test)
X_test = scalerX_fopt.inverse_transform(X_test)
X_val = scalerX_fopt.inverse_transform(X_val)
X_train = scalerX_fopt.inverse_transform(X_train)

print ("Train RMSE is", np.sqrt(mse(y_train_fopt, train_preds_fopt))/np.
    mean(y_train_fopt)*100)
print ("Val RMSE is", np.sqrt(mse(y_val_fopt, val_preds_fopt))/np.mean(
    y_val_fopt)*100)
print ("Test RMSE is", np.sqrt(mse(y_test_fopt, test_preds_fopt))/np.mean(
    y_test_fopt)*100)

xdata = np.arange(1,48,1)
plt.rcParams['figure.figsize'] = (10,8)
plt.rc('axes', titlesize=22)      # fontsize of the axes title
```

```python
plt.rc('axes', labelsize=20)     # fontsize of the x and y labels
plt.rc('xtick', labelsize=15)     # fontsize of the tick labels
plt.rc('ytick', labelsize=15)     # fontsize of the tick labels
plt.rc('legend', fontsize=20)     # legend fontsize
plt.rc('figure', titlesize=20)  # fontsize of the figure title
plt.figure(2)
plt.title('Test Model vs Data Comparison')
plt.xlabel('Timestep')
plt.ylabel('Np (sm3)')
plt.plot(xdata, y_test_fopt[:,0], label='Data', color='black', linewidth
    =3)
plt.plot(xdata, test_preds_fopt[:,0], label='Model', linestyle=':', color=
    'black', linewidth=3)
plt.legend(['Data', 'Model'], loc='right')
plt.ticklabel_format(axis="y", style="sci", scilimits=(0,0))
plt.savefig('Test Model and Data Comparison (edit version).pdf')
plt.show()

# Change the target output for Wi and Wp model and repeat above code
# Below is the GA code

#NPV Calculation
def NPV(Cum_Oil, Cum_Inj, Cum_Water, Oil_Price, Injection_Price,
    Treatment_Price, Disc):
    NPV_calc = 0
    for i in range (0,Cum_Oil.shape[0]):
        if i == 0:
            FOPT = (Cum_Oil[i,0]*6.29*Oil_Price)/(np.power((1+Disc/4),((i
    +1)/4)))
            FWIT = (Cum_Inj[i,0]*6.29*Injection_Price)/(np.power((1+Disc
    /4),((i+1)/4)))
            FWPT = (Cum_Water[i,0]*6.29*Treatment_Price)/(np.power((1+Disc
    /4),((i+1)/4)))
        else:
            FOPT = ((Cum_Oil[i,0]-Cum_Oil[i-1,0])*6.29*Oil_Price)/(np.
    power((1+Disc/4),((i+1)/4)))
            FWIT = ((Cum_Inj[i,0]-Cum_Inj[i-1,0])*6.29*Injection_Price)/(
    np.power((1+Disc/4),((i+1)/4)))
            FWPT = ((Cum_Water[i,0]-Cum_Water[i-1,0])*6.29*Treatment_Price
    )/(np.power((1+Disc/4),((i+1)/4)))
        NPV_calc = NPV_calc + FOPT + FWIT + FWPT
    return NPV_calc

def generate_population(size, I_bound, P_bound):
    lower_I_boundary, upper_I_boundary = I_bound
    lower_P_boundary, upper_P_boundary = P_bound

    population = []
    for i in range(size):
        individual = {
            "I11": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I21": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I31": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I41": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P21": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P31": rn.uniform(lower_P_boundary, upper_P_boundary),
            "I12": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I22": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I32": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I42": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P22": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P32": rn.uniform(lower_P_boundary, upper_P_boundary),
            "I13": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I23": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I33": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I43": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P23": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P33": rn.uniform(lower_P_boundary, upper_P_boundary),
            "I14": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I24": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I34": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I44": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P24": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P34": rn.uniform(lower_P_boundary, upper_P_boundary),
            "I15": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I25": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I35": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I45": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P25": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P35": rn.uniform(lower_P_boundary, upper_P_boundary),
            "I16": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I26": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I36": rn.uniform(lower_I_boundary, upper_I_boundary),
            "I46": rn.uniform(lower_I_boundary, upper_I_boundary),
            "P26": rn.uniform(lower_P_boundary, upper_P_boundary),
            "P36": rn.uniform(lower_P_boundary, upper_P_boundary),
        }
        population.append(individual)

    return population

def apply_function(individual):
    I11 = individual["I11"]; I11 = np.full((8,), I11); X1  = np.arange
    (1,9,1)
    I21 = individual["I21"]; I21 = np.full((8,), I21)
    I31 = individual["I31"]; I31 = np.full((8,), I31)
    I41 = individual["I41"]; I41 = np.full((8,), I41)
    P21 = individual["P21"]; P21 = np.full((8,), P21)
    P31 = individual["P31"]; P31 = np.full((8,), P31)
    I12 = individual["I12"]; I12 = np.full((8,), I12); X2  = np.arange
    (9,17,1)
    I22 = individual["I22"]; I22 = np.full((8,), I22)
    I32 = individual["I32"]; I32 = np.full((8,), I32)
    I42 = individual["I42"]; I42 = np.full((8,), I42)
    P22 = individual["P22"]; P22 = np.full((8,), P22)
    P32 = individual["P32"]; P32 = np.full((8,), P32)
    I13 = individual["I13"]; I13 = np.full((8,), I13); X3  = np.arange
    (17,25,1)
    I23 = individual["I23"]; I23 = np.full((8,), I23)
    I33 = individual["I33"]; I33 = np.full((8,), I33)
    I43 = individual["I43"]; I43 = np.full((8,), I43)
    P23 = individual["P23"]; P23 = np.full((8,), P23)
    P33 = individual["P33"]; P33 = np.full((8,), P33)
```

```python
    I14 = individual["I14"]; I14 = np.full((8,), I14); X4  = np.arange
    (25,33,1)
    I24 = individual["I24"]; I24 = np.full((8,), I24)
    I34 = individual["I34"]; I34 = np.full((8,), I34)
    I44 = individual["I44"]; I44 = np.full((8,), I44)
    P24 = individual["P24"]; P24 = np.full((8,), P24)
    P34 = individual["P34"]; P34 = np.full((8,), P34)
    I15 = individual["I15"]; I15 = np.full((8,), I15); X5  = np.arange
    (33,41,1)
    I25 = individual["I25"]; I25 = np.full((8,), I25)
    I35 = individual["I35"]; I35 = np.full((8,), I35)
    I45 = individual["I45"]; I45 = np.full((8,), I45)
    P25 = individual["P25"]; P25 = np.full((8,), P25)
    P35 = individual["P35"]; P35 = np.full((8,), P35)
    I16 = individual["I16"]; I16 = np.full((7,), I16); X6  = np.arange
    (41,48,1)
    I26 = individual["I26"]; I26 = np.full((7,), I26)
    I36 = individual["I36"]; I36 = np.full((7,), I36)
    I46 = individual["I46"]; I46 = np.full((7,), I46)
    P26 = individual["P26"]; P26 = np.full((7,), P26)
    P36 = individual["P36"]; P36 = np.full((7,), P36)
    X_new1 = np.vstack((X1,I11,I21,I31,I41,P21,P31)); X_new1 = X_new1.T
    X_new2 = np.vstack((X2,I12,I22,I32,I42,P22,P32)); X_new2 = X_new2.T
    X_new3 = np.vstack((X3,I13,I23,I33,I43,P23,P33)); X_new3 = X_new3.T
    X_new4 = np.vstack((X4,I14,I24,I34,I44,P24,P34)); X_new4 = X_new4.T
    X_new5 = np.vstack((X5,I15,I25,I35,I45,P25,P35)); X_new5 = X_new5.T
    X_new6 = np.vstack((X6,I16,I26,I36,I46,P26,P36)); X_new6 = X_new6.T
    X_test = np.vstack((X_new1,X_new2,X_new3,X_new4,X_new5,X_new6))
    X_test_fwpt = scalerX_fwpt.transform(X_test)
    X_test = scalerX_fopt.transform(X_test)

    test_preds_fopt_ga = model_fopt.predict(X_test)
    test_preds_fopt_ga = scalery_fopt.inverse_transform(test_preds_fopt_ga
    )
    test_preds_fwit_ga = model_fwit.predict(X_test)
    test_preds_fwit_ga = scalery_fwit.inverse_transform(test_preds_fwit_ga
    )
    test_preds_fwpt_ga = model_fwpt.predict(X_test_fwpt)
    test_preds_fwpt_ga = scalery_fwpt.inverse_transform(test_preds_fwpt_ga
    )
    NPV_test = NPV(test_preds_fopt_ga, test_preds_fwit_ga,
    test_preds_fwpt_ga, 47, -2, -6, 0.08)
    return NPV_test

def choice_by_roulette(sorted_population, fitness_sum):
    offset = 0
    normalized_fitness_sum = fitness_sum

    lowest_fitness = apply_function(sorted_population[0])
    if lowest_fitness < 0:
        offset = -lowest_fitness
        normalized_fitness_sum += offset * len(sorted_population)

    draw = rn.uniform(0, 1)

    accumulated = 0
    for individual in sorted_population:
        fitness = apply_function(individual) + offset
        probability = fitness / normalized_fitness_sum
        accumulated += probability

        if draw <= accumulated:
            return individual

def sort_population_by_fitness(population):
    return sorted(population, key=apply_function)

def crossover(individual_a, individual_b):
    I11a = individual_a["I11"]; I11b = individual_b["I11"]
    I21a = individual_a["I21"]; I21b = individual_b["I21"]
    I31a = individual_a["I31"]; I31b = individual_b["I31"]
    I41a = individual_a["I41"]; I41b = individual_b["I41"]
    P21a = individual_a["P21"]; P21b = individual_b["P21"]
    P31a = individual_a["P31"]; P31b = individual_b["P31"]

    I12a = individual_a["I12"]; I12b = individual_b["I12"]
    I22a = individual_a["I22"]; I22b = individual_b["I22"]
    I32a = individual_a["I32"]; I32b = individual_b["I32"]
    I42a = individual_a["I42"]; I42b = individual_b["I42"]
    P22a = individual_a["P22"]; P22b = individual_b["P22"]
    P32a = individual_a["P32"]; P32b = individual_b["P32"]

    I13a = individual_a["I13"]; I13b = individual_b["I13"]
    I23a = individual_a["I23"]; I23b = individual_b["I23"]
    I33a = individual_a["I33"]; I33b = individual_b["I33"]
    I43a = individual_a["I43"]; I43b = individual_b["I43"]
    P23a = individual_a["P23"]; P23b = individual_b["P23"]
    P33a = individual_a["P33"]; P33b = individual_b["P33"]

    I14a = individual_a["I14"]; I14b = individual_b["I14"]
    I24a = individual_a["I24"]; I24b = individual_b["I24"]
    I34a = individual_a["I34"]; I34b = individual_b["I34"]
    I44a = individual_a["I44"]; I44b = individual_b["I44"]
    P24a = individual_a["P24"]; P24b = individual_b["P24"]
    P34a = individual_a["P34"]; P34b = individual_b["P34"]

    I15a = individual_a["I15"]; I15b = individual_b["I15"]
    I25a = individual_a["I25"]; I25b = individual_b["I25"]
    I35a = individual_a["I35"]; I35b = individual_b["I35"]
    I45a = individual_a["I45"]; I45b = individual_b["I45"]
    P25a = individual_a["P25"]; P25b = individual_b["P25"]
    P35a = individual_a["P35"]; P35b = individual_b["P35"]

    I16a = individual_a["I16"]; I16b = individual_b["I16"]
    I26a = individual_a["I26"]; I26b = individual_b["I26"]
    I36a = individual_a["I36"]; I36b = individual_b["I36"]
    I46a = individual_a["I46"]; I46b = individual_b["I46"]
    P26a = individual_a["P26"]; P26b = individual_b["P26"]
    P36a = individual_a["P36"]; P36b = individual_b["P36"]

    return {"I11": (I11a + I11b) / 2, "I21": (I21a + I21b) / 2, "I31": (
    I31a + I31b) / 2,
            "I41": (I41a + I41b) / 2, "P21": (P21a + P21b) / 2, "P31": (
    P31a + P31b) / 2,
           "I12": (I12a + I12b) / 2, "I22": (I22a + I22b) / 2, "I32": (I32a + I32b) / 2,
            "I42": (I42a + I42b) / 2, "P22": (P22a + P22b) / 2, "P32": (P32a + P32b) / 2,
           "I13": (I13a + I13b) / 2, "I23": (I23a + I23b) / 2, "I33": (I33a + I33b) / 2,
            "I43": (I43a + I43b) / 2, "P23": (P23a + P23b) / 2, "P33": (P33a + P33b) / 2,
           "I14": (I14a + I14b) / 2, "I24": (I24a + I24b) / 2, "I34": (I34a + I34b) / 2,
            "I44": (I44a + I44b) / 2, "P24": (P24a + P24b) / 2, "P34": (P34a + P34b) / 2,
           "I15": (I15a + I15b) / 2, "I25": (I25a + I25b) / 2, "I35": (I35a + I35b) / 2,
            "I45": (I45a + I45b) / 2, "P25": (P25a + P25b) / 2, "P35": (P35a + P35b) / 2,
           "I16": (I16a + I16b) / 2, "I26": (I26a + I26b) / 2, "I36": (I36a + I36b) / 2,
            "I46": (I46a + I46b) / 2, "P26": (P26a + P26b) / 2, "P36": (P36a + P36b) / 2}

def mutate(individual):
    lower_I_boundary, upper_I_boundary = (220, 280)
    lower_P_boundary, upper_P_boundary = (90, 140)

    a = -30
    b = 30

    next_I11 = individual["I11"] + rn.uniform(a, b)
    next_I21 = individual["I21"] + rn.uniform(a, b)
    next_I31 = individual["I31"] + rn.uniform(a, b)
    next_I41 = individual["I41"] + rn.uniform(a, b)
    next_P21 = individual["P21"] + rn.uniform(a, b)
    next_P31 = individual["P31"] + rn.uniform(a, b)
    next_I12 = individual["I12"] + rn.uniform(a, b)
    next_I22 = individual["I22"] + rn.uniform(a, b)
    next_I32 = individual["I32"] + rn.uniform(a, b)
    next_I42 = individual["I42"] + rn.uniform(a, b)
    next_P22 = individual["P22"] + rn.uniform(a, b)
    next_P32 = individual["P32"] + rn.uniform(a, b)
    next_I13 = individual["I13"] + rn.uniform(a, b)
    next_I23 = individual["I23"] + rn.uniform(a, b)
    next_I33 = individual["I33"] + rn.uniform(a, b)
    next_I43 = individual["I43"] + rn.uniform(a, b)
    next_P23 = individual["P23"] + rn.uniform(a, b)
    next_P33 = individual["P33"] + rn.uniform(a, b)
    next_I14 = individual["I14"] + rn.uniform(a, b)
    next_I24 = individual["I24"] + rn.uniform(a, b)
    next_I34 = individual["I34"] + rn.uniform(a, b)
    next_I44 = individual["I44"] + rn.uniform(a, b)
    next_P24 = individual["P24"] + rn.uniform(a, b)
    next_P34 = individual["P34"] + rn.uniform(a, b)
    next_I15 = individual["I15"] + rn.uniform(a, b)
    next_I25 = individual["I25"] + rn.uniform(a, b)
    next_I35 = individual["I35"] + rn.uniform(a, b)
    next_I45 = individual["I45"] + rn.uniform(a, b)
    next_P25 = individual["P25"] + rn.uniform(a, b)
    next_P35 = individual["P35"] + rn.uniform(a, b)
    next_I16 = individual["I16"] + rn.uniform(a, b)
    next_I26 = individual["I26"] + rn.uniform(a, b)
    next_I36 = individual["I36"] + rn.uniform(a, b)
    next_I46 = individual["I46"] + rn.uniform(a, b)
    next_P26 = individual["P26"] + rn.uniform(a, b)
    next_P36 = individual["P36"] + rn.uniform(a, b)

    # Guarantee we keep inside boundaries
    next_I11 = min(max(next_I11, lower_I_boundary), upper_I_boundary)
    next_I21 = min(max(next_I21, lower_I_boundary), upper_I_boundary)
    next_I31 = min(max(next_I31, lower_I_boundary), upper_I_boundary)
    next_I41 = min(max(next_I41, lower_I_boundary), upper_I_boundary)
    next_P21 = min(max(next_P21, lower_P_boundary), upper_P_boundary)
    next_P31 = min(max(next_P31, lower_P_boundary), upper_P_boundary)
    next_I12 = min(max(next_I12, lower_I_boundary), upper_I_boundary)
    next_I22 = min(max(next_I22, lower_I_boundary), upper_I_boundary)
    next_I32 = min(max(next_I32, lower_I_boundary), upper_I_boundary)
    next_I42 = min(max(next_I42, lower_I_boundary), upper_I_boundary)
    next_P22 = min(max(next_P22, lower_P_boundary), upper_P_boundary)
    next_P32 = min(max(next_P32, lower_P_boundary), upper_P_boundary)
    next_I13 = min(max(next_I13, lower_I_boundary), upper_I_boundary)
    next_I23 = min(max(next_I23, lower_I_boundary), upper_I_boundary)
    next_I33 = min(max(next_I33, lower_I_boundary), upper_I_boundary)
    next_I43 = min(max(next_I43, lower_I_boundary), upper_I_boundary)
    next_P23 = min(max(next_P23, lower_P_boundary), upper_P_boundary)
    next_P33 = min(max(next_P33, lower_P_boundary), upper_P_boundary)
    next_I14 = min(max(next_I14, lower_I_boundary), upper_I_boundary)
    next_I24 = min(max(next_I24, lower_I_boundary), upper_I_boundary)
    next_I34 = min(max(next_I34, lower_I_boundary), upper_I_boundary)
    next_I44 = min(max(next_I44, lower_I_boundary), upper_I_boundary)
    next_P24 = min(max(next_P24, lower_P_boundary), upper_P_boundary)
    next_P34 = min(max(next_P34, lower_P_boundary), upper_P_boundary)
    next_I15 = min(max(next_I15, lower_I_boundary), upper_I_boundary)
    next_I25 = min(max(next_I25, lower_I_boundary), upper_I_boundary)
    next_I35 = min(max(next_I35, lower_I_boundary), upper_I_boundary)
    next_I45 = min(max(next_I45, lower_I_boundary), upper_I_boundary)
    next_P25 = min(max(next_P25, lower_P_boundary), upper_P_boundary)
    next_P35 = min(max(next_P35, lower_P_boundary), upper_P_boundary)
    next_I16 = min(max(next_I16, lower_I_boundary), upper_I_boundary)
    next_I26 = min(max(next_I26, lower_I_boundary), upper_I_boundary)
    next_I36 = min(max(next_I36, lower_I_boundary), upper_I_boundary)
    next_I46 = min(max(next_I46, lower_I_boundary), upper_I_boundary)
    next_P26 = min(max(next_P26, lower_P_boundary), upper_P_boundary)
    next_P36 = min(max(next_P36, lower_P_boundary), upper_P_boundary)

    return {"I11": next_I11, "I21": next_I21, "I31": next_I31, "I41":
    next_I41, "P21": next_P21, "P31": next_P31,
            "I12": next_I12, "I22": next_I22, "I32": next_I32, "I42":
    next_I42, "P22": next_P22, "P32": next_P32,
            "I13": next_I13, "I23": next_I23, "I33": next_I33, "I43":
    next_I43, "P23": next_P23, "P33": next_P33,
            "I14": next_I14, "I24": next_I24, "I34": next_I34, "I44":
    next_I44, "P24": next_P24, "P34": next_P34,
            "I15": next_I15, "I25": next_I25, "I35": next_I35, "I45":
    next_I45, "P25": next_P25, "P35": next_P35,
```

```python
            "I16": next_I16, "I26": next_I26, "I36": next_I36, "I46": next_I46, "P26": next_P26, "P36": next_P36}

def make_next_generation(previous_population):
    next_generation = []
    sorted_by_fitness_population = sort_population_by_fitness(previous_population)
    population_size = len(previous_population)
    fitness_sum = sum(apply_function(individual) for individual in population)

    next_generation.append(sorted_by_fitness_population[-1])
    for i in range(population_size-1):
        first_choice = choice_by_roulette(sorted_by_fitness_population, fitness_sum)
        second_choice = choice_by_roulette(sorted_by_fitness_population, fitness_sum)

        if rn.randint(0,100) < 25:
            individual = crossover(first_choice, second_choice)
        else: individual = first_choice

        if rn.randint(0,100) < 15:
            individual = mutate(individual)
        next_generation.append(individual)

    return next_generation

model_fopt = load_model(path_best_model_fopt)
model_fwit = load_model(path_best_model_fwit)
model_fwpt = load_model(path_best_model_fwpt)

generations = 3000

population = generate_population(size=36, I_bound=(220, 280), P_bound=(90, 140))

start = time.time()
i = 1
NPV_step = []
while True:
    print(f"GENERATION {i}")

    best_individual = sort_population_by_fitness(population)[-1]
    best_npv = apply_function(best_individual)
    NPV_step.append(best_npv)
    print("{:.3e}".format(best_npv))

    if i == generations:
        break

    i += 1

    population = make_next_generation(population)


print("\n BEST RESULT")
print(best_individual, "{:.3e}".format(best_npv))

end = time.time()
hours, rem = divmod(end-start, 3600)
minutes, seconds = divmod(rem, 60)
print("Time elapsed: ", int(hours) ,"hours", int(minutes) ,"minutes", int(
    seconds) ,"seconds")
np.save('NPV_step.npy', NPV_step)

plt.figure()
plt.title('Best NPV')
plt.xlabel('Generation')
plt.ylabel('NPV ($)')
plt.plot(np.arange(1,generations+1,1), NPV_step[:], label='Best NPV',
    color='black', linewidth=3)
plt.legend(loc='lower right')
plt.savefig('Best NPV.pdf')
plt.show()
```

The input file of the OPM model

```
RUNSPEC
TITLE
ECL 5-SPOT 60x60

PATHS
'ECLINC' './include/' /
/

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
DIMENS
60 60 2 /

-- ____________________________________________________
OIL
WATER
-- IMPES: def. solution

-- ____________________________________________________
METRIC
-- unit specification

-- ____________________________________________________
TABDIMS
--   1)   2)   3)   4)   5)  6)
  1    1    20   50   1   20 /
-- Describes size of saturation and PVT tables,
-- also the # of fluids-in-place regions
-- 1) # of sat tables entered (def. 1)
-- 2) # of PVT tables entered (def. 1)
-- 3) max # of sat nodes in any sat table (def. 20)
-- 4) max # of pressure nodes in table (def. 20)
-- 5) max # of FIP regions (def. 1)
-- 6) max # of Rs nodes in live oil PVT table (def. 20)

-- ____________________________________________________
WELLDIMS
```

```
-- 1) 2) 3) 4)
   20  100  2  20 /
-- 1) max # of wells in model (def. 0)
-- 2) max # of connections per well (def. 0)
-- 3) max # of groups in model (def. 0)
-- 4) max # of wells in any one group (def. 0)

WSEGDIMS
    5  200  50  5 /

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
START
1 'FEB' 2015 /

NSTACK
25 /

-- ____________________________________________________
UNIFOUT
UNIFIN

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
GRID
INIT
GRIDFILE
2 /
-- Contains GRID, PROPS and REGIONS summary of data
-- Request init and grid file, necessary for
-- post processing of the simulation with floviz

-- ____________________________________________________
-- RPTGRID orig
-- 0 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 /
-- RPTGRID -- commented out
-- 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 /
-- RPTGRID
-- DX DY DZ TRANX TRANY TRANZ NNC ALLNNC FAULTS /
-- Output of DX, DY, DZ, PERMX, PERMY, PERMZ,
-- MULTZ, PORO and TOPS data is requested, and
-- of the calculated pore volumes and X, Y and
-- Z transmissibilities

-- ____________________________________________________
-- Size of cells in X direction
DX
3600*24
3600*24 /

-- Size of cells in Y direction
DY
3600*24
3600*24 /

-- Size of cells in Z direction
DZ
3600*12
3600*12/
```

```

TOPS
3600*1700/

-- ________________________________________________________
-- PERMX
INCLUDE
'$ECLINC/permx_01_lyr_21X.INC' /
-- 'permporo_lyr21/permx_21_lyr.in' /

-- ________________________________________________________
-- PERMY
INCLUDE
'$ECLINC/permy_01_lyr_21X.INC' /
-- 'permporo_lyr21/permy_21_lyr.in' /

-- ________________________________________________________
-- PERMZ
INCLUDE
'$ECLINC/permz_01_lyr_21X.INC' /
--'permporo_lyr21/permz_21_lyr.in' /

-- ________________________________________________________
-- PORO
INCLUDE
'$ECLINC/poro_01_lyr_21X.INC' /
--'permporo_lyr21/poro_21_lyr.in' /

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
PROPS

-- ________________________________________________________
-- PVT

INCLUDE
'$ECLINC/ECL_5SPOT_PROPS_PVDO_MULT_MRST.INC' /

INCLUDE
'$ECLINC/ECL_5SPOT_PROPS_MRST.INC' /

-- ________________________________________________________
RPTPROPS
1  1  1  0  1  1  1  1 /
-- OUTPUT CONTROLS FOR PROPS DATA
-- Activated for SOF3, SWFN, SGFN,
-- PVTW, PVDG, DENSITY AND ROCK keywords

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
REGIONS

SATNUM
3600*1
3600*1/

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
SOLUTION
-- initial state of solution variables

-- ____________________________________________________
EQUIL
-- Data for initialising fluids to potential equilibrium
-- DATUM  DATUM OWC   OWC   GOC   GOC  RSVD  RVVD  SOLN
-- DEPTH  PRESS DEPTH PCOW  DEPTH PCOG TABLE TABLE METH
    1700  170   2200  0     0     0    0     0     0
--    1)  2)    3)    4)    5)    6)    7)    8)    9)
/
-- 1) Datum depth
-- 2) Pressure at datum depth
-- 3) Depth of water oil contact, if no water
--    initially present it can be below reservoir
-- 4) Oil-water capillary pressure at the water contact
-- 5) Depth of the gas-oil contact
-- 6) Gas-oil capillary pressure at the gas-oil contact
-- 7) Integer selecting the type of
--    initialization for live black oil
-- 8) Integer selecting the type of initialization
--    for black oil runs with wet gas
-- 9) Integer (N) defining the accuracy of
--    the initial fluids in place calculation.

-- ____________________________________________________
RPTSOL
0 0 1 /

-- 1: PRESSURE: Output of initial oil pressures

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
SUMMARY
SEPARATE
RUNSUM
--NARROW
RPTSMRY
1 /

SUMTHIN
50 /

--SKIP

-- ____________________________________________________
-- PRESSURE
FPR           Field pressure
FPRH          Field reservoir pressure (hydrocarbon?)
FPRP          Field pressure weighted by pore volume
ALL
-- ____________________________________________________
-- PORE VOLUME
FRPV          Pore Volume at Reservoir
FOPV          Pore Volume containing Oil
FHPV          Pore volume containing hydrocardon
FORMW         Total stock tank oil produced by water influx

-- ____________________________________________________
-- INIT VOLUME
-- FGIP          Gas init in place
-- FGIPG         Gas init in place (gas phase)
-- FGIPL         Gas init in place (liquid phase)
FOIP          Oil init in place
FOIPG         Oil init in place (gas phase)
FOIPL         Oil init in place (liquid phase)
FWIP          Water initially in place

-- ____________________________________________________
-- CURRENT CPU USAGE IN SECONDS

TCPU
PERFORMA


-- ____________________________________________________
-- WELL CONNECTION FACTORS
-- INCLUDE
-- ECL_5SPOT_CTFAC.INC /
-- CTFAC
-- * /
-- /

-- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
SCHEDULE
-- CONTROLS ON OUTPUT AT EACH REPORT TIME

-- ____________________________________________________
RPTSCHED
RESTART=1
/

-- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
--   0 0 1 1 0 0 2 2 2 0  0  2  0  1  0
--   0 0 0 0 0 0 0 0 0 0  0  0  0  0  0
--   0 0 0 0 0 0 0 0 1 0  0  0  0  0  0 /

-- 1: PRESSURE: Output of grid block pressures
-- 14: WELSPECS

-- RPTSCHED
-- -- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
--   0 0 0 0 0 0 0 0 0 0  0  0  0  0  0
--   0 0 0 0 0 0 0 0 0 0  0  0  0  0  0
--   0 0 0 0 0 0 0 0 0 0  0  0  0  0  0 /

-- RPTSCHED
-- 1 1 1 1 1 0 0 0 1 0 0 2 0 1 2 0 0
-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 /

-- ____________________________________________________
-- REQUEST RESTART FILE
RPTRST
 'NORST=1'/

-- ____________________________________________________
```

```
INCLUDE
    'include/ECL_5SPOT_SCH.INC' /
-- ____________________________________________________
TUNING
-- min/max timestep (3 sections)
-- 0.1  30  /
1*  1*  /
-- (1,1) TSINIT Max lngth next time step
-- (1,2) TSMAXZ Max lngth time steps after the next
-- 5*  0.1  /
/
-- (2,1) TRGTTE Target time truncation error
-- (2,2) TRGCNV Target non-linear convergence error
-- (2,3) TRGMBE Target material balance error
-- (2,4) TRGLCV Target linear convergence error
-- (2,5) XXXTTE Maximum time truncation error
-- (2,6) XXXCNV Maximum non-linear convergence error
2*   100  /
-- (3,1) NEWTMX Max # of Newton iters in a time step
-- (3,2) NEWTMN Min # of Newton iters in a time step
-- (3,3) LITMAX Max # of linear iters in a Newton
-- iter (def. 25)

-- ____________________________________________________
-- TSTEP

-- 160 years
-- 160*365 /

-- 60 years
-- 60*365 /

-- 48 years
-- 48*365 /
-- 240*73 /

-- 32 years
-- 32*365 /
-- 160*73 /

-- 16 years
-- 16*365 /
-- 80*73 /

-- 12 years
-- 60*73 /

-- 8 years
-- 40*73 /

-- 1 day
-- 1 /

-- ____________________________________________________
END
```