PROJECT REPORT

ON

Artificial Neural Network-Based Fault Classifier & Distance Locator for Double-Circuit Transmission Lines

Submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology in **ELECTRICAL ENGINEERING**

Under the guidance of

Dr. Abhishek Yadav

BY:

Sanchit Sayala (52083)

Suraj Negi (52091)

Tushar Sharma (52093)

DEPARTMENT OF ELECTRICAL ENGINEERING,

COLLEGE OF TECHNOLOGY

GOVIND BALLABH PANT UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

PANTNAGAR (U.K.), INDIA 263145

JULY, 2021

DECLARATION

We declare that this written submission represents our ideas in our own words and where other's ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented any idea /data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

DATE ……

PLACE …….

NAME OF STUDENTS

Sanchit Sayala (52083)

Suraj Negi (52091)

Tushar Sharma (52093)

CERTIFICATE

This is to certified that project work entitled **"Artificial Neural Network-Based Fault Classifier & Distance Locator for Double-Circuit Transmission Lines"** has been carried out in fulfilment of requirement for the award of the degree of the Bachelor of Technology in Electrical Engineering submitted in the department of ELECTRICAL ENGINEERING COLLEGE OF TECHHNOLOGY PANTNAGAR under the guidance of Dr. Abhishek Yadav. And the embodies the result of original work, and studies are carried out by

NAME OF STUDENTS

Sanchit Sayala (52083)

Suraj Negi (52091)

Tushar Sharma (52093)

PANTNAGAR

DATE-09\07\2021

Dr. Abhishek Yadav

(Project Guide)

APPROVAL

The project entitled "**Artificial Neural Network-Based Fault Classifier & Distance Locator for Double-Circuit Transmission Lines"** is here by approved a creditable study of an engineering subject as a prerequisite to degree for which has been submitted.

Name of Committee Members:

- Dr. ABHISHEK YADAV (Assistant Professor and Project Guide) _____
- Dr. AJAY SRIVASTAVA (Professor) _____
- DR. SUNIL SINGH (Associate Professor) _____
- DR.RAVI SAXENA(Associate Professor) _____

_____
Dr. A K SWAMI
(Head of the Department)

DEPARTMENT OF ELECTRICAL ENGINEERING,
COLLEGE OF TECHNOLOGY,
Gobind Bhallabh Pant University of Agriculture and Technology, PANTNAGAR

ACKNOWLEDGEMENT

This project is not the work of solely our team but it's a result of combined efforts put in by many people. We feel honored in expressing our profound sense of gratitude and sincere thanks to our project guide **Dr. Abhishek Yadav** for his gracious efforts, guidance and wise council without which completion of this project would not possible. it was his guidance that encouraged us to expedite our project process and complete in time .

We are thankful to Dr. A K SWAMI, Dr. SUDHA ARORA, Dr. AJAY SRIVASTAVA, DR.S K GOEL, DR.SUNIL SINGH and other faculty member of the department of the electrical engineering for their intellectual support throughout the course and project work. The paucity of words does not compromise for extending our thanks to all for support and encouragement.

We are indebted to all our friends and colleagues who so ever has contributed directly and indirectly to provide help to carry out the project work.

Sanchit Sayala (52083)

Suraj Negi (52091)

Tushar Sharma (52093)

DATED -09-07-2021

Contents

1. Introduction

1.1. Overview

This project approaches using two different methods for fault classification and distance location in a double circuit transmission lines, using artificial neural networks. The single and modular artificial neural networks were developed for determining the fault distance location under varying types of faults in both the circuits. The proposed method uses the voltages and currents signals available at only the local end of the line. The model of the example power system is developed using Matlab/Simulink software. Effects of variations in power system parameters, for example, fault inception angle, fault resistance, fault type and distance to fault have been investigated extensively on the performance of the neural network based protection scheme (for all ten faults in both the circuits). Thus, the present work considers the entire range of possible operating conditions, which has not been reported earlier. It is adaptive to variation in power system parameters, network changes and works successfully under a variety of operating conditions. The comparative results of single and modular neural network indicate that the modular approach gives correct fault location with better accuracy.

1.2. Literature Review

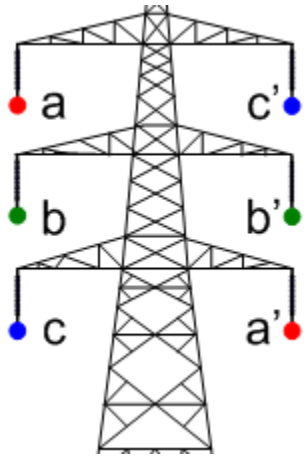
Protection of double-circuit transmission lines poses additional problems due to zero sequence mutual coupling between faulted and healthy circuits during earth faults .The nature of mutual coupling is highly variable; and it is affected by network changes such as switching in/out of one of the parallel lines, thus causing under reach/overreach of conventional distance relaying. Most of the research on ANN-based protection schemes has been carried out for single-circuit transmission lines. An adaptive distance protection of double-circuit line using zero sequence thevenin equivalent impedance and compensation factor for mutual coupling to increase the reach and selectivity of relay has been developed in. Fault classification using ANN for one circuit of parallel double-circuit line has been reported. Artificial neural network has emerged as a relaying tool for protection of power system equipments. ANN has pattern recognition, classification, generalization, and fault tolerance capability. ANN has been widely used for developing protective relaying schemes for transmission lines protection. The Artificial Neural Network is a mathematical model inspired by biological neural networks. It consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. It has the special ability of learning which implies that it can estimate the output of a system based on its experience on a set of previously trained data.

A neural network based protection technique for combined 275 kV/400 kV double-circuit transmission lines has been proposed by Q. Y. Xuan in 1998. Similarly, R. K. Aggarwal, Q. Y. Xuan, R. W. Dunn, A. T. Johns, and A. Bennett presented a novel fault classification technique of double-circuit lines based on a combined unsupervised/supervised neural network.

2. Double Circuit transmission line

Double Circuit Transmission Line refers to the arrangement in which a total of six conductors are provided to make two different Transmission Circuits. In the Double Circuit Transmission Line, there are two circuits each consisting of three conductors corresponding to three phases. Both the circuits in the Double Circuit Transmission Line are mounted or run through the same Transmission Line. In the Double Circuit Transmission Line, bundle conductors are mostly used.

A general thumb rule is that, if the numbers of conductors per bundle are 2 then it will be 220 kV line while if the number of conductors per bundle is 4 then it will be 400 kV line. This is a general thumb rule and variation is expected.



3. Fault in Transmission lines

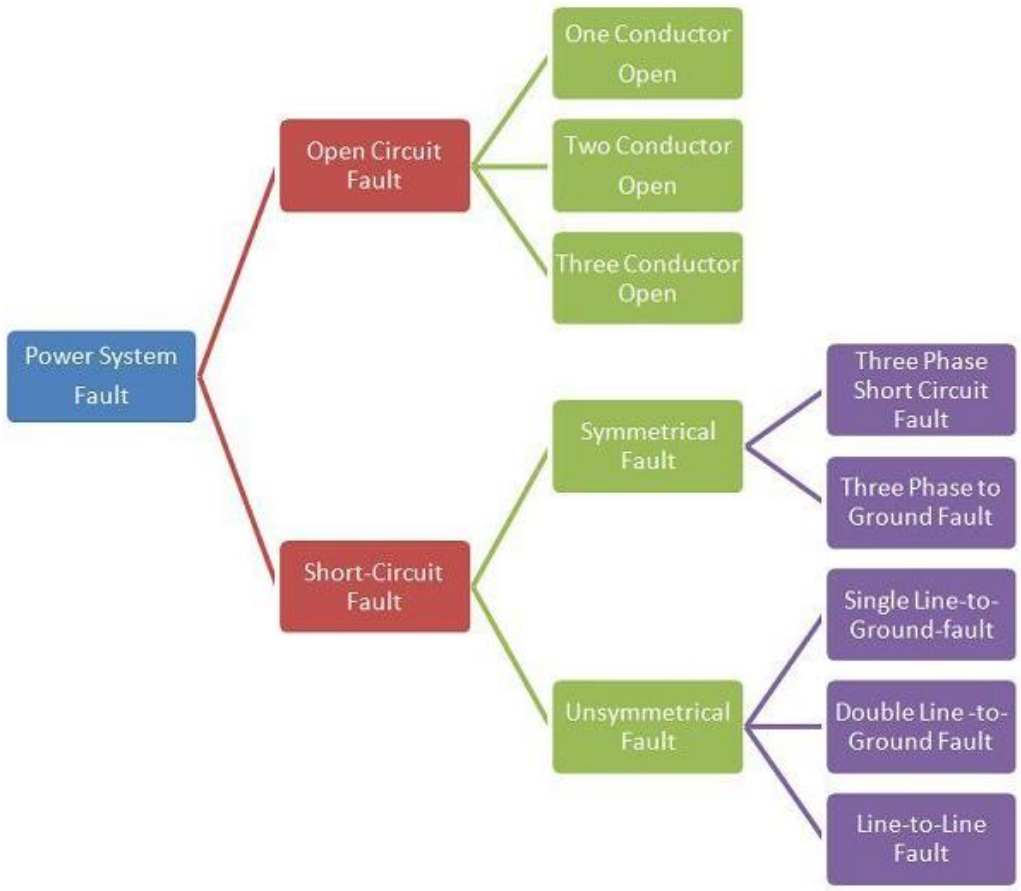
An electrical fault is the deviation of voltages and currents from nominal values or states. Under normal operating conditions, power system equipment or lines carry normal voltages and currents which results in safer operation of the system. But when a fault occurs, voltage and current values deviate from their nominal ranges which cause damage to equipment and devices. The fault inception also involves insulation failures and conducting path failures which results in short circuit and open circuit of conductors.

The faults in the power system cause over current, under voltage, unbalance of the phases, reversed power and high voltage surges. This results in the interruption of the normal operation of the network, failure of equipments, electrical fires, etc. Usually power system networks are protected with switchgear protection equipments such as circuit breakers and relays in order to limit the loss of service due to the electrical failures.

Fault detection and analysis are necessary to select or design suitable switchgear equipment, electromechanical relays, circuit breakers, and other protection devices.

3.1. Types of Faults

The Different types of faults in electrical power systems or lines are:-

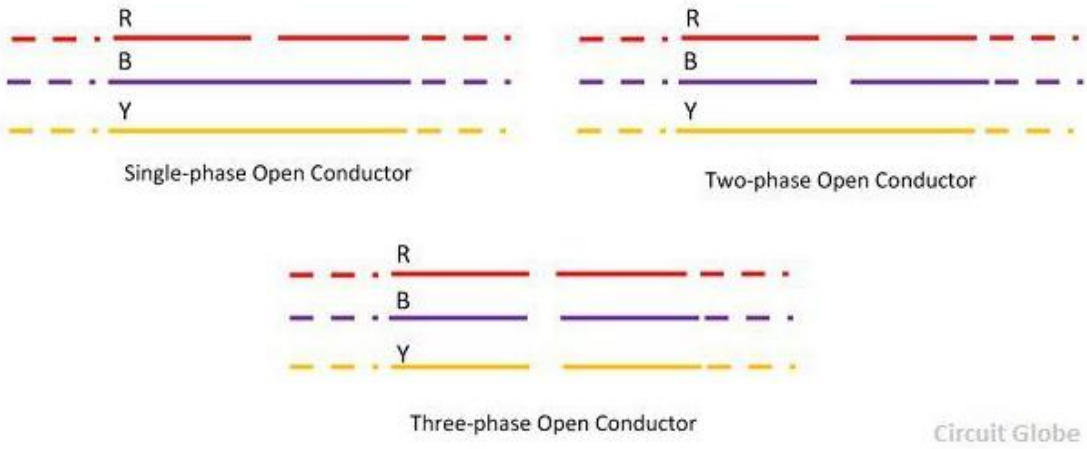


3.1.1. Open Circuit Fault

The open circuit fault mainly occurs because of the failure of one or two conductors. The open circuit fault takes place in series with the line, and because of this, it is also called the series fault. Such types of faults affect the reliability of the system. The open circuit fault is categorized as

- Open Conductor Fault
- Two conductors Open Fault
- Three conductors Open Fault.

The open circuit fault is shown in the figure below.



3.1.2. Short-Circuit Fault

In this type of fault, the conductors of the different phases come into contact with each other with a power line, power transformer or any other circuit element due to which the large current flow in one or two phases of the system. The short-circuit fault is divided into the symmetrical and unsymmetrical fault.

3.1.3. Symmetrical Fault

The faults which involve all the three phases are known as the symmetrical fault. Such types of fault remain balanced even after the fault. The symmetrical faults mainly occur at the terminal of the generators. The fault on the system may arise on account of the resistance of the arc between the conductors or due to the lower footing resistance. The symmetrical fault is sub-categorized into line-to-line-to-line fault and three-phase line-to-ground-fault

3.1.4. Unsymmetrical Faults

The fault gives rise to unsymmetrical current, i.e., current differing in magnitude and phases in the three phases of the power system are known as the unsymmetrical fault. It is also defined as the fault which involves one or two phases such as L- G, L – L, L – L – G fault.

The symmetrical and unsymmetrical fault mainly occurs in the terminal of the generator, and the open circuit and short circuit fault occur on the transmission line.

3.2. Faults in Double circuit transmission line

There will be 20 different faults including all line-line, line-ground, line-line-ground and line-line-line in the double circuit transmission line.

Line -Ground faults:-

A1G, A2G, B1G, B2G, C1G, C2G

Line-Line faults:-

A1B1, B1C1, C1A1, A2B2, B2C2, C2A2

Line-Line-Ground faults:-

A1B1G, B1C1G, C1A1G, A2B2G, B2C2G, C2A2G

Line-Line-Line faults:

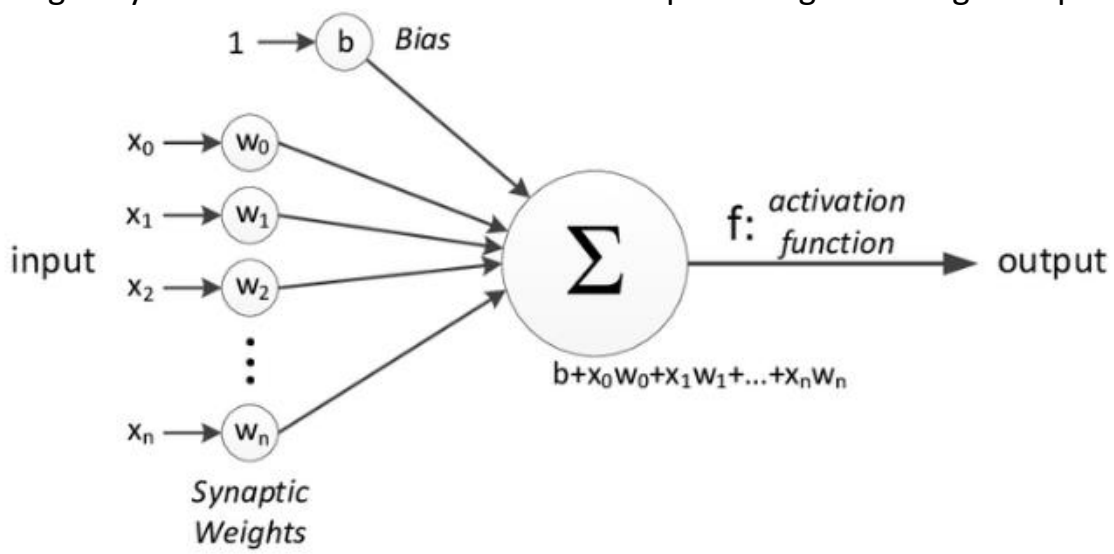
A1B1C1, A2B2C2

4. Artificial Neural Network

4.1. Introduction

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 large number of highly interconnected processing elements (neurons) working in unison to solve a specific problem.

The following diagram represents the general model of ANN which is inspired by a biological neuron. It is also called Perceptron.
A single layer neural network is called a Perceptron. It gives a single output.



Perceptron

In the above figure, for one single observation, x0, x1, x2, x3...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 w0, w1, w2, w3….w(n). **Weight shows the strength of a particular node.**

'b' is a bias value. A bias value allows you to shift the activation function up or down.

In the simplest case, these products are summed, fed to a transfer function (activation function) to generate a result, and this result is sent as output.

Mathematically, x1.w1 + x2.w2 + x3.w3 ..... xn.wn = ∑ xi.wi

Now activation function is applied $\phi(\sum xi.wi)$

- **Activation function**

  The Activation function is important for an ANN to learn and make sense of something really 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.

  Some common examples of activation functions are:

  - Sigmoid function
  - Hyperbolic Tangent function (tanh)
  - Rectified Linear Units — (ReLu)

4.2. How does an Artificial Neural Network learn?

Learning in a neural network is closely related to how we learn in our regular lives and activities — we 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.

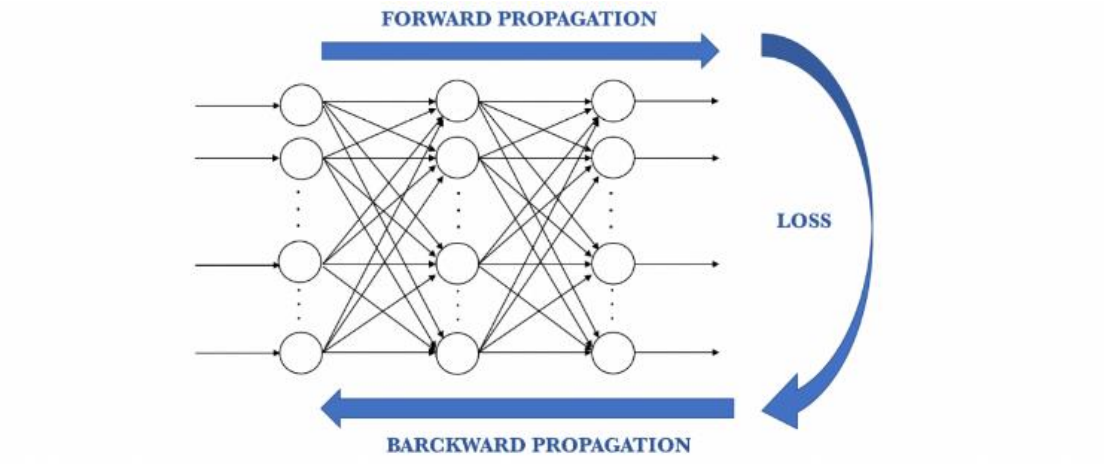
Cost Function: One half of the squared difference between actual and output value.

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

We feed the resulting data back through the entire neural network. The weighted synapses connecting input variables to the neuron are the only thing we have control over.

As long as there exists a disparity between the actual value and the predicted value, we need to adjust those weights. Once we tweak them a little and run the neural network again, a new Cost function will be produced, hopefully, smaller

than the last.
We need to repeat this process until we scrub the cost function down to as small as possible.



The procedure described above is known as **Back-propagation** and is applied continuously through a network until the error value is kept at a minimum.

4.3. Training of an ANN

Step-1 → Randomly initialize the weights to small numbers close to 0 but not 0.

Step-2 → Input the first observation of your dataset in the input layer, each feature in one node.

Step-3 → **Forward-Propagation**: From left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights. Propagate the activations until getting the predicted value.

Step-4 → Compare the predicted result to the actual result and measure the generated error (Cost function).

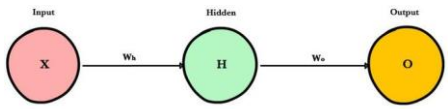
Step-5 → **Back-Propagation**: from right to left, the error is back propagated. Update the weights according to how much they are responsible for the error. The learning rate decides how much we update weights.

Step-6 → Repeat step-1 to 5 and update the weights after each observation.

Step-7 → When the whole training set passed through the ANN, that makes and epoch. Redo more epochs.

- **Forward propagation working**

  Let's start with forward propagation



  Here, input data is "forward propagated" through the network layer by layer to the final layer which outputs a prediction. The simple network can be seen as a series of nested functions.

  For the neural network above, a single pass of forward propagation translates mathematically to:

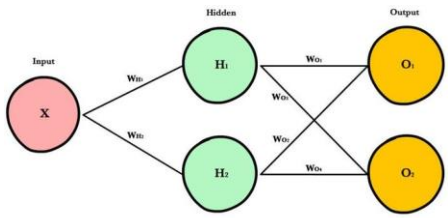
  A (A(X Wh) Wo)

  Where

  A is an activation function like ReLU,

  X is the input

  Wh and Wo are weights for the hidden layer and output layer respectively

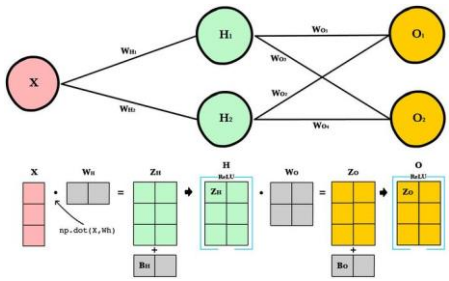


  INPUT_LAYER_SIZE = 1

  HIDDEN_LAYER_SIZE = 2

  OUTPUT_LAYER_SIZE = 2

  In matrix form, this is represented as:

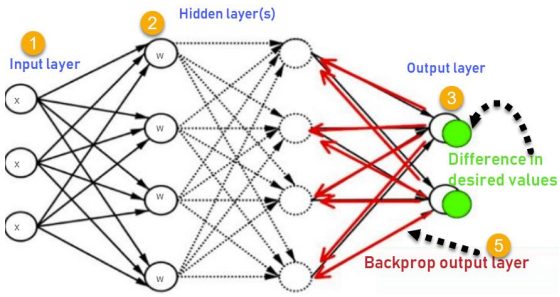


VarName	Dimensions
X Input.	(3, 1)
Wh Hidden weights.	(1, 2)
Bh Hidden bias.	(1, 2)
Zh Hidden weighted input	(1, 2)
H Hidden activations.	(3, 2)
Wo Output weights.	(2, 2)
Bo Output bias.	(1, 2)
Zo Output weighted input	(3, 2)
O Output activations.	(3, 2)

- **Back propagation working**

  The Back propagation algorithm in a neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule.

  Consider the following Back propagation neural network example diagram to understand:



How Backpropagation Algorithm Work

1.Inputs X, arrive through the preconnected path

2.Input is modeled using real weights W. The weights are usually randomly selected.

3.Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.

4.Calculate the error in the outputs

ErrorB= Actual Output – Desired Output

5.Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved.

4.4. Implementation of ANN in Python

```
# Importing libraries

import numpy as np


# Helper functions

def softmax(Z):

    e_power_Z=np.exp(Z)

    return e_power_Z/np.sum(e_power_Z, axis=1, keepdims=True)


tanhDerivative=lambda A: 1-np.square(A) # Derivative of tanh(Z) is 1-
(tanh(Z))^2, here A=tanh(Z)


def loss(Y1hot, Y_1hot): # Cross entropy loss

    l=-np.mean(Y1hot*np.log(Y_1hot))
```

```
    return l

def oneHot(Y, no_of_classes): # Convert label encoding to one hot encoding
    m=Y.shape[0]
    Y1hot=np.zeros((m, no_of_classes))
    Y1hot[np.arange(m), Y]=1
    return Y1hot

class NeuralNetwork:
    def __init__(self, input_size, output_size, hidden_layers, seed=0):
        np.random.seed(seed)
        model={}
        total_layers=len(hidden_layers)+1

        # First layer
        model['W0']=np.random.randn(input_size, hidden_layers[0])
        model['b0']=np.zeros((1, hidden_layers[0]))
        # Intermediate layers
        for i in range(total_layers-2):
            model['W'+str(i+1)]=np.random.randn(hidden_layers[i],
hidden_layers[i+1])
            model['b'+str(i+1)]=np.zeros((1, hidden_layers[i+1]))
        # Last layer
        model['W'+str(total_layers-1)]=np.random.randn(hidden_layers[-1],
output_size)
        model['b'+str(total_layers-1)]=np.zeros((1, output_size))

        self.total_layers=total_layers
        self.model=model
        self.activation_outputs=None
        self.no_output_classes=output_size
        self.no_input_features=input_size

    def forward(self, X):
        W, b={}, {}
        for i in range(self.total_layers):
```

```
            W[i]=self.model['W'+str(i)]
            b[i]=self.model['b'+str(i)]
          Z, A={}, {}
        # First layer
        Z[0]=np.dot(X, W[0])+b[0]
        A[0]=np.tanh(Z[0])
        # Intermediate layers
        for i in range(self.total_layers-2):
            Z[i+1]=np.dot(A[i], W[i+1])+b[i+1]
            A[i+1]=np.tanh(Z[i+1])
        # Last layer
        Z[self.total_layers-1]=np.dot(A[self.total_layers-2],
W[self.total_layers-1])+b[self.total_layers-1]
        Y_1hot=softmax(Z[self.total_layers-1])

        self.activation_outputs=(A, Y_1hot)
        return Y_1hot

    def backward(self, X, Y1hot, learning_rate):
        W, b={}, {}
        for i in range(self.total_layers):
            W[i]=self.model['W'+str(i)]
            b[i]=self.model['b'+str(i)]
        m=X.shape[0]
        A, Y_1hot=self.activation_outputs

        delta, dW, db={}, {}, {}
        # Last layer
        delta[self.total_layers-1]=Y_1hot-Y1hot
        dW[self.total_layers-1]=np.dot(A[self.total_layers-2].T,
delta[self.total_layers-1])
        db[self.total_layers-1]=np.sum(delta[self.total_layers-1], axis=0)
        # Intermediate layers
        for i in range(self.total_layers-2, 0, -1):
            delta[i]=tanhDerivative(A[i])*np.dot(delta[i+1],W[i+1].T)
```

```
            dW[i]=np.dot(A[i-1].T, delta[i])
            db[i]=np.sum(delta[i], axis=0)
        # First layer
        delta[0]=tanhDerivative(A[0])*np.dot(delta[1],W[1].T)
        dW[0]=np.dot(X.T, delta[0])
        db[0]=np.sum(delta[0], axis=0)

        # Update model parameters using Gradient Descent
        for i in range(self.total_layers):
            self.model['W'+str(i)]-=learning_rate*dW[i]
            self.model['b'+str(i)]-=learning_rate*db[i]

    def predict(self, X):
        Y_1hot=self.forward(X)
        Y_=np.argmax(Y_1hot, axis=1)
        return Y_

    def summary(self):
        print("Total layers:", self.total_layers)
        print("Number of input features:", self.no_input_features)
        print("Number of output classes:", self.no_output_classes)
        print("Shapes of weights and biases:")
        total_params=0
        for i in range(self.total_layers):
            print('W'+str(i), self.model['W'+str(i)].shape)
            print('b'+str(i), self.model['b'+str(i)].shape)
            total_params+=self.model['W'+str(i)].shape[0]                    *
self.model['W'+str(i)].shape[1]  +  self.model['b'+str(i)].shape[1]
        print("Total parameters", total_params)

    def accuracy(self, X, Y):
        Y_=self.predict(X)
        return np.sum(Y_==Y)/float(Y.shape[0])

    def  train(self,  X,  Y,  max_epochs=500,  learning_rate=0.001,  logs=True,
validation_set=None):
```

```
no_classes=self.no_output_classes
losses={'training': [], 'validation': None}
accuracy_list={'training': [], 'validation': None}
Y1hot=oneHot(Y, no_classes)
if validation_set is not None:
    Xval=validation_set[0]
    Yval=validation_set[1]
    losses['validation']=[]
    accuracy_list['validation']=[]
    Yval1hot=oneHot(Yval, no_classes)
Y1hot=oneHot(Y, no_classes)
for epoch in range(max_epochs):
    Y_1hot=model.forward(X) #Forward propagation

    # Loss
    l=loss(Y1hot, Y_1hot)
    losses['training'].append(l)
    # Accuracy
    acc=self.accuracy(X, Y)
    accuracy_list['training'].append(acc)

    model.backward(X, Y1hot, learning_rate) # Backward propagation

    # Validation loss
    if validation_set is not None:
        Y_val1hot=model.forward(Xval) #Forward propagation
        val_l=loss(Yval1hot, Y_val1hot)
        losses['validation'].append(val_l)
        val_acc=self.accuracy(Xval, Yval)
        accuracy_list['validation'].append(val_acc)
    if logs and epoch%10==0:
        print("Epoch: %d Training Loss: %.4f"%(epoch, l))
return losses, accuracy_list
```

5. Fault classification and location detection using ANN

5.1. Selecting Input and output of ANN

One factor in determining the right size and architecture for the neural network is the number of inputs and outputs that it must have. The lower the number of inputs, the smaller the network can be. However, sufficient input data to characterize the problem must be ensured. The signals recorded at one end of the line only are used. The inputs to conventional distance relays are mainly the voltages and currents. Hence the network inputs chosen here are the magnitudes of the fundamental components (50 Hz) of three-phase voltages and three-phase currents of each circuit, that is, six currents measured at the relay location. As the basic task of fault location is to determine the distance to the fault, fault distance location, in km (Lf) with regard to the total length of the line, is the only output provided by the fault location network.

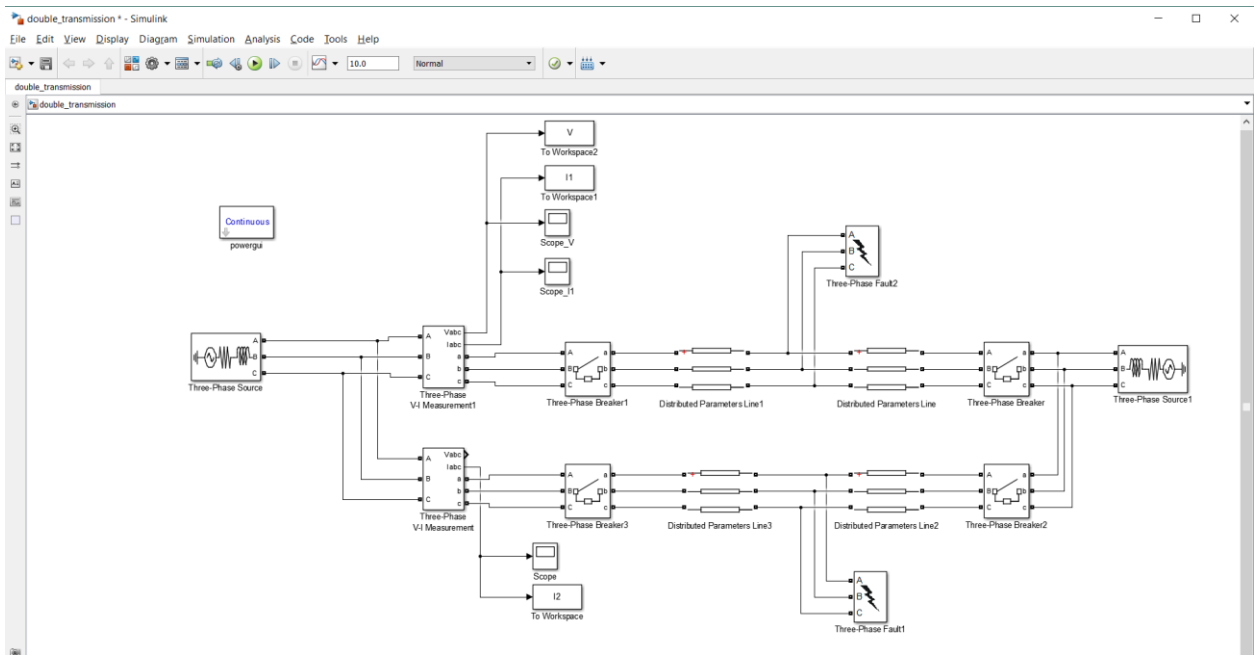
Thus, the inputs X and the outputs Y for the ANN are given by:

$$X = \left[V_{af}, V_{bf}, V_{cf}, I_{a1f}, I_{b1f}, I_{c1f}, I_{a2f}, I_{b2f}, I_{c2f}\right],$$

$$Y = \left[L_f\right].$$

5.2. Data Collection

5.2.1. Simulink Modeling



The simulation of fault in a Double Circuit Transmission Line is created using Simulink and Simscape. At the both ends, there are 'Three-Phase Source' blocks. The transmission line is divided into two parts with 'Three-Phase Fault' blocks in the middle of both, thus there are four 'Distributed Parameters Line' blocks required to separate them. With each 'Distributed Parameters Line' block there is a 'Three Phase Breaker' block for the safety of transmission lines. On one end of the transmission line there are two 'Three-Phase V-I Measurement' blocks for each circuit. These are used to measure the instantaneous voltage and current and form a graph. This graph can be seen in the simulation with the help of 'Scope' block. To send this as Timeseries data structure to the Matlab workspace, 'To Workspace' blocks are used.

The peak voltages for both of the 'Three-Phase Source' blocks are set as 220kV and the frequency is 50Hz. The values of Line Length parameter of 'Distributed Parameters Line' blocks on the right is set as 100 km. For the blocks on the left, it is set as a variable Lf. This value is set from the workspace (or Matlab script in this case). Similarly, for both of the 'Three-Phase Fault' blocks the value of parameters Fault Resistance is set as variable 'Rf' and Inception Angle as 'Ia'. These values are also given to the simulation from the Matlab script. The values of other parameters for different block models are shown below.

Double circuit transmission line parameters:

Parameters	Set Value
Positive sequence resistance R_1,ohm/km	0.01809
Zero sequence resistance R_0 ,ohm/km	0.2188
Zero sequence mutual resistance R_{0m},ohm/km	0.20052
Positive sequence inductance L_1,H/km	0.00092974
Zero sequence inductance L_0,H/km	0.0032829
Zero sequence mutual inductance L_{0m},H/km	0.0020802
Positive sequence capacitance C_1,F/km	1.2571e - 008
Zero sequence capacitance C_0,F/km	7.855e - 009
Zero sequence mutual capacitance C_{0m},F/km	-2.0444e - 009

A 220 kV double-circuit transmission line of line length100 km which is fed from sources at each end is simulated using Matlab/Simulink and Simscape toolbox. The transmission line is simulated using a distributed parameter line model using the power line parameter of the Simscape toolbox of Matlab software. Using the Simulink and Simscape toolbox of Matlab all the ten types of faults at different fault locations between 0 and 100% of line length and fault inception angles 0 and 90 degree have been simulated. The total number of ground faults simulated is $12*10*2*3=720$ and phase faults $8*10*2 = 160$; thus total fault cases are 880, and from each fault cases10 number of post fault samples have been extracted, also 35 no fault samples are taken to form the training data set for neural network. Thus the total number of patterns generated for training is 8800 + 35 =8835. Training matrices were built in such a way that the network trained produces an output corresponding to the fault distance location.

The Fault in line starts from t=20s to t=30s that is for 20 seconds of simulation period. Data acquisition was done in the time of fault simulation. Sample size is taken 1000 as MATLAB only has max. Size for array 10000, the sample size couldn't be larger. That is for each instance simulated 10000 data points were generated.

Parameters which were used for generating data and fault using MATLAB simulation:-

Parameter	Set value
Fault type	LG: A1N, A2N, B1N, B2N, C1N, and C2N LL: A1B1, A2B2, B1C1, B2C2, A1C1, and A2C2 LLG: A1B1N, A2B2N, B1C1N, B2C2N, A1C1N, and A2C2N LLL: A1B1C1, A2B2C2
Fault location (L_f in KM)	1, 10, 20, 30, . . ., 80, and 90 KM
Fault inception angle (Φ_i)	0° and 90°
Fault resistance (R_f)	0, 50 and 100 Ω
Prefault power flow angle (δ_s)	45°

'to workspace' block has been used to direct the data into Matlab workspace directly.

5.2.2. Matlab code

Each simulation generates 1 data point and the total number of data points required are 8835. This is not humanly possible to execute this many simulations so instead the simulations are executed using a Matlab code with varying values of Fault Resistance(Rf), Fault Inception Angle(Ia) and Fault Location(Lf). With the help of nested for loops and 'sim()' function to run simulations and get the data. The code for the same is given below:

```
% Script to generate the required data

clear
clc
close all

% Type of fault
a1b1c1a2b2c3n=[1 1 1 0 0 0 0];

% Defining the variables to be changed the simulink model
Rf_values=[0 50 100]; % Fault resistance (ohm)
Lf_values=[1 10 20 30 40 50 60 70 80 90]; % Fault location (km)
Ia_values=[0 90]; % Fault inception angle (degrees)
```

```
for i=1:length(Rf_values)
    for j=1:length(Lf_values)
        for k=1:length(Ia_values)
            Rf=Rf_values(i);
            Lf=Lf_values(j);
            Ia=Ia_values(k);
            % Simulating the model in simulink model
            sim('double_transmission.slx');
            % Extracting the data generated by simulink model
            % V is a 'To workspace' block (using a Timeseries structure)
            t=V.Time;
            Va=V.Data(:, 1);
            Vb=V.Data(:, 2);
            Vc=V.Data(:, 3);
            % I1 is a 'To workspace' block (using a Timeseries structure)
            I1a=I1.Data(:, 1);
            I1b=I1.Data(:, 2);
            I1c=I1.Data(:, 3);
            % I2 is a 'To workspace' block (using a Timeseries    structure)
            I2a=I2.Data(:, 1);
            I2b=I2.Data(:, 2);
            I2c=I2.Data(:, 3);




            % Applying anti-aliasing filter {anti-alising: aa=abs(fi-Nfs)}
            Va=abs(Va-lenght(Va));
            Vb=abs(Vb-lenght(Vb));
            Vc=abs(Vc-lenght(Vc));
            I1a=abs(I1a-lenght(I1a));
            I1b=abs(I1b-lenght(I1b));
            I1c=abs(I1c-lenght(I1c));
            I2a=abs(I2a-lenght(I2a));
            I2b=abs(I2b-lenght(I2b));
            I2c=abs(I2c-lenght(I2c));



            % Applying FFT
            Vfa=fft(Va);
            Vfb=fft(Vb);
            Vfc=fft(Vc);
            I1fa=fft(I1a);
            I1fb=fft(I1b);
            I1fc=fft(I1c);
            I2fa=fft(I2a);
            I2fb=fft(I2b);
            I2fc=fft(I2c);
```

```
        % Calculating Fundamental (peak) component of fft
        [Vfpa, ~]=max(Vfa);
        [Vfpb, ~]=max(Vfb);
        [Vfpc, ~]=max(Vfc);
        [I1fpa, ~]=max(I1fa);
        [I1fpb, ~]=max(I1fb);
        [I1fpc, ~]=max(I1fc);
        [I2fpa, ~]=max(I2fa);
        [I2fpb, ~]=max(I2fb);
        [I2fpc, ~]=max(I2fc);

        % Making a row for data to be written in excel
        % Attribute order: a1 b1 c1 a2 b2 c3 n Vfpa Vfpb Vfpc I1fpa I1fpb I1fpc I2fpa I2fpb I2fpc Lf
        if(aa1b1c1a2b2c3n==[0 0 0 0 0 0 0]) % If there is no fault location is given as -1
          data=[a1b1c1a2b2c3n Vfpa Vfpb Vfpc I1fpa I1fpb I1fpc I2fpa I2fpb I2fpc -1];
        else
          data=[a1b1c1a2b2c3n Vfpa Vfpb Vfpc I1fpa I1fpb I1fpc I2fpa I2fpb I2fpc Lf];
        end

        % Saving the data to
        xlswrite('generated_data', data, 'append');
      end
    end
end
```

As discussed, the values of A1, B1, C1, A2, B2, C2 and N are manually changed. These values are not stored in seven separate variables but in a single array ‘a1b1c1a2b2c2n’ which stores the seven values in respective order. All the possible values for the other three variables Fault Resistance(Rf), Fault Inception Angle(Ia) and Fault Location(Lf) are stored as ‘Rf_values’, ‘Ia_values’ and ‘Lf_values’ respectively. All of these are iterated upon using nested looping to get all the possible values of each. Then for each value simulation is run using the ‘sim()’ function. The ‘To Workspace’ blocks return the data in form of Timeseries data structure. There are three such ‘To Workspace’ blocks V, I1 and I2. The data in each contains 3 vectors of lines A, B and C respectively and the timestamps. For example, for V, these three vectors are Va, Vb and Vc. Thus three are in total 9 of such vectors, namely, ’Va’, ’Vb’, ’Vc’, ’I1a’, ’I1b’, ’I1c’, ’I2a’, ’I2b’ and ’I2c’. All of these 9 values are then preprossed, first using the anti-aliasing filter and then FFT. Finally the peak value of FFT is found using the ‘max()’ function which converts these 9 vectors into 9 values saved as variables ’Vfpa’, ’Vfpb’, ’Vfpc’, ’I1fpa’, ’I1fpb’, ’I1fpc’, I2fpa, I2fpb and I2fpc’.

After calculating these 9 values for each simulation, these values are now ready to become an input datapoint to the neural networks. But before that, these need to be structurally made into an Excel sheet. For each iteration, an array called ‘data’ is made. The array ‘data’ represents a row in the Excel sheet. The first seven columns of ‘data’ are the values of the array ‘a1b1c1a2b2c2n’, the next nine values are ’Va’, ’Vb’, ’Vc’, ’I1a’, ’I1b’, ’I1c’, ’I2a’, ’I2b’ and ’I2c’ and the last value is the value of ‘Lf’. With the help of ‘xlswrite()’ function, in append mode, the data is appended to the Excel sheet named ‘generated_data.xls’ and it is uploaded to Google Drive.

5.3. Data Preprocessing

As each instance produces a signal which gives enormous amounts of data (1000*10 data points) so each one of 8835 instances is being preprocessed to generate one single observation of 9 attributes (9 inputs). Also to train the network, a suitable number of representative examples of the relevant phenomenon must be selected, so that the network can learn the fundamental characteristics of the problem. The steps involved in data preprocessing, all the data exported from simulation then goes through these steps. Three-phase voltages and three-phase current signals of both the circuits obtained through Matlab simulation are sampled at a sampling frequency of 1 kHz. Then one full cycle discrete Fourier transform is used to calculate the fundamental component of three-phase voltages and currents of both circuits which are used as input to the ANN. The input signals have to be normalized in order to reach the ANN input level (±1).

Each 8835 instance is being preprocessed to generate one single observation of 9 attributes (9 inputs) as each instance produces a signal which gives numerous amounts of data.

5.3.1. Anti-Aliasing filter

Aliasing is an undesired effect in which the sampling frequency is too low/high to accurately reproduce the original analog content, resulting in signal distortion. Frequency aliasing is a common problem in signal conversion systems whose sampling rate is too slow/fast to read input signals of a much higher frequency.

Hence, Anti-Aliasing filter is used to match or at least reduce the gap between the sample rate of the data acquisition system to the frequency of the signal. If the sample rate of the data acquisition system is too slow relative to the frequency of the signal, the measurement literally falls apart.

In this project we have created a user defined function using the basic mathematical formula of aliasing.

Basic mathematical formula:-

fa(N)	= *|fin - Nfs|*
fa	= alias frequency
fin	= input signal frequency
fs	= sample rate
N	= an integer greater than or equal to 0

5.3.2. Discrete Fourier transformation

The discrete Fourier transform (DFT), implemented by one of the computationally efficient fast Fourier transform (FFT) algorithms, has become the core of many digital signal processing systems. These systems can perform general time domain signal processing and classical frequency domain processing. Its fast computation is considered as an advantage. With this mathematical tool, it is possible to have an estimation of the fundamental amplitude and its harmonics with a reasonable approximation.

The data is processed in MATLAB itself. The *fft* function is being used to process the data and calculate the magnitude of the fundamental component

of all the 9 inputs (3- voltages and 6- current).This fundamental component of all the 9 values will serve as the input data to the ANN models.

The script for DFT using FFT is already mentioned in MATLAB script.

5.3.3. Normalization

Normalizing data generally speeds up the learning and leads to faster convergence. The input signals have to be normalized in order to reach the ANN input level (±1).Initially the input data is normalised with the help of Normalizer provided in scikit-learn library.Data is automatically normalised in the hidden layer with the help of activation function (tanh).After the normalization the data can be directly fed to the ANN models.

```
from sklearn import preprocessing
scaler=preprocessing.StandardScaler().fit(input_data)
input_data=scaler.transform(input_data)
```

The ‘preprocessing.StandardScaler()’ class of sklearn is used to an object ‘scaler’ by fitting the data points of input data. Then the input data is scaled by using the ‘transform()’ method of the ‘scalar’ object. ‘StandardScaler()’ scales the data to make it a normal distribution, thus making the mean of the data as 0 and the standard deviation as 1.

5.3.4. Loading the Data in Python

```
import pandas as pd
import numpy as np
from google.colab import drive
drive.mount('/content/drive')
```

```
df=pd.read_excel("drive/My Drive/generated_data.xls")
df.head()
```

The Excel sheet 'generated_data.xls', which contains the data is uploaded to the Google drive and to access this file in the code, 'google.colab.drive' is used. The 'mount()' method is used for mounting the drive before reading the Excel sheet using the 'read_excel()' method from 'pandas' library.

```
lables=df.values[:, :7]
input_data=df.values[:, 7: 16]
target=df.values[:, -1]
```

As aforementioned, the sheet contains the first 7 columns as the values of A1, B1, C1, A2, B2, C2 and N. These are saved as an array of shape = (7, 8835) called 'labels'. The next 9 values are the input to the neural network namely Vfa, Vfb, Vfc, I1fa, I1fb, I1fc, I2fa, I2fb and I2fc. These are saved as 'input_data', an array of shape=(9, 8835). The last column is the target value, Lf, saved as 'target', an array of shape = (8835,_).

5.4. ANN Model

Two different models have been developed for Fault classification and distance locator. One is Single ANN based FDL and the other is Modular where first ANN classifies the fault and next there are 4 different ANN based on type of fault this model is Modular ANN based FDL.

5.4.1. Single ANN based FDL

```
from keras.models import Sequential
from keras.layers import Dense, Activation
optm='adam' # Adam is used for now instead of Levenberg Marquardt optimizer

model_single=Sequential()
model_single.add(Dense(40, input_shape=(9, )))
model_single.add(Activation('tanh'))
model_single.add(Dense(1))
model_single.add(Activation('relu'))
model_single.compile(loss='mean_squared_error', optimizer=optm, metrics=['loss'])
model_single.summary()
```

The python library used for making the Neural Network architecture is ‘keras’. Keras api provides two types of models: Sequential and Functional. Both of their classes are present in ‘keras.models’. ‘Sequential()’ class is used for sequential models and ‘Model()’ class for functional models. ‘Sequential()’ is used for both of the neural network architectures used in the project. The optimizer used for both of the models is Adam.

Adam: The paper which we followed for the project used a Levenberg Marquardt optimizer for achieving convergence but in the project we have used Adam optimizer instead. Adam is . Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications.

To the object of ‘Sequential()’ called ‘model_single’ the first layer of vanilla neural network is added called a Dense layer (or Fully Connected Layer). The number of neurons is 40 and the input shape is (9,). The activation of this layer is tanh function. The next layer is the output layer with one neuron and a linear function ReLU as its activation.

The value of the output neuron is the predicted distance. Since the predicted distance is a continuous value, the model is regressor type and so the loss function is taken as MSE (Mean Squared Error) and the evaluation metric is loss.

5.4.2. Modular ANN based FDL

Function to find the type of the fault based on label values:

```
def fault_type(A1, B1, C1, A2, B2, C2, N):
  if A1==0 and A2==0 and B1==0 and B2==0 and C1==0 and C2==0 and N==0:
    return 'nofault'
  if N:
    if (A1 and B1) or (A2 and B2) or (B1 and C1) or (B2 and C2) or (C1 and A1) or (C2 and A2):
      return 'doublephase2ground'
    if A1 or A2 or B1 or B2 or C1 or C2:
      return 'phase2ground'
  else:
    if (A1 and B1 and C1) or (A2 and B2 and C2):
      return 'threephase'
    if (A1 and B1) or (A2 and B2) or (B1 and C1) or (B2 and C2) or (C1 and A1) or (C2 and A2):
      return 'phase2phase'
```

The function ‘fault_type()’ is used in many parts of the code for Modular ANN type FDL. This function takes in as input the values of A1, B1, C1, A2, B2, C2 and N and it returns as output the type of fault based on these values. Firstly, the fault is categorised as “nofault” if all of these 7 values are 0. Then the value of N is checked to classify the fault as ground or no-ground fault. If the fault is of ground type and any two of the three values i.e., A, B and C have value as 1, then the fault is “doublephase2ground” if any not two but one of the three has value as 1, then the fault is “phase2ground”. Similarly for the other case where N is 0, if all three of A, B and C are 1, the fault is “threephase”, but if not all three but any two of the three has values as 1, then the fault is “phase2phase”.

- **Model Architecture:**

The architecture of the second neural network consists of two different parts. The first part consists of a classifier which is used to classify the fault as one of either “nofault”, “doublephase2ground”, “phase2ground”, “threephase” or “phase2phase” based on the 9 input values given. The second part consists of four regressor type ANNs each specialised for a different kind of fault and each with a different number of neurons in the hidden layer. Each of these takes in 9 inputs and returns one output which is the predicted distance of the fault. The first part that is the classifier decides which of the four regressors will get the inputs and return the output.

There are two ways to make such a Neural Network architecture in ‘keras’. It can be made as four separate ANNs with straight-forward architecture using the ‘Sequential()’ class or as a single ANN containing different branches using the ‘Model()’ class. In the project, the prior method is used.

a) Classifier:

```
model_classifier=Sequential()
model_classifier.add(Dense(10, input_shape=(9, )))
model_classifier.add(Activation('tanh'))
model_classifier.add(Dense(7))
model_classifier.add(Activation('sigmoid'))
model_classifier.compile(loss='categorical_crossentropy', optimizer=optm, metrics=['accuracy'])
model_classifier.summary()
```

As discussed, ‘model_classifier’ an object of ‘Sequential()’ is used to create the model. First layer is a Dense layer with 10 neurons and the input shape is (9,). The activation of this layer is tanh function. The output layer contains seven neurons and sigmoid as its activation function which returns a value in range 0 to 1. The predicted values are the values of A1, B1, C1, A2, B2, C2 and N. Since the predicted values are discrete (either 0 or 1) , the model is classifier type and so the loss function is taken as Categorical Cross Entropy and the evaluation metric is accuracy.

b) Regressors:

```
model_modular={}
model_types=['phase2ground', 'phase2phase', 'doublephase2ground', 'threephase']
hidden_layer_neurons={
    'phase2ground':30,
    'phase2phase':30,
    'doublephase2ground':20,
    'threephase':5
}
for model_type in model_types:
  model_modular[model_type]=Sequential()
  model_modular[model_type].add(Dense(hidden_layer_neurons[model_type], input_shape=(9, )))
  model_modular[model_type].add(Activation('tanh'))
  model_modular[model_type].add(Dense(1))
  model_modular[model_type].add(Activation('relu'))
  model_modular[model_type].compile(loss='mean_squared_error', optimizer=optm, metrics=['loss'])
  model_modular[model_type].summary()
```

The classifier model is followed by four different regressor models. Each of the regressor model is an object stored in a dictionary called ‘model_modular’ whose keys are the names of different faults, namely, “doublephase2ground”, “phase2ground”, “threephase” and “phase2phase”. The number of neurons in the hidden layer of each of these models is saved in another dictionary ‘hidden_layer_neurons’ with the same keys as that of ‘model_modular’. The number of neurons for “doublephase2ground”, “phase2ground”, “threephase” and “phase2phase” are 20, 30, 5 and 30 respectively. The rest of the architecture is the same as Single ANN type FDL.

Custom function to train the modular ANN:

```
def modular_train(input_data, output_data, labels, epoch_c=300, epoch_r=300, batch=10, val_split=0.2):
  # 1. Training classifier

  hist_c=model_classifier.fit(input_data, labels, epochs=epoch_c, batch_size=batch, validation_split=val_split)

  # 2. Training regressor

  # Seperating the data according to type of fault
  for model_type in model_types:
    labelled_input_data[model_type]=[]
    labelled_output_data[model_type]=[]
  for i in range(input_data.shape[0]):
    fault=fault_type(*labels[i]) # Finding the type of data
    if fault=='nofault':
      continue
    labelled_input_data[fault].append(input_data[i])
    labelled_output_data[fault].append(output_data[i])
  for model_type in model_types:
    labelled_input_data[model_type]=np.array(labelled_input_data)
    labelled_output_data[model_type]=np.array(labelled_output_data)
  hist_r={}
  for model_type in model_types:
    hist_r[model_type]=model_modular[model_type].fit(labelled_input_data[model_type],
                                                     labelled_output_data[model_type],
                                                     epochs=epoch_r,
                                                     batch_size=batch,
                                                     validation_split=val_split)
  return hist_c, hist_r
```

The training of the modular ANN is done by training each of the 5 ANN models separately on their respective data. The function ‘modular_train()’ is defined and then used for this task. For the first part of the function, which is training the classifier called ‘model_classifier’, the model is directly fed with the input data stored as ‘input_data’ and the output consists of 7 values A1, B1, C1, A2, B2, C2 and N stored as ‘labels’. These seven values can then be used to make the classification of the fault as one of the 5 types (including No fault) with the help of ‘fault_type()’ function already defined above. For this purpose, the ‘fit()’ method of ‘model_classifier()’ is used. Along with the input and output data, the number of epochs/iterations, batch size and the validation split is provided to the ‘fit()’ method as given by the user. Their default values are 300, 10 and 0.2 respectively. The values of losses generated by training and validation data for each iteration is returned by the ‘fit()’ method as a dictionary which is saved in the variable ‘hist_c’.

For the second part, which is training the regressor, first the data need to be categorised fault type wise and then the data for each of the faults is fed to the ANN model of the respective fault. For that, for each fault type a dictionary is made to store the data-points. There are two such dictionaries each for input and output data, namely ‘labelled_input_data’ and ‘labelled_output_data’. The keys of these dictionaries are the same as that of ‘model_modular’ i.e., fault names. Initially the value for each key is an

empty list for both the dictionaries. Then the data points are iterated upon and using the 'fault_type()', the function defined above, the type of fault is found out. This function takes in input the current value of 'labels' and returns the type of fault as a string which is saved in the variable 'fault'. If the current data point is of fault type "nofault", then that point is just left out and the iteration over remaining data points continues. If not, then the input and the output data is saved in the 'labelled_input_data' and 'labelled_output_data" respectively, according to the fault type. Then, the list of data for each fault type is converted to arrays of data points, since in Keras all training is done on data points stored as array data structure and not list data structure. Then the segregated data is simply fed to their respective regressor model and each model is trained using the 'fit()' method. Similar to the case of classifier, along with the input and output data, the number of epochs/iterations, batch size and the validation split is provided to the 'fit()' method for each model as given by the user. Their default values are 300, 10 and 0.2 respectively for each of the four regressor models. The losses for each iteration for each model are also as a dictionary called 'hist_r'.

After both the models the losses 'hist_c' and 'hist_r' are returned by the function.

Custom function to predict the output of the modular ANN:

```
def modular_predict(data):
 # 1. Predicting the class

 classes_pred=[1 if x>=0.5 else 0 for x in model_classifier.predict(data)]

 # 2. Predicting the distance

 fault=fault_type(*classes_pred)
 if fault=='nofault':
   return -1
 pred_distance=model_modular[fault].predict(data)
 return pred_distance
```

Unlike the training part in which each model is trained separately and has to do nothing with each other, in this part all of the 5 ANN models are combined together to work together as a single unit and predict the fault distance for a given input datapoint. Inside the 'modular_predict' function, the first task is to predict the type of fault using the classifier. This is done with the help of the 'predict()' method of the 'model_classifier' object. It returns a list of probabilities of values of A1, B1, C1, A2, B2, C2 and N as a list. Therefore, we have a list of 7 float values each in range 0 to 1, exclusive. The values close to 1 are predicted as 1 and those close to 0 as 0. This is done with the help of list comprehension in python.

Inside the list comprehension, the values which are greater than or equal to 0.5 are set as 1 and those which are less than 0.5 are set as 0 and so it returns a list of 0s and 1s. This list is saved as ‘classes_pred’.

In the second part, the required value of the fault distance is predicted. For this, first the function ‘fault_type()’ defined earlier is used to get the type of fault as a string with the help of the 7 values. This string is saved in the variable ‘fault’. If the value of fault is “nofault”, then we just return -1 as the distance. If not, then ‘predict()’ method of the respective model is used to predict the fault location of that specific type of fault. The predicted distance, stored in variable ‘pred_distance’ is then returned by the function.

5.5. Model Training and Performance

Single ANN based FDL

```
hist=model_single.fit(input_data, target, epochs=300, batch_size=10, validation_split=0.2)

# Final loss (validation)
final_loss_single=hist['val_loss'][-1]
final_loss_single
```

To train the single ANN model, simply 'fit()' method of object 'model_single' is used, with number of iterations as 300, batch size as 10 and validation split as 0.2.

Then the final value of validation loss is saved in a variable called 'final_loss_single'.

Modular ANN based FDL

```
hist_c, hist_r=modular_train(input_data, target, labels)

# Final loss (validation)
final_losses_modular={}
for model_type in model_types:
  final_losses_modular[model_type]=hist_r[model_type]['val_loss'][-1]
  print(final_losses_modular[model_type])
```

To train the modular ANN model, the function 'modular_train()' defined above is called. The number of iterations, batch size and validation split are set as default values.

The final value of validation loss for each model is saved in a dictionary called 'final_losses_modular'.

6. Result and Analysis

Single ANN based FDL was tested for different architectures for similar data for better performance, as it can seen at 40 number of hidden neuron, the error is lowest.

S.no	Number of hidden neurons	Number of epochs	Mean square error
1	20	300	0.00182571
2	25	300	0.00070987
3	30	300	0.000598021
4	35	300	0.000492030
5	40	300	0.000381971

Architecture of modular ANN-based fault distance locator.

	Modular ANN-based Fault distance locators	Architecture	Mean square error (MSE)
1	Phase to ground	9-30-1	4.30680e-04
2	Phase to Phase	9-30-1	2.0189e-04
3	Double phase to ground	9-20-1	2.4078e-04
4	Three Phase	9-5-1	7.92508e-05

Test results of single and modular ANN-based FDL

Fault type	Fault inception angle Φ (°)	Fault resistance R_f (Ω)	Fault location L_f (km)	Output of single ANN-based FDL L_e (km)	Output of modular ANN-based FDL L_e (km)	% Error of single ANN-based FDL $e = ((L_f - L_e)/L_f) \times 100\%$	% Error of modular ANN-based FDL $e = ((L_f - L_e)/L_f) \times 100\%$
A1N	45	80	67	64.470	66.95	2.53	0.05
A2N	90	90	77	76.447	57.111	0.553	−0.111
B1N	135	0	5	4.9724	4.8895	0.0276	0.1105
B2N	270	80	89	88.624	88.681	0.376	0.319
C1N	360	95	95	92.405	94.604	2.595	0.396
C2N	180	70	38	35.449	38.385	2.551	−0.385
A1B1	270	—	83	83.020	83.25	−0.02	−0.25
A2B2	0	—	15	15.066	15.009	−0.066	−0.009
B1C1	0	—	76	75.049	75.894	0.951	0.106
B2C2	135	—	90	89.490	89.977	0.51	0.023
C1A1	90	—	22	22.144	22.134	−0.144	−0.134
C2A2	225	—	59	58.762	58.984	0.238	0.016
A1B1N	135	30	85	82.955	83.799	2.045	1.201
A2B2N	45	60	57	56.459	56.047	0.541	0.953
B1C1N	225	100	88	88.246	88.1006	−0.246	−0.1006
B2C2N	270	80	89	88.632	90.133	0.368	−1.133
C1A1N	225	30	58	50.838	57.047	7.162	0.953
C2A2N	90	40	24	20.105	25.362	3.895	−1.362
A1B1C1	225	—	33	33.757	32.952	−0.757	0.048
A2B2C2	360	—	85	86.973	85.136	−1.973	−0.136

7. Conclusion

As expected we have found that modular ANN based Fault distance locator is more efficient than single ANN based FDL. The accuracies of both of the models are fitting such that these can be used for different data as well. The test results of single and modular ANN-based FDLs have been shown under variety of the fault situations, namely, LG faults (A1N, A2N, B1N, B2N, C1N, and C2N), LL faults (A1B1, A2B2, B1C1, B2C2, C1A1, and C2A2), LLG faults (A1B1N, A2B2N, B1C1N, B2C2N, C1A1N, and C2A1N), and LLL faults (A1B1C1 and A2B2C2).The ANN based fault locators calculates the fault distance up to 0– 90% of the line length with high accuracy and enhances the performance of distance relaying scheme by increasing its reach setting.

8. Future Scope

- For future work, the data preprocessing could add more elements like some filters to reduce higher order harmonics making the signal noise less.
- For data generation, the variables in project are Fault Distance, Inception Angle, Fault Resistance, and type of faults. The number of variables can be increased for data generation. This will make the data much more dynamic.
- Advanced neural networks layers can be used to make a hybrid DNN (Deep Neural Network). These layers can be CNN (Convolutional Neural Network), RNN (Recurrent Neural Network)/ LSTM (Long Short Term Memory) etc. A highly advanced neural network wouldn't even need the preprocessing and can generate much more accurate results from the raw data.

9. References

- M. Agrasar, F. Uriondo, and J. R. Hern´andez, “Evaluation of uncertainties in double line distance relaying. A global sight,” IEEE Transactions on Power Delivery.
- R. K. Aggarwal, Q. Y. Xuan, R. W. Dunn, A. T. Johns, and A. Bennett, “A novel fault classification technique for double-circuit lines based on a combined unsupervised/supervised neural network,” IEEE Transactions on Power Delivery,vol.14,no. 4, pp. 1250–1256, 1999.
- L. S. Martins, J. F. Martins, V. F. Pires, and C. M. Alegria, “A neural space vector fault location for parallel double-circuit distribution lines,” International Journal of Electrical Power and Energy Systems, vol.27, no.3, pp.225–231, 2005.
- www.circuitglobe.com
- A. Jain, A. S. Thoke, and R. N. Patel, “Double circuit transmission line fault distance location using artificial neural network,” in Proceedings of the World Congress on Nature and Biologically Inspired Computing (NABIC’09), pp. 13–18, Coimbatore, India, December 2009.