DEPARTMENT OF MECHATRONICS ENGINEERING

MINI PROJECT

MCTA 3371 COMPUTATIONAL INTELLIGENCE

MCTE 4322 INTELLIGENT CONTROL

SECTION 2

SEMESTER I, 2023/2024

INTELLIGENT HEART RISK PREDICTION USING ARTIFICIAL NEURAL NETWORK AND GENETIC ALGORITHM

Dr. Azhar Bin Mohd Ibrahim
Dr. Hasan Firdaus Bin Mohd Zaki

15th February 2024

Group 4

Tashfin Muqtasid	2119609
Norhezry Hakimie bin Noor Fahmy	2110061
Adli Hakim bin Zulkifli	2110959
Muhammad Nazim Bin Akhmar	2114551

Table of Contents

1. Introduction

Cardiovascular diseases, primarily heart disease, remain a leading cause of morbidity and mortality worldwide. Early identification of individuals at high risk of developing heart disease is crucial for preventative interventions and improved patient outcomes. Traditional risk assessment methods, while valuable, have limitations in capturing the complex, nonlinear relationships among risk factors. This project explores the potential of artificial neural networks (ANNs), among other computational intelligence methods, to enhance the prediction of heart disease risk, offering a powerful tool to aid in proactive healthcare strategies.

The primary objective of this project was to develop an ANN-based model for predicting heart disease risk. By leveraging a dataset containing various patient attributes and medical indicators, we aimed to construct a predictive model that could identify individuals at higher risk of developing heart disease. Our primary reasons for choosing an ANN model are:

- Pattern Recognition: ANNs excel at recognizing patterns within complex datasets. Heart disease prediction involves analyzing various factors such as blood pressure, cholesterol levels, age, and lifestyle habits. ANNs can learn intricate patterns from these factors and their interactions, enabling accurate risk assessment.
- Feature Extraction: ANNs can automatically extract relevant features from raw data.
- Scalability: ANNs can scale effectively to handle large and diverse datasets
- Continuous Learning: ANNs can continuously learn and adapt to new information, making them suitable for dynamic environments such as healthcare.

In this report, we present the methodology employed in building the ANN model, the dataset utilized for training and evaluation, the results obtained, and the implications of our findings. We developed and documented our project using a jupyter notebook. The following pages of this report document our work from the jupyter notebook, going through each step of the process. The analysis, challenges, and possible improvements are also discussed in each step of the process and also towards the end.

2. Three Approaches to the project

1. In our first approach, we coded a basic neural network from scratch using NumPy. We did this so that we can understand all the computations that are done during forward and backward propagation and learn how to implement those computations in code.
2. In our second approach we used keras built on top of tensorflow to create the same model as the first one and train it on the same dataset. We have also used **Genetic Algorithm to optimise some of the parameters of the model**.
3. Dissatisfied with the results of the first dataset, we used a higher quality dataset in the third approach to train the neural networks built in the first two approaches and see how they perform.

3. Setup

Firstly, we will need to import the necessary libraries to pre-process the data and create a neural network model. We will code a neural network from scratch using NumPy in our first approach. For the second approach we will be using keras built on top of tensorflow to create a model with the same architecture. For the third approach we will test our tensorflow model on another dataset.

In [109…

```
import tensorflow as tf
from sklearn.metrics import accuracy_score, confusion_matrix, classification
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from keras.utils import plot_model
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
# Set seed for NumPy
np.random.seed(42)
# Set seed for TensorFlow
tf.random.set_seed(42)

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/heart-attack-prediction-dataset/heart_attack_prediction_datase
t.csv
/kaggle/input/heart-failure-prediction/heart.csv
/kaggle/input/heart-attack-prediction-dataset/heart_attack_prediction_datase
t.csv
/kaggle/input/heart-failure-prediction/heart.csv
```

Importing the dataset and looking at the first five rows

> We will use the pandas library to import the training data and store it in the 'file' variable as a pandas DataFrame.

In [110…

```
file = pd.read_csv("/kaggle/input/heart-attack-prediction-dataset/heart_atta
file.head()
```

Out[110]:

	Patient ID	Age	Sex	Cholesterol	Blood Pressure	Heart Rate	Diabetes	Family History	Smoking	Obesity
0	BMW7812	67	Male	208	158/88	72	0	0	1	0
1	CZE1114	21	Male	389	165/93	98	1	1	1	1
2	BNI9906	21	Female	324	174/99	72	1	0	0	0
3	JLN3497	84	Male	383	163/100	73	1	1	1	0
4	GFO8847	66	Male	318	91/88	93	1	1	1	1

5 rows × 26 columns

Setting the Objective

As we can see from the data, it provides us with different features of a patient and in the last column it indicates whether the person has a risk of heart attack using 0 or 1. Thus, we will treat this as a **Binary Classification** problem by building the model to predict whether a person has a risk of heart attack or not. The output of the model will be 0 or 1 instead of a probability.

4. Data Pre-Processing

Before the data can be fed into our model to train, the data needs to be processed. Irrelevant features will be removed and categorical features will be OneHotEncoded.

> First, we use file.shape to determine the number of rows and columns in the data.

```
In [111…  file.shape
```

```
Out[111]: (8763, 26)
```

> The shape of the dataframe is (8763, 26), which means there's 8763 patients and 26 features (columns) per patient. We will look at the columns in the dataframe and select the suitable ones for training.

```
In [112…  file.columns #outputs the column names
```

```
Out[112]: Index(['Patient ID', 'Age', 'Sex', 'Cholesterol', 'Blood Pressure',
                 'Heart Rate', 'Diabetes', 'Family History', 'Smoking', 'Obesity',
                 'Alcohol Consumption', 'Exercise Hours Per Week', 'Diet',
                 'Previous Heart Problems', 'Medication Use', 'Stress Level',
                 'Sedentary Hours Per Day', 'Income', 'BMI', 'Triglycerides',
                 'Physical Activity Days Per Week', 'Sleep Hours Per Day', 'Country',
                 'Continent', 'Hemisphere', 'Heart Attack Risk'],
                dtype='object')
```

- Features such as Patient ID, Income, Continent, Hemisphere, Country are unlikely factors that contribute to the risk of heart attack, thus we will drop these columns from the dataframe; these features will not be considered when training the data and predicting the risk of heart attack.
- We will also drop the "Heart Attack Risk" column as this is our target column. We will store the dataframe with the dropped columns in the variable 'x' which will be the input to the neural network.
- The target column will be stored in the variable 'y' which the model will use to calculate error. Thus, y will be an array with only 1 column and 8763 rows.

```
In [113…  x = file.drop(["Patient ID", "Income","Continent","Hemisphere","Heart Attack
          y = file['Heart Attack Risk']
```

```
In [114…  print("Shape of y: ", y.shape)
          y.head()
```

```
          Shape of y:  (8763,)
Out[114]:  0    0
           1    0
           2    0
           3    0
           4    0
           Name: Heart Attack Risk, dtype: int64
```

OneHotEncoding and converting non-numerical data into numerical data

- The columns Diet and Sex contain non-numerical values, thus these cannot be fed directly into the neural network as the network will not be able to do any mathematical operations on them. We will one-hot encode the 'Diet' and 'Sex' columns. One-hot encoding is a technique used to convert categorical variables into a binary matrix format, where each category is represented by a binary vector. For example, in the 'Diet' column, 'healthy' might be represented as [1, 0, 0], 'unhealthy' as [0, 1, 0], and 'average' as [0, 0, 1]. Similarly, in the 'Sex' column, 'male' might be represented as [1, 0] and 'female' as [0, 1]. This transformation allows the neural network to effectively process categorical data by treating each category as a separate binary feature, enabling it to learn relationships between the categories and the target variable.
- The blood pressure column is given as a string in the format Systolic/Diastolic. We will create seperate columns for Systolic and Diastolic that contains the integer value. As we created the two new columns, the 'Blood Pressure' column will not be required anymore so we will drop it.

In [115…

```
#OneHotEncoding Diet and Sex

x = pd.get_dummies(x, columns=["Diet","Sex"],prefix=["Diet","Sex"])

#Seperating the Systolic and Diastolic Blood Pressure and dropping the Blood

x[["Systolic","Diastolic"]] = x["Blood Pressure"].str.split('/', expand=True
x[['Systolic', 'Diastolic']] = x[['Systolic', 'Diastolic']].apply(pd.to_nume
x = x.drop('Blood Pressure', axis=1)
```

> Now the data has been processed as required. All irrelevant columns have been dropped and all non-numeric values have been converted into numeric values. We can see the processed data below along with it's new shape. It still has 8763 rows but the number of columns have changed since we dropped a few and created a few through OneHotEncoding.

In [116…

```
print("Shape: ", x.shape)
x.head()
```

```
Shape:  (8763, 24)
```

Out[116]:

	Age	Cholesterol	Heart Rate	Diabetes	Family History	Smoking	Obesity	Alcohol Consumption	Exercise Hours Per Week	Prev I Prob
0	67	208	72	0	0	1	0	0	4.168189	
1	21	389	98	1	1	1	1	1	1.813242	
2	21	324	72	1	0	0	0	0	2.078353	
3	84	383	73	1	1	1	0	1	9.828130	
4	66	318	93	1	1	1	1	0	5.804299	

5 rows × 24 columns

Normalisation

> There is one last step remaining in pre-processing the data before we can feed it into the neural network. Since the columns have a range of different values, we will carry out normalisation to improve gradient descent. In this step, the data will also be converted from a DataFrame to a numpy array so that it can be easily fed into the neural network.

In [117…
```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
x=scaler.fit_transform(x)
x
```

Out[117]:
```
array([[ 6.25557131e-01, -6.41578894e-01, -1.47042098e-01, ...,
         6.58765153e-01,  8.70044389e-01,  1.93781814e-01],
       [-1.53932232e+00,  1.59689495e+00,  1.11817855e+00, ...,
         6.58765153e-01,  1.13571444e+00,  5.34480428e-01],
       [-1.53932232e+00,  7.93023127e-01, -1.47042098e-01, ...,
        -1.51799165e+00,  1.47729021e+00,  9.43318765e-01],
       ...,
       [-3.15694803e-01, -1.22154025e-01,  1.45881488e+00, ...,
         6.58765153e-01,  9.83902981e-01, -6.92034583e-01],
       [-8.33383367e-01, -1.01259666e+00, -7.30990089e-01, ...,
         6.58765153e-01, -6.10117296e-01, -1.23715237e+00],
       [-1.35107193e+00,  1.18877541e+00, -1.05510022e-03, ...,
        -1.51799165e+00,  1.10987115e-01, -1.23715237e+00]])
```

Creating a training set and validation set

- Since we want our model to generalise, we will be splitting the data into a train set and validation set. The validation set will comprise of 20% of the original data and the rest of the 80% of the data will be used for training. Once training is done using the training set, we can use the validation set to see how well our model has generalised.

In [118…
```
from sklearn.model_selection import train_test_split
random_seed = 42 #to get consistent results
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, rar

#Adjusting the shape of y
y_train = y_train.values.reshape(-1, 1)
y_test = y_test.values.reshape(-1, 1)
y_test.shape
```
Out[118]:
```
(1753, 1)
```

5. First Approach - Coding a Neural Network from Scratch

- Below is the code for the neural network we coded using numpy. The code is divided into several functions for each step which are compiled together in **L_layer_model(X, Y, layers_dims, learning_rate=0.01, num_iterations=30, print_cost=True)** to create a functional model. The model is hard-coded to have **ReLU activation in the hidden layers and Sigmoid activation** in the output layer. The loss is computed using the **compute_cost()** function which computes the loss using **Binary Crossentropy** since we are trying to predict **True** or **False** instead of a probability. The description of each function is given as a comment under the respective function.

Here is an overview of the functions that are used to built the neural network.

> - **relu**: Computes the ReLU.
> - **relu_backward**: Implements the backward propagation for a single ReLU unit.
> - **sigmoid_backward**: Implements the backward propagation for a single sigmoid unit.
> - **initialize_parameters_deep**: Initializes the parameters of a deep neural network.
> - **linear_forward**: Implements the linear part of a layer's forward propagation.
> - **linear_activation_forward**: Implements the forward propagation for the linear -> activation.
> - **L_model_forward**: Implements forward propagation for the entire network.
> - **compute_cost**: Computes the cost function (cross-entropy cost).
> - **linear_backward**: Implements the linear portion of backward propagation for a single layer.
> - **linear_activation_backward**: Implements the backward propagation for the linear -> activation.

> - **L_model_backward**: Implements the backward propagation for the entire network.
> - **update_parameters**: Updates parameters using gradient descent.
> - **L_layer_model**: Implements a L-layer neural network. L is the number of layers defined by the user.
> - **predict**: Predicts the results of a L-layer neural network.
> - **plot_costs**: Plots the learning curve.

In [119…

```
import numpy as np
import matplotlib.pyplot as plt
import copy

def sigmoid(Z):
    """
    Compute the sigmoid activation function element-wise.

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of the sigmoid function, same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """
    A = 1 / (1 + np.exp(-Z))
    cache = Z
    return A, cache

def relu(Z):
    """
    Compute the ReLU activation function element-wise.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the b
    """
    A = np.maximum(0, Z)
    assert (A.shape == Z.shape)
    cache = Z
    return A, cache

def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficient

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0
    assert (dZ.shape == Z.shape)
    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficient

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """
    Z = cache
    s = 1 / (1 + np.exp(-Z))
    dZ = dA * s * (1 - s)
    assert (dZ.shape == Z.shape)
    return dZ

def initialize_parameters_deep(layer_dims):
    """
    Initialize the parameters of the deep neural network.

    Arguments:
    layer_dims -- python array (list) containing the dimensions of each laye

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", .
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[
                    bl -- bias vector of shape (layer_dims[l], 1)
    """
    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)
    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
        assert (parameters['W' + str(l)].shape == (layer_dims[l], layer_dims
        assert (parameters['b' + str(l)].shape == (layer_dims[l], 1))
    return parameters

def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous 
    W -- weights matrix: numpy array of shape (size of current layer, size o
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation pa
    cache -- a python dictionary containing "A", "W" and "b" ; stored for co
    """
    Z = W.dot(A) + b
    cache = (A, W, b)
    return Z, cache
```

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of prev
    W -- weights matrix: numpy array of shape (size of current layer, size o
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text

    Returns:
    A -- the output of the activation function, also called the post-activat
    cache -- a python tuple containing "linear_cache" and "activation_cache"
             stored for computing the backward pass efficiently
    """
    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
    cache = (linear_cache, activation_cache)
    return A, cache

def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGM

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- activation value from the output (last) layer
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L of t
    """
    caches = []
    A = X
    L = len(parameters) // 2
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)
        caches.append(cache)
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], param
    caches.append(cache)
    return AL, caches

def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat

    Returns:
    cost -- cross-entropy cost
    """
    m = Y.shape[1]
    cost = -np.sum(np.multiply(np.log(AL), Y) + np.multiply(np.log(1 - AL),
    cost = np.squeeze(cost)
    return cost

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current
    cache -- tuple of values (A_prev, W, b) coming from the forward propagat

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the p
    dW -- Gradient of the cost with respect to W (current layer l), same sha
    db -- Gradient of the cost with respect to b (current layer l), same sha
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    db = (np.sum(dZ, axis=1, keepdims=True)) / m
    dA_prev = np.dot(W.T, dZ)
    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for c
    activation -- the activation to be used in this layer, stored as a text

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the p
    dW -- Gradient of the cost with respect to W (current layer l), same sha
    db -- Gradient of the cost with respect to b (current layer l), same sha
    """
    linear_cache, activation_cache = cache
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LIN

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_for
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (it's
                the cache of linear_activation_forward() with "sigmoid" (it'
    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    current_cache = caches[L - 1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current
    grads["dA" + str(L - 1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp
    for l in reversed(range(L - 1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["d
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
    return grads

def update_parameters(params, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    params -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """
    parameters = copy.deepcopy(params)
    L = len(parameters) // 2
    for l in range(L):
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learni
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learni
    return parameters

def L_layer_model(X, Y, layers_dims, learning_rate=0.01, num_iterations=30,
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMO

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (
    layers_dims -- list containing the input size and each layer size, of le
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 5 steps

    Returns:
    parameters -- parameters learnt by the model. They can then be used to p
    """
    np.random.seed(1)
    costs = []  # keep track of cost
    parameters = initialize_parameters_deep(layers_dims)
    for i in range(0, num_iterations):
        AL, caches = L_model_forward(X, parameters)
        cost = compute_cost(AL, Y)
        grads = L_model_backward(AL, Y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)
        if print_cost and i % 100 == 0 or i == num_iterations - 1:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if i % 100 == 0 or i == num_iterations:
            costs.append(cost)
    return parameters, costs

def predict(X, y, parameters):
    """
    This function is used to predict the results of a  L-layer neural networ

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """
    m = X.shape[1]
    n = len(parameters) // 2
    p = np.zeros((1, m))
    probas, caches = L_model_forward(X, parameters)
    for i in range(0, probas.shape[1]):
        if probas[0, i] > 0.5:
            p[0, i] = 1
        else:
            p[0, i] = 0
    print("Accuracy: " + str(np.sum((p == y) / m)))
    return p

def plot_costs(costs, learning_rate=0.0075):
    """
    Plot the learning curve.

    Arguments:
    costs -- list of costs
    learning_rate -- learning rate used during training
    """
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per hundreds)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()
```

Creating the Neural Network Architecture

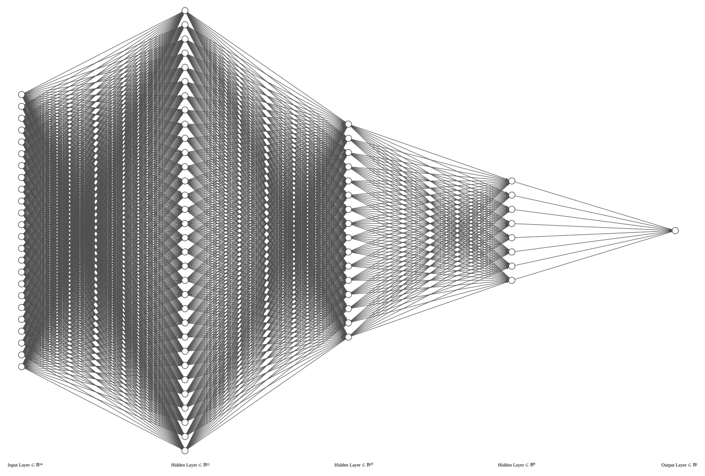
- We are going to use Dense layers, which means all the neurons in a layer will be connected to all the neurons in the previous layer.

- Using the neural network coded in the previous cell, we can build a custom model by specifying the number of layers and the neurons per layer. This is done in the **layer_dims** list in the following code cell.
- Building a model is an iterative process, it is difficult to built the optimum model in the first try. For our baseline model we will use 5 layers (including the input and output layer). The **input layer consists of 24 nodes as there are 24 features**. The **3 hidden layers contain 32, 16 and 8 neurons respectively**. The **output layer contains only 1 neuron**. The architecture strikes a balance between model complexity and capacity. With 5 layers, the network has the capacity to learn intricate patterns and relationships within the data. However, it's not overly complex, which could lead to overfitting, especially in scenarios where the dataset is not sufficiently large.
- The input layer has no activation function. The **hidden layers compute ReLU activation** as it will increase the rate of gradient descent and the **output layer computes a sigmoid activation** since it is the best choice for binary classification.

In [120…

```
### Layers ###
layers_dims = [24, 32, 16, 8, 1] #Specify the number of neurons in each laye
```

Below is the visual represenation of our neural network.



Training the model

> After we're done with creating the architecture for the neural network, we can train the model using model.compile.
>
> - The learning rate is set to a low value of **0.01**
> - The **loss** is calculated using **BinaryCrossentropy()**
> - The network is trained for **1000 epochs**

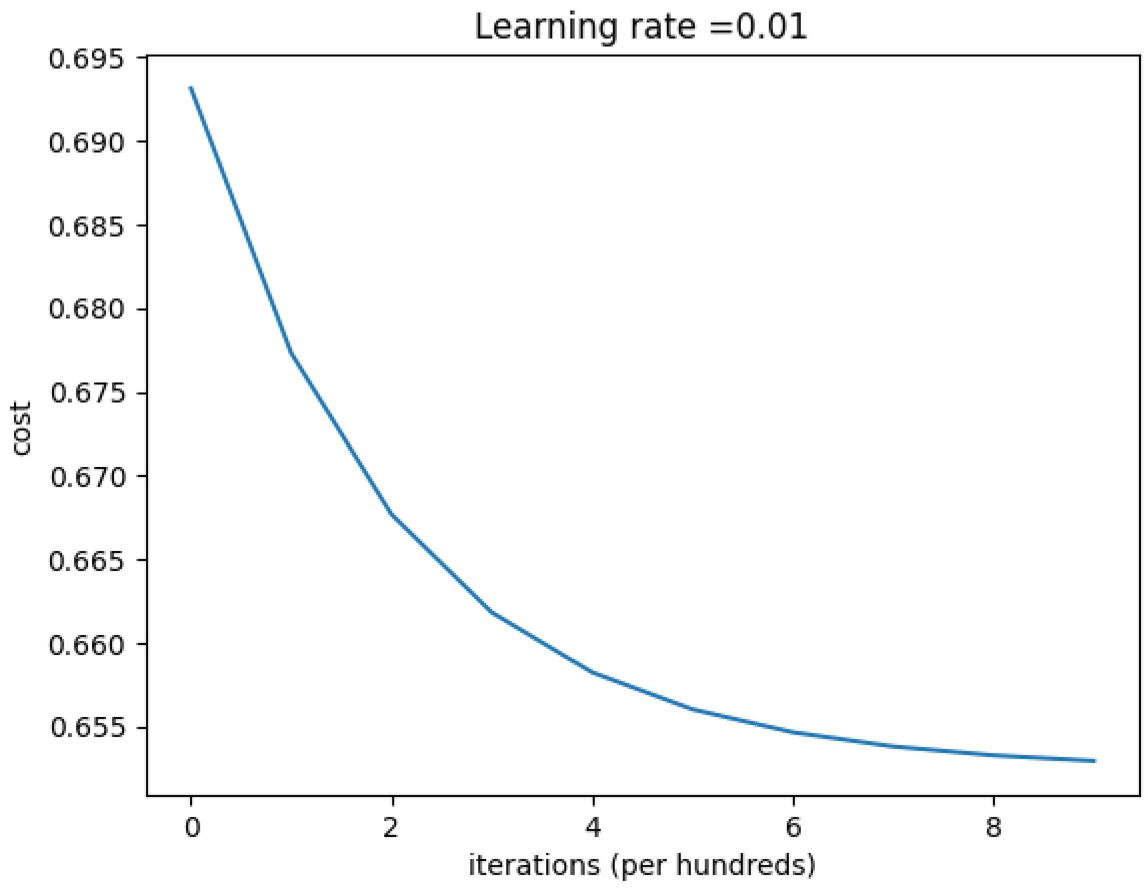
> - The learned parameters *(updated weights and biases)* from the training is stored in the **parameters** variable

A graph plot is produced to observe the cost against the epochs.

In [121…

```
#Train the model
parameters, costs = L_layer_model(x_train.T, y_train.T, layers_dims, num_ite
#Plot the graph of cost against iteration
plot_costs(costs, learning_rate=0.01)
```

```
Cost after iteration 0: 0.6931472024014913
Cost after iteration 100: 0.6773066895726916
Cost after iteration 200: 0.6676857039133842
Cost after iteration 300: 0.6618185462939213
Cost after iteration 400: 0.6582232688049151
Cost after iteration 500: 0.6560096455468513
Cost after iteration 600: 0.654640822241673
Cost after iteration 700: 0.6537912266471969
Cost after iteration 800: 0.6532622416640248
Cost after iteration 900: 0.6529320195290441
Cost after iteration 999: 0.6527270598939531
```



Predicting with the model

We are going to use the learned parameters of the model to predict from both the training set and test set

In [122…

```
#predicting on the train set
predictions_train = predict(x_train.T, y_train.T, parameters)
#predicting on the test set
predictions_test = predict(x_test.T, y_test.T, parameters)
```

```
Accuracy: 0.6417974322396576
Accuracy: 0.6417569880205363
```

Analysing the model

The accuracy on both the training set and the test set is similar. The model isn't overfitting to the training set but it is also not generalising properly. We are not using any momentum optimisation, thus the loss may haven gotten stuck in a local minima which is why it is not overfitting even after 1000 epochs.

Possible reasons for low accuracy

> The low accuracy can be due to a number of reasons. Firstly, it can be due to a bad model with suboptimal hyperparameters or due to incorrect feature selection. However, in this case, the most likely reason for the low accuracy is because of the data. According to the publisher of the data, the dataset is a synthetic dataset generated using ChatGPT. Given this, it's understandable that the data may not faithfully represent real-world scenarios or exhibit meaningful patterns. Therefore, it's conceivable that the output accuracy is hindered by the inherent limitations of the dataset quality, adhering to the principle of ***garbage in, garbage out.***

6. Second Approach - Building a Model Using Tensorflow Keras

In our first approach we built a neural network from scratch to understand how to implement the computations in code. In the second approach, we are going to use Keras on top of tensorflow to train the model on the same dataset. We are opting for a Keras model because it comes in-built with optimisations that will help us run the model faster. It will also allow us to easily tune the hyperparameters using Genetic Algorithm.

Building the Tensorflow Model

As we are going to use the same dataset from the previous approach, there is no further need to pre-process the data; we can jump straight to building the Keras model.

> We are using Keras to build our neural network model since it allows us to easily create the neural network architecture without having to do any math. Using Keras, we can simply just add a layer using Dense() and specify the number of neurons in the layer and the activation function. The architecture of this network will be the same as the first approach.
>
> - We are going to use Dense layers, which means all the layers will be connected to all the neurons in the previous layer.
> - We will use 5 layers (including the input and output layer). The **input layer consists of 24 nodes as there are 24 features**. The **3 hidden layers contain 32, 16 and 8 neurons respectively**. The **output layer contains only 1 neuron**. (The same baseline model as the previous approach)
> - The input layer has no activation function. The **hidden layers compute ReLU activation** and the **output layer computes a sigmoid activation** since it is the best choice for binary classification.

In [123…

```
n = x.shape[1] #number of features

#Creating the neural network model
model = Sequential([
    tf.keras.Input(shape=(n,), name="input_layer"), #Input Layer with 'n' ne
    Dense(units=32, activation="relu", name="hidden_layer1"),
    Dense(units=16, activation="relu", name="hidden_layer2"),
    Dense(units=8, activation="relu", name="hidden_layer3"),
    Dense(units=1, activation="sigmoid", name="output_layer"),
])

model.summary()
```

```
Model: "sequential_52"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer1 (Dense)       (None, 32)                800

 hidden_layer2 (Dense)       (None, 16)                528

 hidden_layer3 (Dense)       (None, 8)                 136

 output_layer (Dense)        (None, 1)                 9

=================================================================
Total params: 1473 (5.75 KB)
Trainable params: 1473 (5.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the model

After we're done with creating the architecture for the neural network, we can train the model using model.compile.

- We're using the **Adam optimisation** algorithm which makes use of **momentum** and **RMSProp** to smooth out the gradient descent.
- The learning rate is set to a low value of **0.01**
- The **loss** is calculated using **BinaryCrossentropy()**
- The network is trained for **250 epochs** with a **batch size = dataset size**

In [124…
```
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=0.01), loss
history = model.fit(x_train, y_train, batch_size = x_train.shape[0], epochs
Epoch 237/250
1/1 [==============================] - 0s 33ms/step - loss: 0.4973 - accurac
y: 0.7579 - val_loss: 0.8890 - val_accuracy: 0.5533
Epoch 238/250
1/1 [==============================] - 0s 34ms/step - loss: 0.4975 - accurac
y: 0.7532 - val_loss: 0.9047 - val_accuracy: 0.5459
Epoch 239/250
1/1 [==============================] - 0s 30ms/step - loss: 0.4962 - accurac
y: 0.7595 - val_loss: 0.8960 - val_accuracy: 0.5533
Epoch 240/250
1/1 [==============================] - 0s 31ms/step - loss: 0.4948 - accurac
y: 0.7562 - val_loss: 0.9024 - val_accuracy: 0.5488
Epoch 241/250
1/1 [==============================] - 0s 33ms/step - loss: 0.4936 - accurac
y: 0.7603 - val_loss: 0.9011 - val_accuracy: 0.5539
Epoch 242/250
1/1 [==============================] - 0s 32ms/step - loss: 0.4926 - accurac
y: 0.7619 - val_loss: 0.9008 - val_accuracy: 0.5533
Epoch 243/250
1/1 [==============================] - 0s 33ms/step - loss: 0.4921 - accurac
y: 0.7593 - val_loss: 0.9068 - val_accuracy: 0.5454
Epoch 244/250
1/1 [==============================] - 0s 34ms/step - loss: 0.4922 - accurac
y: 0.7616 - val_loss: 0.8974 - val_accuracy: 0.5573
Epoch 245/250
1/1 [==============================] - 0s 35ms/step - loss: 0.4938 - accurac
y: 0.7555 - val_loss: 0.9186 - val_accuracy: 0.5448
Epoch 246/250
1/1 [==============================] - 0s 34ms/step - loss: 0.4952 - accurac
y: 0.7588 - val_loss: 0.8991 - val_accuracy: 0.5579
Epoch 247/250
1/1 [==============================] - 0s 34ms/step - loss: 0.4951 - accurac
y: 0.7515 - val_loss: 0.9172 - val_accuracy: 0.5408
Epoch 248/250
1/1 [==============================] - 0s 36ms/step - loss: 0.4909 - accurac
y: 0.7616 - val_loss: 0.9201 - val_accuracy: 0.5425
Epoch 249/250
1/1 [==============================] - 0s 34ms/step - loss: 0.4891 - accurac
y: 0.7635 - val_loss: 0.9091 - val_accuracy: 0.5556
Epoch 250/250
1/1 [==============================] - 0s 35ms/step - loss: 0.4920 - accurac
y: 0.7569 - val_loss: 0.9260 - val_accuracy: 0.5414
```

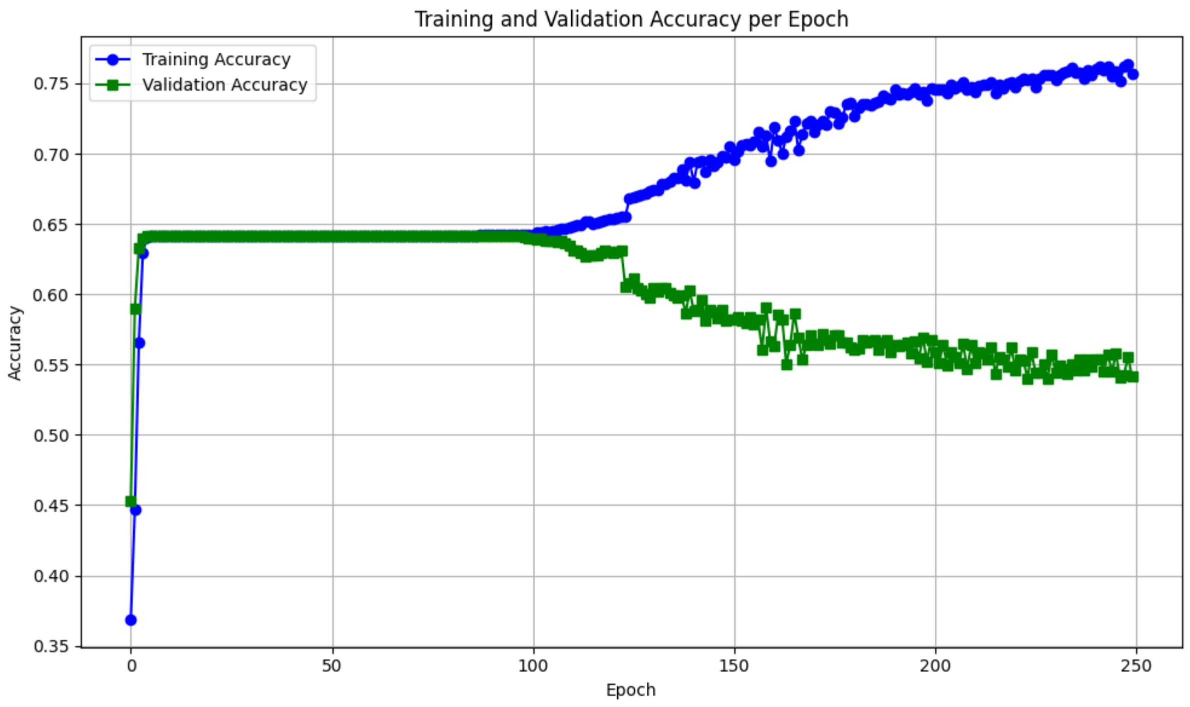
Analysing the model

We can analyse how well our tensorflow model performs by looking at the loss and accuracy plotted against epoch. As we can see from the graphs below, the accuracy when the model predicts from data it has already been trained on (the training set) increases per epoch. However, when the model sees new data (validation set) the accuracy decreases per epoch.

In [125…

```
import matplotlib.pyplot as plt
val_acc = history.history['val_accuracy']  # Assuming the metric is named 'v
plt.figure(figsize=(10, 6))

# Plot training accuracy
```

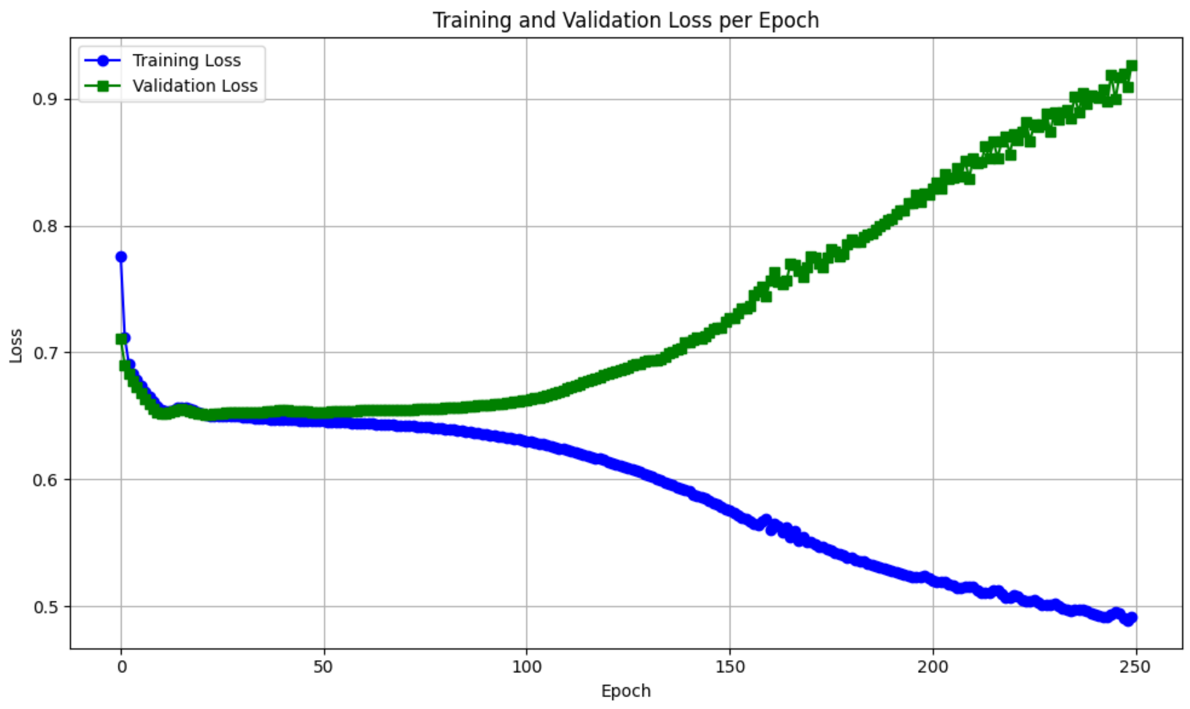
```
plt.plot(range(len(history.history['accuracy'])), history.history['accuracy'

# Plot validation accuracy
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy', marker='

# Add labels, title, legend, grid, etc.
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy per Epoch')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



From the graph under the code cell below, we can see the Loss for the training set decreases per epoch, but the loss for validation increases per epoch.

In [126…

```
import matplotlib.pyplot as plt
val_acc = history.history['val_loss']
plt.figure(figsize=(10, 6))

# Plot training accuracy
plt.plot(range(len(history.history['loss'])), history.history['loss'], label
# Plot validation accuracy
plt.plot(range(len(val_acc)), val_acc, label='Validation Loss', marker='s',

# Add labels, title, legend, grid, etc.
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss per Epoch')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Below we have the classification report when the model predicts from the training set.

In [127…

```
# Apply threshold for binary predictions
a = x_train
b = y_train
predictions = model.predict(a)
threshold = 0.5
binary_predictions = (predictions >= threshold).astype(int)

# Evaluate and print results
accuracy = accuracy_score(b, binary_predictions)
conf_matrix = confusion_matrix(b, binary_predictions)
class_report = classification_report(b, binary_predictions)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
```

```
220/220 [==============================] - 0s 1ms/step
Accuracy: 0.7620542082738945
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.83      0.82      4499
           1       0.68      0.65      0.66      2511

    accuracy                           0.76      7010
   macro avg       0.74      0.74      0.74      7010
weighted avg       0.76      0.76      0.76      7010
```

Below we have the classification report when the model predicts from the test set.

In [128…

```
# Apply threshold for binary predictions
a = x_test
```

```
b = y_test
predictions = model.predict(a)
threshold = 0.5
binary_predictions = (predictions >= threshold).astype(int)

# Evaluate and print results
accuracy = accuracy_score(b, binary_predictions)
conf_matrix = confusion_matrix(b, binary_predictions)
class_report = classification_report(b, binary_predictions)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
```

```
55/55 [==============================] - 0s 1ms/step
Accuracy: 0.5413576725613235
Classification Report:
              precision    recall  f1-score   support

           0       0.64      0.64      0.64      1125
           1       0.36      0.37      0.37       628

    accuracy                           0.54      1753
   macro avg       0.50      0.50      0.50      1753
weighted avg       0.54      0.54      0.54      1753
```

As we can see from the accuracy against epochs graph, the tensorflow model performs the same on the validation set as the neural network coded from scratch until about 30 epochs. From this we can conclude that the calculations within our coded neural network in the first approach were accurate. However, **the tensorflow model starts overfitting which our first model did not**, even though we ran more epochs in the first model. This is most likely due to the use of **Adam optimisation, which helps the gradient descent to escape a local minima and start fitting well to the training set**. Due to the overfitting the accuracy on the training set increases but the accuracy on the validation set decreases.

Training the Keras neural network using Genetic Algorithm

Genetic Algorithm can be used to to tune the hyperparameters of a model by searching for the best combination of parameters such as number of hidden layers, number of neurons, learning rate etc.

For our demo we are using GA to omptimise the model by only finding the best combination of neurons in the Hidden Layers. This is a maximisation operation where the best choice is that which produces the model with the most accurate results (i.e. the **accuracy** of the model is the **fitness value**). Since training a genetic algorithm takes a lot of time, we will only be using a **initial population size of 5**, and **3 maximum generations**. The **crossover probability is 0.8** and the **mutation probabilty is 0.2.**

As training a Genetic Algorithm is computationally resource intensive, we have opted for a basic implementation here. With enough time and resources, we can test for other hyperparameters as well.

In [129…

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from deap import base, creator, tools

# Define evaluation function
def evaluate(individual):
    # Decode the individual to obtain the architecture
    n_neurons_hidden1, n_neurons_hidden2, n_neurons_hidden3 = individual
    # Construct the neural network
    model = Sequential([
        tf.keras.Input(shape=(n,), name="input_layer"),
        Dense(units=n_neurons_hidden1, activation="relu", name="hidden_layer
        Dense(units=n_neurons_hidden2, activation="relu", name="hidden_layer
        Dense(units=n_neurons_hidden3, activation="relu", name="hidden_layer
        Dense(units=1, activation="sigmoid", name="output_layer"),
    ])
    # Compile the model
    model.compile(optimizer=Adam(learning_rate=0.1), loss="binary_crossentro
    # Train the model
    history = model.fit(x_train, y_train, batch_size=8763, epochs=200, valid
    # Evaluate the model
    loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
    # Return negative accuracy as fitness (maximization problem)
    return accuracy,

# Define genetic algorithm parameters
POP_SIZE = 5
NUM_GENERATIONS = 3
CROSSOVER_PROB = 0.8
MUTATION_PROB = 0.2

# Create a toolbox for the genetic algorithm
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("attr_int", np.random.randint, low=1, high=100)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=1, up=100, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Generate initial population
population = toolbox.population(n=POP_SIZE)

# Run the genetic algorithm
for gen in range(NUM_GENERATIONS):
    print("Generation", gen + 1)
    # Evaluate the fitness of the population
    fitnesses = list(map(toolbox.evaluate, population))
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    # Print the parents' values and fitness values
    print("Parents:")
    for ind in population:
        print("Individual:", ind, "Fitness:", ind.fitness.values[0])

    # Select parents for reproduction
    offspring = toolbox.select(population, len(population))
    offspring = list(map(toolbox.clone, offspring))

    # Apply crossover and mutation
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if np.random.rand() < CROSSOVER_PROB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:
        if np.random.rand() < MUTATION_PROB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Print the children's values
    print("Children:")
    for ind in offspring:
        print("Individual:", ind)

    # Replace the population with the offspring
    population[:] = offspring

# Get the best individual
best_individual = tools.selBest(population, k=1)[0]
print("Best individual:", best_individual)
```

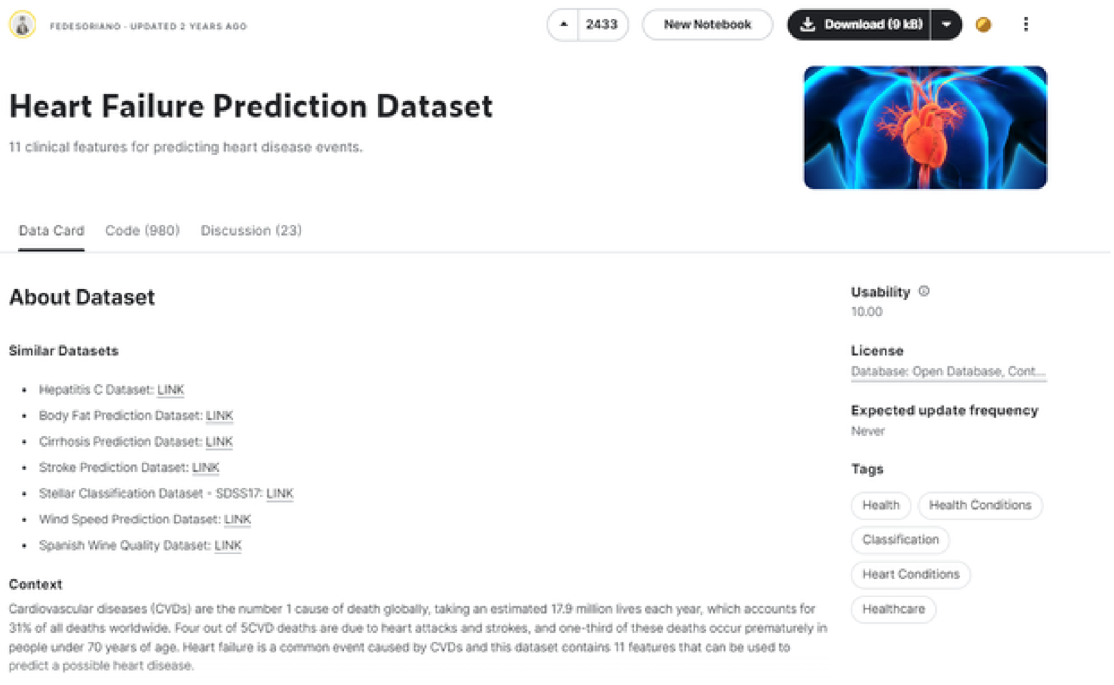
```
Generation 1
```

```
/opt/conda/lib/python3.10/site-packages/deap/creator.py:185: RuntimeWarning:
A class named 'FitnessMax' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
/opt/conda/lib/python3.10/site-packages/deap/creator.py:185: RuntimeWarning:
A class named 'Individual' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
Parents:
Individual: [12, 95, 16] Fitness: 0.6417570114135742
Individual: [92, 26, 49] Fitness: 0.6417570114135742
Individual: [6, 18, 66] Fitness: 0.6417570114135742
Individual: [22, 56, 47] Fitness: 0.6417570114135742
Individual: [6, 17, 52] Fitness: 0.6417570114135742
Children:
Individual: [22, 56, 49]
Individual: [92, 26, 47]
Individual: [22, 56, 49]
Individual: [92, 26, 47]
Individual: [92, 26, 49]
Generation 2
Parents:
Individual: [22, 56, 49] Fitness: 0.6417570114135742
Individual: [92, 26, 47] Fitness: 0.6417570114135742
Individual: [22, 56, 49] Fitness: 0.6417570114135742
Individual: [92, 26, 47] Fitness: 0.6417570114135742
Individual: [92, 26, 49] Fitness: 0.6417570114135742
Children:
Individual: [22, 56, 49]
Individual: [92, 26, 49]
Individual: [89, 56, 49]
Individual: [22, 26, 49]
Individual: [22, 56, 49]
Generation 3
Parents:
Individual: [22, 56, 49] Fitness: 0.6417570114135742
Individual: [92, 26, 49] Fitness: 0.6417570114135742
Individual: [89, 56, 49] Fitness: 0.6417570114135742
Individual: [22, 26, 49] Fitness: 0.6417570114135742
Individual: [22, 56, 49] Fitness: 0.6417570114135742
Children:
Individual: [22, 26, 49]
Individual: [22, 56, 49]
Individual: [92, 26, 81]
Individual: [22, 26, 49]
Individual: [22, 56, 49]
Best individual: [22, 56, 49]
```

After training for 3 iterations, the genetic algorithm gives us the best combination of neurons per hidden layer. However, as we can see from the fitness function of this combination, it is still only around 64% accurate. This low accuracy can be attributed to the quality of the data instead of the quality of the model.

7. Third Approach - Testing the Neural Network Models on a Different Dataset

We have noticed in the previous two approaches that our neural networks did not perform well on the dataset it was trained on. The possible reason for the poor performance is the low quality of the synthetic dataset. Therefore, to demonstrate that our neural network models are working properly, we are going to train and test them on a different dataset. Click here to see the new dataset we are going to use



Heart Failure Prediction Dataset

11 clinical features for predicting heart disease events.

About Dataset

Similar Datasets

- Hepatitis C Dataset: LINK
- Body Fat Prediction Dataset: LINK
- Cirrhosis Prediction Dataset: LINK
- Stroke Prediction Dataset: LINK
- Stellar Classification Dataset - SDSS17: LINK
- Wind Speed Prediction Dataset: LINK
- Spanish Wine Quality Dataset: LINK

Context

Cardiovascular diseases (CVDs) are the number 1 cause of death globally, taking an estimated 17.9 million lives each year, which accounts for 31% of all deaths worldwide. Four out of 5CVD deaths are due to heart attacks and strokes, and one-third of these deaths occur prematurely in people under 70 years of age. Heart failure is a common event caused by CVDs and this dataset contains 11 features that can be used to predict a possible heart disease.

Usability
10.00

License
Database: Open Database, Cont...

Expected update frequency
Never

Tags
Health, Health Conditions, Classification, Heart Conditions, Healthcare

Data Loading and Pre-Processing

Since the new dataset has slightly different features from the previous dataset, we will need to pre-process the data using the same steps we did the first time.

In [130…

```
df = pd.read_csv("/kaggle/input/heart-failure-prediction/heart.csv")
df.head()
```

Out[130]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	Exercis
0	40	M	ATA	140	289	0	Normal	172	
1	49	F	NAP	160	180	0	Normal	156	
2	37	M	ATA	130	283	0	ST	98	
3	48	F	ASY	138	214	0	Normal	108	
4	54	M	NAP	150	195	0	Normal	122	

In [131…

```
#One Hot Encoding
df = pd.get_dummies(df, columns=["Sex","ChestPainType","RestingECG","Exercis

#Splitting the input and the output
X_new = df.drop(["HeartDisease"], axis = 1)
y_new = df["HeartDisease"]

#Normalisation of the data and splitting into train and test set
X_new = scaler.fit_transform(X_new)
x_train_new, x_test_new, y_train_new, y_test_new = train_test_split(X_new, y

#Adjusting the shape of y
```

```
y_train_new = y_train_new.values.reshape(-1, 1)
y_test_new = y_test_new.values.reshape(-1, 1)
```

In [132…

```
y_train_new.shape
```

Out[132]:

```
(734, 1)
```

Building the Keras Model

We will be using the same baseline model as before with 5 layers, however the numper of neurons in the input layer will be different as we have different number of features in this new dataset

In [133…

```
n_new = X_new.shape[1] #number of features

#Creating the neural network model
model_new = Sequential([
    tf.keras.Input(shape=(n_new,), name="input_layer"), #Input Layer with 'n
    Dense(units=32, activation="relu", name="hidden_layer1"),
    Dense(units=16, activation="relu", name="hidden_layer2"),
    Dense(units=8, activation="relu", name="hidden_layer3"),
    Dense(units=1, activation="sigmoid", name="output_layer"),
])

model.summary()
```

```
Model: "sequential_52"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer1 (Dense)       (None, 32)                800

 hidden_layer2 (Dense)       (None, 16)                528

 hidden_layer3 (Dense)       (None, 8)                 136

 output_layer (Dense)        (None, 1)                 9

=================================================================
Total params: 1473 (5.75 KB)
Trainable params: 1473 (5.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the Keras Model

- Learning rate = 0.001
- Batch size = 64
- Epochs = 100

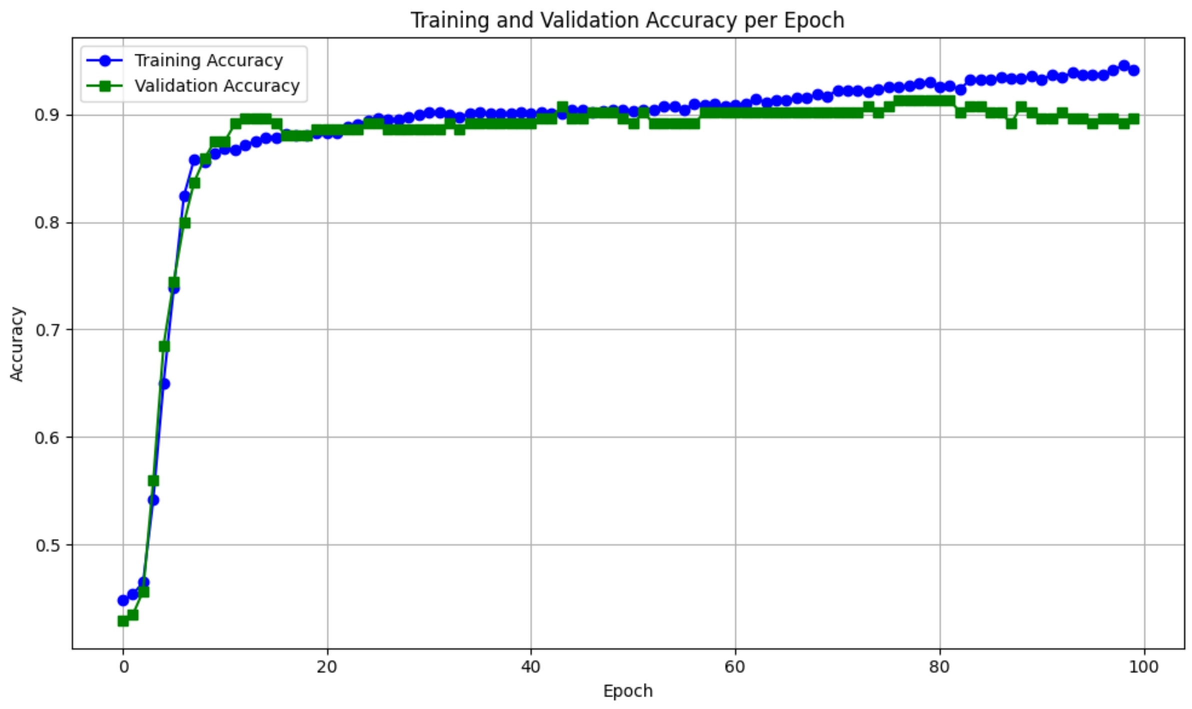
In [134…

```
model_new.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
history = model_new.fit(x_train_new, y_train_new, batch_size = 64, epochs =
12/12 [==============================] - 0s 6ms/step - loss: 0.1569 - accura
cy: 0.9455 - val_loss: 0.3926 - val_accuracy: 0.8913
Epoch 100/100
12/12 [==============================] - 0s 6ms/step - loss: 0.1555 - accura
cy: 0.9414 - val_loss: 0.3814 - val_accuracy: 0.8967
```

Analysing the model

In [135…

```python
import matplotlib.pyplot as plt
val_acc = history.history['val_accuracy']  # Assuming the metric is named 'v
plt.figure(figsize=(10, 6))

# Plot training accuracy
plt.plot(range(len(history.history['accuracy'])), history.history['accuracy'

# Plot validation accuracy
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy', marker='

# Add labels, title, legend, grid, etc.
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy per Epoch')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



As observed from the graph above, the model has performed exceptionally well on the new dataset. Instead of the accuracies of the training set and validation set diverging by a large margin, both of the accuracies improved per epoch. While the model slightly overfits to the training set, it has generalised well to the validation set with an accuracy of almost 90%

Optimising the new model using Genetic Algorithm

Like we did in our second approach, we are going to try to optimise this model using Genetic Algorithm by trying to find the optimum number of neurons per layer. In future practice, we can use genetic algorithm to tune the other hyperparameters as well.

In [136…

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from deap import base, creator, tools

# Define evaluation function
def evaluate(individual):
    # Decode the individual to obtain the architecture
    n_neurons_hidden1, n_neurons_hidden2, n_neurons_hidden3 = individual
    # Construct the neural network
    model = Sequential([
        tf.keras.Input(shape=(n_new,), name="input_layer"),
        Dense(units=n_neurons_hidden1, activation="relu", name="hidden_layer
        Dense(units=n_neurons_hidden2, activation="relu", name="hidden_layer
        Dense(units=n_neurons_hidden3, activation="relu", name="hidden_layer
        Dense(units=1, activation="sigmoid", name="output_layer"),
    ])
    # Compile the model
    model.compile(optimizer=Adam(learning_rate=0.01), loss="binary_crossentr
    # Train the model
    history = model.fit(x_train_new, y_train_new, batch_size=128, epochs=50,
    # Evaluate the model
    loss, accuracy = model.evaluate(x_test_new, y_test_new, verbose=0)
    # Return negative accuracy as fitness (maximization problem)
    return accuracy,

# Define genetic algorithm parameters
POP_SIZE = 8
NUM_GENERATIONS = 3
CROSSOVER_PROB = 1.0
MUTATION_PROB = 0.2

# Create a toolbox for the genetic algorithm
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("attr_int", np.random.randint, low=1, high=100)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=1, up=100, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Generate initial population
```

```python
population = toolbox.population(n=POP_SIZE)

# Run the genetic algorithm
for gen in range(NUM_GENERATIONS):
    print("Generation", gen + 1)

    # Evaluate the fitness of the population
    fitnesses = list(map(toolbox.evaluate, population))
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    # Print the parents' values and fitness values
    print("Parents:")
    for ind in population:
        print("Individual:", ind, "Fitness:", ind.fitness.values[0])

    # Select parents for reproduction
    offspring = toolbox.select(population, len(population))
    offspring = list(map(toolbox.clone, offspring))

    # Apply crossover and mutation
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if np.random.rand() < CROSSOVER_PROB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:
        if np.random.rand() < MUTATION_PROB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Print the children's values
    print("Children:")
    for ind in offspring:
        print("Individual:", ind)

    # Replace the population with the offspring
    population[:] = offspring

# Get the best individual
best_individual = tools.selBest(population, k=1)[0]
print("Best individual:", best_individual)
```

```
/opt/conda/lib/python3.10/site-packages/deap/creator.py:185: RuntimeWarning:
A class named 'FitnessMax' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
/opt/conda/lib/python3.10/site-packages/deap/creator.py:185: RuntimeWarning:
A class named 'Individual' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
Generation 1
Parents:
Individual: [33, 62, 81] Fitness: 0.8369565010070801
Individual: [63, 50, 53] Fitness: 0.842391312122345
Individual: [51, 20, 60] Fitness: 0.8804348111152649
Individual: [42, 34, 8] Fitness: 0.820652186870575
Individual: [9, 76, 3] Fitness: 0.8478260636329651
Individual: [4, 64, 3] Fitness: 0.8478260636329651
Individual: [15, 23, 89] Fitness: 0.842391312122345
Individual: [29, 1, 41] Fitness: 0.85326087474823
Children:
Individual: [9, 76, 3]
Individual: [9, 76, 3]
Individual: [4, 64, 60]
Individual: [51, 20, 3]
Individual: [15, 23, 89]
Individual: [15, 23, 89]
Individual: [51, 1, 60]
Individual: [29, 20, 41]
Generation 2
Parents:
Individual: [9, 76, 3] Fitness: 0.8478260636329651
Individual: [9, 76, 3] Fitness: 0.804347813129425
Individual: [4, 64, 60] Fitness: 0.85326087474823
Individual: [51, 20, 3] Fitness: 0.8369565010070801
Individual: [15, 23, 89] Fitness: 0.8260869383811951
Individual: [15, 23, 89] Fitness: 0.820652186870575
Individual: [51, 1, 60] Fitness: 0.875
Individual: [29, 20, 41] Fitness: 0.8586956262588501
Children:
Individual: [4, 64, 60]
Individual: [51, 1, 60]
Individual: [39, 64, 60]
Individual: [4, 20, 41]
Individual: [51, 64, 60]
Individual: [4, 1, 60]
Individual: [9, 76, 3]
Individual: [9, 76, 3]
Generation 3
Parents:
Individual: [4, 64, 60] Fitness: 0.820652186870575
Individual: [51, 1, 60] Fitness: 0.8260869383811951
Individual: [39, 64, 60] Fitness: 0.8152173757553101
Individual: [4, 20, 41] Fitness: 0.820652186870575
Individual: [51, 64, 60] Fitness: 0.8369565010070801
Individual: [4, 1, 60] Fitness: 0.58152174949646
Individual: [9, 76, 3] Fitness: 0.8695651888847351
Individual: [9, 76, 3] Fitness: 0.820652186870575
Children:
Individual: [9, 76, 3]
Individual: [9, 76, 3]
Individual: [9, 76, 3]
Individual: [9, 76, 3]
Individual: [9, 76, 60]
Individual: [4, 64, 3]
Individual: [51, 1, 60]
Individual: [51, 64, 60]
Best individual: [9, 76, 3]
```

As we can see from training the model with this new dataset, the model performs much better. Both the validation and training accuracy increase per epoch. However, after about 40 epochs, the model starts overfitting to the training data and thus the validation and training accuracy starts diverging. This was trained only for 100 epochs and yet it performed better, so from this we can conclude that the model failed to learn the patterns from the previous dataset is because the dataset is of low quality. The genetic algorithm is likely to improve the model a lot more if a it is run for more generations with a bigger population size and more hyperparameters are tested.

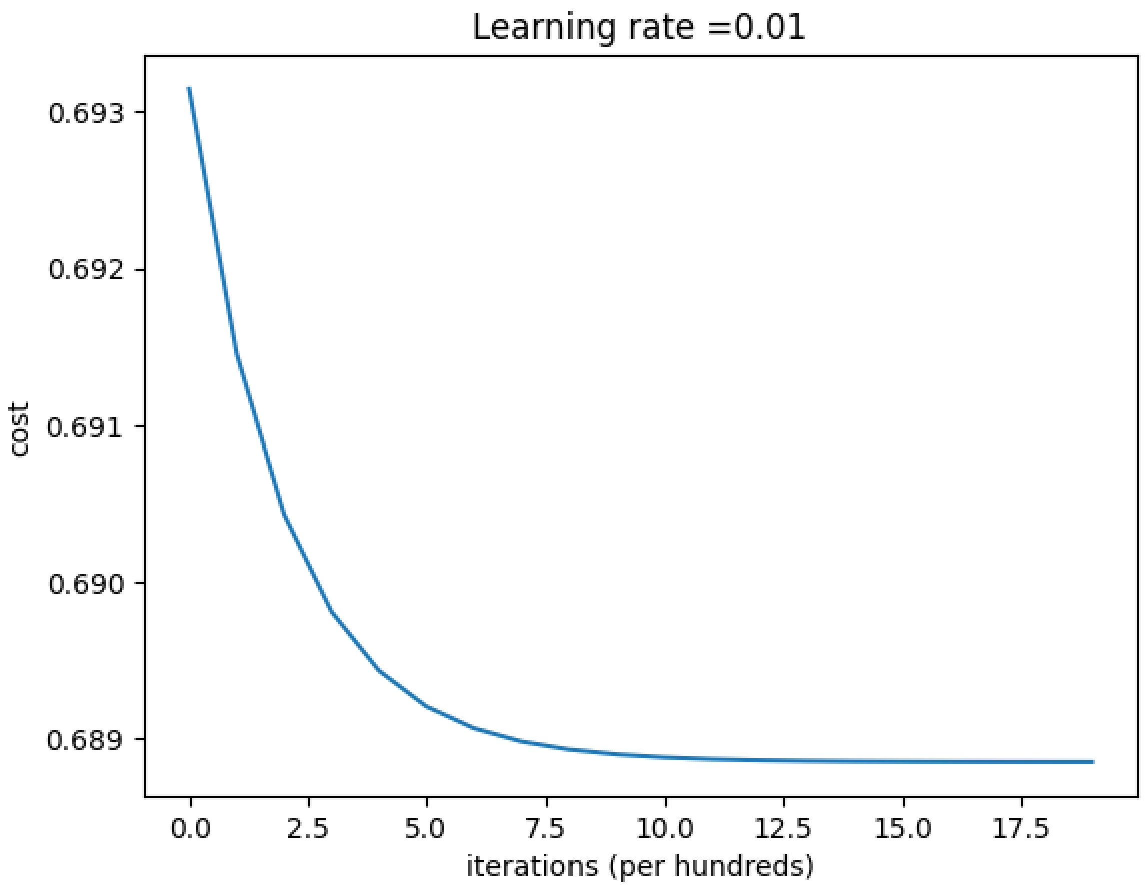
Training the Neural Network coded from scratch on the new dataset

We've seen our tensorflow model perform much better on the new dataset. Now we will see how our neural network used in the first approach performs.

In [137…

```
### Layers ###
layers_dims_new = [20, 32, 16, 8, 1] #Specify the number of neurons in each
#Train the model
parameters_new, costs_new = L_layer_model(x_train_new.T, y_train_new.T, laye
#Plot the graph of cost against iteration
plot_costs(costs_new, learning_rate=0.01)
```

```
Cost after iteration 0: 0.693147214548188
Cost after iteration 100: 0.6914567664160254
Cost after iteration 200: 0.6904319656431439
Cost after iteration 300: 0.689810436168662
Cost after iteration 400: 0.6894332885060412
Cost after iteration 500: 0.6892043143089308
Cost after iteration 600: 0.6890652336045588
Cost after iteration 700: 0.6889807202461612
Cost after iteration 800: 0.6889293473949845
Cost after iteration 900: 0.6888981105688945
Cost after iteration 1000: 0.6888791127032516
Cost after iteration 1100: 0.6888675561291318
Cost after iteration 1200: 0.6888605249445937
Cost after iteration 1300: 0.6888562463840541
Cost after iteration 1400: 0.6888536423801755
Cost after iteration 1500: 0.6888520572300743
Cost after iteration 1600: 0.6888510920706812
Cost after iteration 1700: 0.6888505041682214
Cost after iteration 1800: 0.6888501458461407
Cost after iteration 1900: 0.6888499272704893
Cost after iteration 1999: 0.6888497947752762
```



As we can see from the results, the cost tapers off and stops decreasing after some iterations. This is most likely because of a gradient vanishing problem where the gradient descent is stuck in a local minima. Since we did not use any optimisation such as momentum for this network, the model is not escaping the local minima. It is possible that if we run the model for a lot more iterations, it will be able to break out from this local minimum and fall into the global basin.

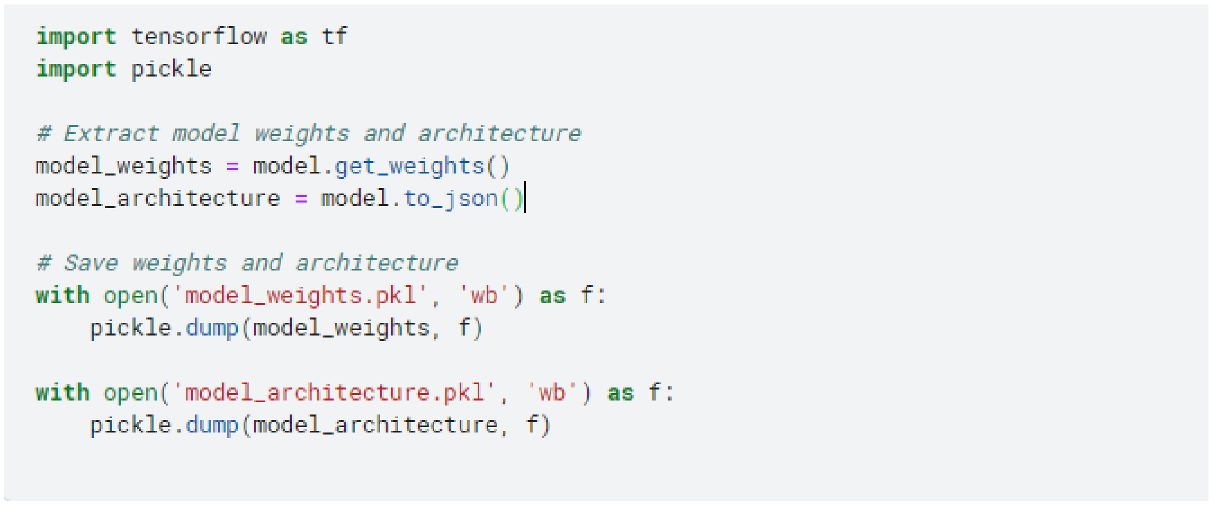
In [138…

```
#predicting on the train set
predictions_train_new = predict(x_train_new.T, y_train_new.T, parameters_new
#predicting on the test set
predictions_test_new = predict(x_test_new.T, y_test_new.T, parameters_new)
```

```
Accuracy: 0.5463215258855586
Accuracy: 0.5815217391304348
```

8. Building a GUI

We are going to use the tensorflow model we developed in the third approach to build a very simple GUI using the tkinter library. To simplify the process to build the GUI, we will train the model on only 4 features of the dataset: Age, Cholesterol, Max Heart Rate and Gender. We chose these four as they are more likely factors that can be linked to the risk of a heart disease. After training the model on those four features, we export the architecture and weights of the model as pickle files as done in the following code:



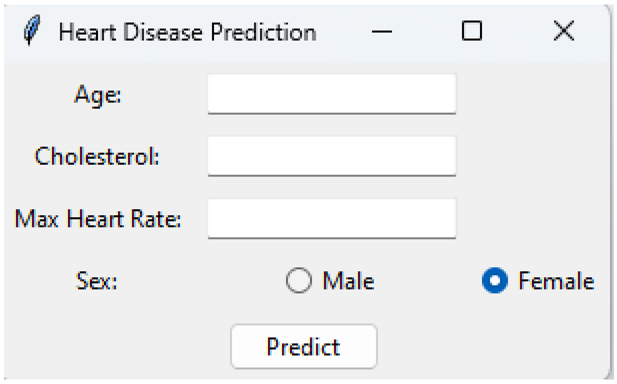
```python
import tensorflow as tf
import pickle

# Extract model weights and architecture
model_weights = model.get_weights()
model_architecture = model.to_json()

# Save weights and architecture
with open('model_weights.pkl', 'wb') as f:
    pickle.dump(model_weights, f)

with open('model_architecture.pkl', 'wb') as f:
    pickle.dump(model_architecture, f)
```

Then, the following code is used to create the basic GUI:

```python
import tkinter as tk
from tkinter import ttk
import tkinter.messagebox as messagebox
import tensorflow as tf
import numpy as np
import pickle

# Load model architecture
with open('model_architecture.pkl', 'rb') as f:
   model_architecture = pickle.load(f)

# Load model weights
with open('model_weights.pkl', 'rb') as f:
   model_weights = pickle.load(f)

# Reconstruct model
model = tf.keras.models.model_from_json(model_architecture)
model.set_weights(model_weights)

# Function to preprocess input data
def preprocess(age, cholesterol, max_heart_rate, sex):
   sex_encoded = [1 if sex == 'M' else 0, 1 if sex == 'F' else 0]
   return np.array([[age, cholesterol, max_heart_rate, *sex_encoded]],
dtype=np.float32)

# Function to make predictions
def predict(age, cholesterol, max_heart_rate, sex):
   input_data = preprocess(age, cholesterol, max_heart_rate, sex)
   prediction = model.predict(input_data)
   return prediction[0][0]

# Function to handle button click event
def predict_button_clicked():
   age = float(entry_age.get())
   cholesterol = float(entry_cholesterol.get())
   max_heart_rate = float(entry_max_heart_rate.get())
   sex = 'M' if var_sex.get() == 1 else 'F'

   prediction = predict(age, cholesterol, max_heart_rate, sex)

   messagebox.showinfo("Prediction", f"Probability of heart disease:
{prediction}")

# Create the main window
root = tk.Tk()
root.title("Heart Disease Prediction")

# Create input fields
label_age = ttk.Label(root, text="Age:")
label_age.grid(row=0, column=0, padx=5, pady=5)
entry_age = ttk.Entry(root)
entry_age.grid(row=0, column=1, padx=5, pady=5)

label_cholesterol = ttk.Label(root, text="Cholesterol:")
label_cholesterol.grid(row=1, column=0, padx=5, pady=5)
entry_cholesterol = ttk.Entry(root)
entry_cholesterol.grid(row=1, column=1, padx=5, pady=5)

label_max_heart_rate = ttk.Label(root, text="Max Heart Rate:")
label_max_heart_rate.grid(row=2, column=0, padx=5, pady=5)
entry_max_heart_rate = ttk.Entry(root)
entry_max_heart_rate.grid(row=2, column=1, padx=5, pady=5)

label_sex = ttk.Label(root, text="Sex:")
label_sex.grid(row=3, column=0, padx=5, pady=5)
var_sex = tk.IntVar()
radio_male = ttk.Radiobutton(root, text="Male", variable=var_sex, value=1)
radio_male.grid(row=3, column=1, padx=5, pady=5)
radio_female = ttk.Radiobutton(root, text="Female", variable=var_sex, value=0)
radio_female.grid(row=3, column=2, padx=5, pady=5)
```

```
# Create prediction button
button_predict = ttk.Button(root, text="Predict",
command=predict_button_clicked)
button_predict.grid(row=4, column=0, columnspan=3, padx=5, pady=5)

# Run the application
root.mainloop()
```

This is the GUI that is created:



Heart Disease Prediction

Age:

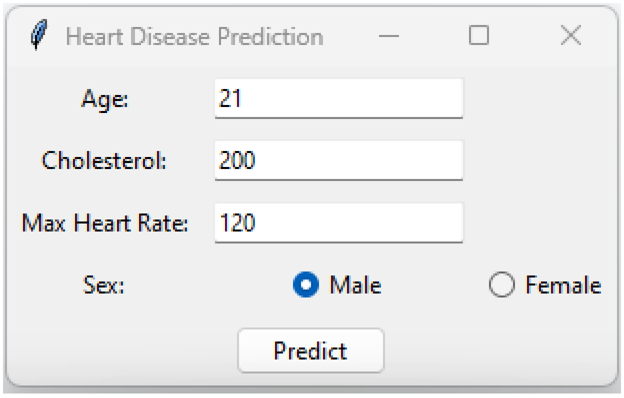
Cholesterol:

Max Heart Rate:

Sex: Male [✓ Female]

Predict

Once the user inputs the relevant values, their probability of getting a heart disease is predicted:



Heart Disease Prediction

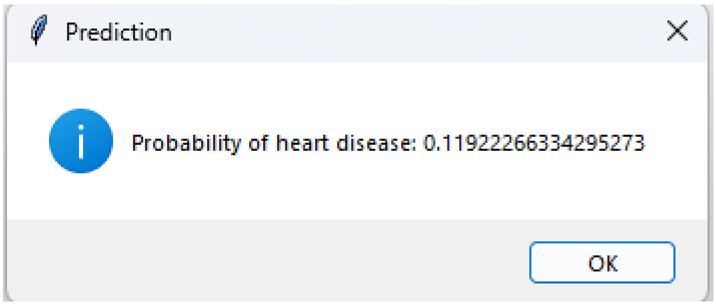
Age: 21

Cholesterol: 200

Max Heart Rate: 120

Sex: [✓ Male] Female

Predict



Prediction

Probability of heart disease: 0.11922266334295273

OK

Challenges

As observed from our working, the dataset we had to work with was a synthetic dataset made using ChatGPT. This made it quite challenging to develop a good model. We overcame this challenge by using another dataset to train and test our model.

Future Improvements

Development of AI is an iterative process, thus there is always opportunities to improve a model. Firstly, we can combine data from various sources to train the neural network. Generally, a larger dataset leads to a better model. Secondly, we can run the Genetic Algorithm for more generations with a bigger population size to tune the hyperparameters of the neural network. Moreover, we can combine the neural network with Fuzzy Logic and create an ANFIS to potentially get a more accurate model. Lastly, we can use libraries such as Streamlit or Gradio to build a user friendly GUI.

Conclusion

In conclusion, This project provided a hands-on exploration of artificial neural network development. We began by implementing a neural network from scratch, gaining a foundational understanding of its inner workings. Subsequently, we leveraged the TensorFlow library to construct and fine-tune models with greater efficiency and optimization.

We also explored the use of Genetic Algorithms to tune the hyperparameters and find a optimum combination for a highly accurate model. This project opens avenues for further investigation into advanced deep learning architectures or the potential impact of different data preprocessing techniques on model performance. Beyond heart disease prediction, this study highlights the value of combining fundamental understanding with powerful libraries and optimization techniques for effective machine learning development.

Appendice

1. The jupyter notebook for our project can be found here: Heart Attack Risk - ANN from scratch using np
2. GitHub: https://github.com/muqtasid87/HeartDiseaseRisk_Project
3. Our presentation slides can be found here: https://www.canva.com/design/DAF8r0XKbgA/iB0RIsaGuc-L3fnc0aYVXQ/edit?utm_content=DAF8r0XKbgA&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton
4. Original dataset: https://www.kaggle.com/datasets/iamsouravbanerjee/heart-attack-prediction-dataset
5. Alternative dataset used in third approach: https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction

References

1. https://github.com/dennybritz/nn-from-scratch
2. https://www.coursera.org/learn/neural-networks-deep-learning/home/week/4

Student's Declaration

This is to certify that we are responsible for the work submitted in this report, that the original work is our own except as specified in the references and acknowledgement, and that the original work contained herein have not been untaken or done by unspecified sources or persons.
We hereby certify that this report has not been done by only one individual and all of us have contributed to the report. The length of contribution to the reports by each individual is noted within this certificate.
We also hereby certify that we have read and understand the content of the total report and no further improvement on the reports is needed from any of the individual's contributors to the report.
We therefore, agreed unanimously that this report shall be submitted for marking and this final printed report has been verified by us.

Signature: *Muqtasid*	Read
Name: Tashfin Muqtasid	Understand
Matric Number: 2119609	Agree
Contribution: Tensorflow model, Genetic Algorithm and Parameters tuning	
Signature: *Nazim*	Read
Name: Muhammad Nazim Bin Akhmar	Understand
Matric Number: 2114551	Agree
Contribution : Neural Network from scratch using numpy	
Signature: *Adli*	Read
Name: Adli Hakim Zulkifli	Understand
Matric Number: 2110959	Agree
Contribution: Data cleaning and pre-processing	
Signature: *Norhezry*	Read
Name: Norhezry Hakimie bin Noor Fahmy	Understand
Matric Number:2110061	Agree
Contribution: Report	