# Experment-3: A MLP using keras with TensorFlow for classification problem

Aim : To implement a MLP using keras with TensorFlow for classification problem (heart disease predication).

```
#Importing Nessary Library and Module
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Setting Seaborn Style and Displaying TensorFlow Version
sns.set()
tf.__version__

# Load the dataset
data = pd.read_csv("/content/heart.csv")

# Separate features (X) and target variable (y)
X = data.drop('HeartDisease', axis=1)
y = data['HeartDisease']

# One-hot encode categorical variables
X = pd.get_dummies(X, columns=['Sex', 'ChestPainType', 'FastingBS', 'RestingECG', 'ExerciseAngina', 'ST_Slope'])

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Normalize the data using Min-Max scaling
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Build the neural network model with dropout
model = Sequential([
    Dense(300, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(150, activation='relu'),
    Dense(75, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Convert labels to 0 or 1 for binary classification
y_train_binary = y_train.astype('float32')

# Train the model
history = model.fit(
    X_train_scaled, y_train_binary, epochs=10,
    verbose=2
)

# Displaying the summary of the model
model.summary()

# Print layer information
print("List of layers:", model.layers)
print("\nName of the second layer:", model.layers[1].name)

# Accessing the second hidden layer and printing weights and bias
hidden2 = model.layers[2]
weights, bias = hidden2.get_weights()
print("Weights:", weights)
print("Bias:", bias)

# Plotting the training history
pd.DataFrame(history.history).plot()
plt.xlabel("Epoch")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2)
plt.show()

# Evaluate the model on the test set
loss, acc = model.evaluate(X_test_scaled, y_test)
print("Test Loss:", loss)
print("Test Accuracy:", acc)
```

## Code Explanation

### Importing Nessary Library and Module

```
#@title Importing Nessary Library and Module
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

### Setting Seaborn Style and Displaying TensorFlow Version

```
#@title Setting Seaborn Style and Displaying TensorFlow Version
sns.set()
tf.__version__
```

### Load the dataset

```
#@title Load the dataset
data = pd.read_csv("/content/heart.csv")
data.info()
```

### Separate features (X) and target variable (y)

```
#@title Separate features (X) and target variable (y)
X = data.drop('HeartDisease', axis=1)
y = data['HeartDisease']
```

### One-hot encode categorical variables

```
#@title One-hot encode categorical variables
X = pd.get_dummies(X, columns=['Sex', 'ChestPainType', 'FastingBS', 'RestingECG', 'ExerciseAngina', 'ST_Slope'])
X.head()
```

***pd.get_dummies():***

- This function is provided by the Pandas library and is used for **one-hot encoding** categorical variables. It converts categorical variable(s) into dummy/indicator variables.
- For each categorical column specified in the columns parameter, pd.get_dummies creates new binary columns (dummy variables) representing the unique values in those categorical columns.
- The original categorical columns are then dropped from the DataFrame.
- ***Example:*** If the original DataFrame X had a column 'Sex' with values 'Male' and 'Female', after applying pd.get_dummies, it would create two new columns 'Sex_Male' and 'Sex_Female' with binary values (0 or 1) indicating the presence of each category.

### Split the dataset into training and testing sets

```
#@title Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

### Normalize the data using Min-Max scaling

```
#@title Normalize the data using Min-Max scaling
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- Normalizing the input data helps in achieving faster convergence during the training of neural networks. It prevents certain features with larger scales from dominating the learning process and ensures that the model is not sensitive to the scale of input features.
- **Min-Max Scaling:** This is a specific type of normalization where the values are scaled to a specific range, typically between 0 and 1. Min-Max scaling is performed using the following formula: Xscaled = ( X - Xmin) / ( Xmax - Xmin )

## Build the neural network model with dropout

```
#@title Build the neural network model with dropout
model = Sequential([
    Dense(300, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(150, activation='relu'),
    Dense(75, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

1. **Sequential Model**: This line initializes a sequential model. In a sequential model, layers are added one by one in a linear stack.
2. **Dense Layers**: The model consists of four dense layers:
    - **First Layer (Input Layer)**:
        - Dense(300, activation='relu', input_shape=(X_train.shape[1],)):
            - 300 units in the layer.
            - activation='relu': Rectified Linear Unit (ReLU) activation function is used.
            - input_shape=(X_train.shape[1],): Specifies the shape of the input data. The input shape corresponds to the number of features in the training data.
    - **Second Layer**:
        - Dense(150, activation='relu'):
            - 150 units.
            - activation='relu': ReLU activation.
    - **Third Layer**:
        - Dense(75, activation='relu'):
            - 75 units.
            - activation='relu': ReLU activation.
    - **Output Layer**:
        - Dense(1, activation='sigmoid'):
            - 1 unit, as it's a binary classification task (sigmoid activation is commonly used for binary classification).
            - activation='sigmoid': Sigmoid activation function squashes the output between 0 and 1, making it suitable for binary classification.

## Compile the model

```
#@title Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

1. `optimizer='adam'` :
    - The optimizer is the algorithm used to update the weights of the neural network during training. 'Adam' is a popular optimization algorithm that combines the advantages of two other methods: RMSprop and Momentum.
    - `Adam` optimizes the learning rate during training.
2. `loss='binary_crossentropy'` :
    - The loss function (or objective function) is a measure of how well the model is performing. For binary classification problems, 'binary_crossentropy' is commonly used.
    - Binary Crossentropy is suitable when dealing with binary classification tasks (two classes).

- **Optimizer**: 'Adam' optimizer with adaptive learning rates.
- **Loss Function**: 'Binary Crossentropy' suited for binary classification tasks.
- **Metrics**: Monitoring the accuracy of the model during training and evaluation

## Convert labels to 0 or 1 for binary classification

```
#@title Convert labels to 0 or 1 for binary classification
y_train_binary = y_train.astype('float32')
```

**astype('float32')**: This method is used to change the data type of the target variable (y_train) to 'float32'. In binary classification, the target variable is often encoded as floats (0.0 and 1.0) instead of integers (0 and 1). This is done to ensure compatibility with the output layer's activation function, which is commonly set to 'sigmoid' for binary classification.

- 0 → 0.0
- 1 → 1.0

## Train the model

```
#@title Train the model
history = model.fit(
    X_train_scaled, y_train_binary, epochs=10,
    verbose=2
)
```

***history = model.fit(X_train_scaled, y_train_binary, epochs=10, verbose=0)*** : This line trains the model using the training data (X_train_scaled and y_train_binary). It specifies the number of training epochs (10) and sets verbose=0 to suppress training progress updates.

## Displaying the summary of the model

```
#@title Displaying the summary of the model
model.summary()
```

***model.summary()*** :This line prints a summary of the model's architecture, including the type of layers, the number of parameters, and the output shapes. The summary provides a concise overview of the neural network.

## Print layer information

```
#@title Print layer information
print("List of layers:", model.layers)
print("\nName of the second layer:", model.layers[1].name)
```

## Accessing the second hidden layer and printing weights and bias

```
#@title Accessing the second hidden layer and printing weights and bias
hidden2 = model.layers[2]
weights, bias = hidden2.get_weights()
print("Weights:", weights)
print("Bias:", bias)
```

## Plotting the training history

```
#@title Plotting the training history
pd.DataFrame(history.history).plot()
plt.xlabel("Epoch")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2)
plt.show()
```

## Evaluate the model on the test set

```
#@title Evaluate the model on the test set
loss, acc = model.evaluate(X_test_scaled, y_test)
print("Test Loss:", loss)
print("Test Accuracy:", acc)
```