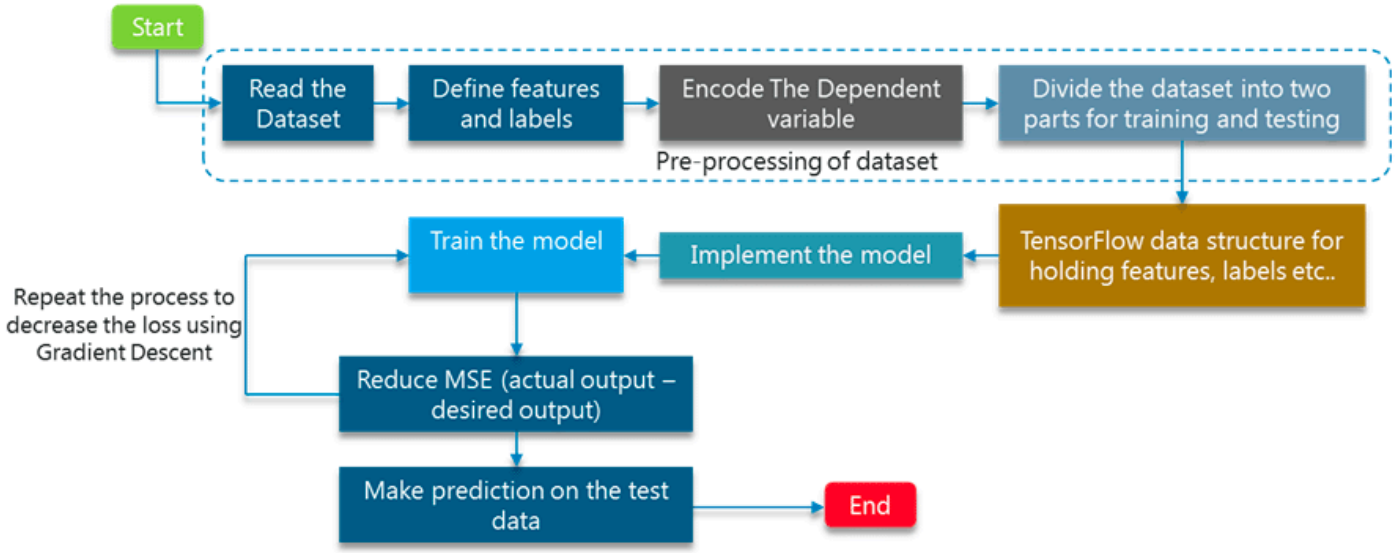
# Deep Learning

# Naval Mine Detector Case Study Applications WorkFlow

## Applications WorkFlow




## Step 1. Import all the required Libraries:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

Import all the required libraries as listed below:

- matplotlib library: It provides functions for plotting graph.
- tensorflow library: It provides functions for implementing Deep Learning Model.
- pandas, numpy and sklearn library: It provides functions for pre-processing the data.

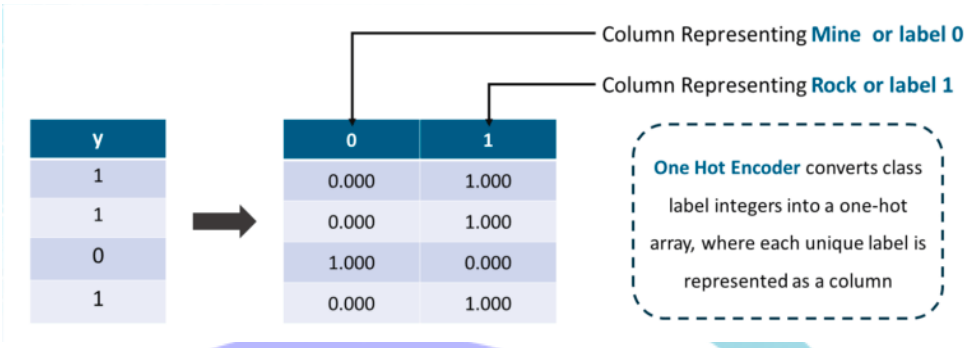
## Step 2. Read and Pre-process the data set:

```
def read_dataset():
    df = pd.read_csv("sonar.csv")
    print("Marvellous Infosystems : Dataset loaded succesfully")
    print("Marvellous Infosystems : Number of columns: ",len(df.columns))

    #Fetures of dataset
    X = df[df.columns[0:60]].values

    #Label of dataset
    y = df[df.columns[60]]

    # Encode the dependant variable
    encoder = LabelEncoder()

    # Encode character labeles into integer ie 1 or 0 (One hot encode)
    encoder.fit(y)
    y = encoder.transform(y)
    Y = one_hot_encode(y)

    print("Marvellous Infosystems : X.shape",X.shape)

    return (X,Y)
```

- At first we will read the CSV file (input data set) using read_csv() function
- Then, we will segregate the feature columns (independent variables) and the output column (dependent variable) as X and y respectively
- The output column consists of string categorical values as 'M' and 'R', signifying Rock and Mine respectively. So, We will label them them as 0 and 1 w.r.t. 'M' and 'R'
- After we have converted these categorical values into integer labels, we will apply one hot encoding using one_hot_encode() function that is discussed in the next step.

## 3. Function for One Hot Encoder:

One Hot Encoder adds extra columns based on number of labels present in the column. In this case, WE have two labels 0 and 1 (for Rock and Mine). Therefore, two extra columns will be added corresponding to each categorical value as shown in the image below:




```
# Define the encoder function to set M => 1, R => 0
def one_hot_encode(labels):
    n_labels = len(labels)
    n_unique_labels = len(np.unique(labels))
    one_hot_encode = np.zeros((n_labels,n_unique_labels))
    one_hot_encode[np.arange(n_labels),labels] = 1
    return one_hot_encode
```

## Step 4. Dividing data set into Training and Test Subset

```
# Convert the dataset into train and test datasets
# For  testing we use 30% and for training we use 70%
train_x, test_x, train_y, test_y = train_test_split(X, Y, test_size=0.30, random_state=415)
```

While working on any deep learning project, you need to divide your data set into two parts where one of the parts is used for training your deep learning model and the other is used for validating the model once it has been trained. Therefore, in this step We will also divide the data set into two subsets:

- Training Subset: It is used for training the model
- Test Subset: It is used for validating our trained model

We will be use train_test_split() function

## Step 5. Define Variables and Placeholders

Here, We will be define variables for following entities:

- Learning Rate: The amount by which the weight will be adjusted.

```
# Change in a variable in each iteration
learning_rate = 0.3
```

- Training Epochs: No. of iterations

```
# Total number of iterations to minimize the error
training_epochs = 1000
```

- Cost History: An array that stores the cost values in successive epochs.

```
cost_history = np.empty(shape=[1], dtype=float)
```

- Weight: Tensor variable for storing weight values

```
# define the weights and the biases for each layer
# Create variable which contens random values
weights = {
    'h1': tf.Variable(tf.random.truncated_normal([n_dim, n_hidden_1])),
    'h2': tf.Variable(tf.random.truncated_normal([n_hidden_1, n_hidden_2])),
    'h3': tf.Variable(tf.random.truncated_normal([n_hidden_2, n_hidden_3])),
    'h4': tf.Variable(tf.random.truncated_normal([n_hidden_3, n_hidden_4])),
    'out': tf.Variable(tf.random.truncated_normal([n_hidden_4, n_class])    ),
    }
```

- Bias: Tensor variable for storing bias values

```
# Create variable which contens random values
biases = {
    'b1': tf.Variable(tf.random.truncated_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random.truncated_normal([n_hidden_2])),
    'b3': tf.Variable(tf.random.truncated_normal([n_hidden_3])),
    'b4': tf.Variable(tf.random.truncated_normal([n_hidden_4])),
    'out': tf.Variable(tf.random.truncated_normal([n_class])),
    }
```

Apart from variable, We will also need placeholders that can take input. So, We will create place holder for our  input and feed it with the data set later on. At last, We will call global_variable_initializer() to initialize all the variables.

## Step 6. Calculate the Cost or Error

Similar to AND Gate implementation, We will calculate the cost or error produced by our model. Instead of Mean Squared Error, We will use cross entropy to calculate the error in this case.

```
# Initialization of variables
init = tf.compat.v1.global_variables_initializer()

saver = tf.compat.v1.train.Saver()

# Call to model function for training
y = multilayer_perceptron(x, weights, biases)

# Define the cost function to calculate loss
cost_function = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_))

# Function to reduce loss
training_step = tf.compat.v1.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)
```

## Step 7. Training the Perceptron Model in Successive Epochs

Now, we will train our model in successive epochs.
In each of the epochs, the cost is calculated and then, based on this cost the optimizer modifies the weight and bias variables in order to minimize the error.

```
# Calculate the cost and the accuracy for each epoch
for epoch in range(training_epochs):
    sess.run(training_step, feed_dict={x:train_x, y_:train_y})
    cost = sess.run(cost_function,feed_dict={x:train_x, y_:train_y})
    cost_history = np.append(cost_history, cost)
    correct_prediction = tf.equal(tf.argmax(y,1),tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    pred_y = sess.run(y,feed_dict={x:test_x} )
    mse = tf.reduce_mean(tf.square(pred_y - test_y))
    mse_ = sess.run(mse)
    accuracy = (sess.run(accuracy,feed_dict={x:train_x, y_:train_y}))
    accuracy_history.append(accuracy)
    print('epoch: ', epoch,' - ', 'cost: ', cost, " - MSE: ", mse_, "- Train Accuracy: ", accuracy)
```

## Step 8. Validation of the Model based on Test Subset

As discussed earlier, the accuracy of a trained model is calculated based on Test Subset. Therefore, at first, We will feed the test subset to our model and get the output (labels).

Then, We will compare the output obtained from the model with that of the actual or desired output and finally, will calculate the accuracy as percentage of correct predictions out of total predictions made on test subset.

```
# Print the final mean square error
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.square(pred_y - test_y))
print("Test Accuracy: ", (sess.run(y, feed_dict={x:test_x, y_:test_y} )))
```