PERIYAR MANIAMMAI
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University)
Established Under Sec. 3 of UGC Act, 1956 • NAAC Accredited
think • innovate • transform

RECORD NOTE BOOK

COURSE NAME	:	
COURSE CODE	:	
STUDENT NAME	:	
REGISTER NUMBER	:	
BRANCH	:	
YEAR	:	
SEMESTER	:	

PERIYAR MANIAMMAI INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University)
Established Under Sec. 3 of UGC Act, 1956 • NAAC Accredited
think • innovate • transform

CERTIFICATE

This is to certify that Mr. ______________________________

a student of ______________________________

bearing the register number ______________ has been completed the record notebook

for the course ______________________________

as per the curriculum during the academic year 2024 – 2025. The work submitted In this notebook has been verified and found to be satisfactory.

Year: III Semester : VI

Laboratory/Course in-Charge
(With date)
Name:

Head of the Department
(with date and seal)

Sumbitted for the Practical Examination held on ______________

Internal Examiner
(with Date)
Name:

External Examiner
(with Date)
Name:

INDEX

Exp 01: Building And Programming of Teleoperation Differential Drive Robot Virtually

Aim:

Construct a differential robot using primitive shapes and program it to demonstrate fundamental movements.

Description :

Differential Drive Robot: A differential drive robot is a type of mobile robot that moves by varying the speeds of its two drive wheels. Here's how it works:

- **Drive System:** The robot is typically driven by two wheels mounted on either side of the robot's body. Each wheel can be independently controlled, allowing the robot to move forward, backward, or turn in place.
- **Differential Steering:** To move forward, both wheels rotate in the same direction at the same speed. To turn, the wheels rotate in opposite directions, with the wheel on the side of the turn moving slower than the wheel on the other side.
- **Speed Control:** By controlling the speed of each wheel independently, the robot can achieve different types of movements, such as straight-line motion, curves, and spins.
- **Limitations:** Differential drive robots can be prone to slipping, especially when turning on uneven or slippery surfaces. This can affect the accuracy of their movements.

Differential drive robots are commonly used in robotics education and research due to their simplicity and effectiveness in a wide range of applications. They are often used in tasks such as exploration, mapping, and surveillance.

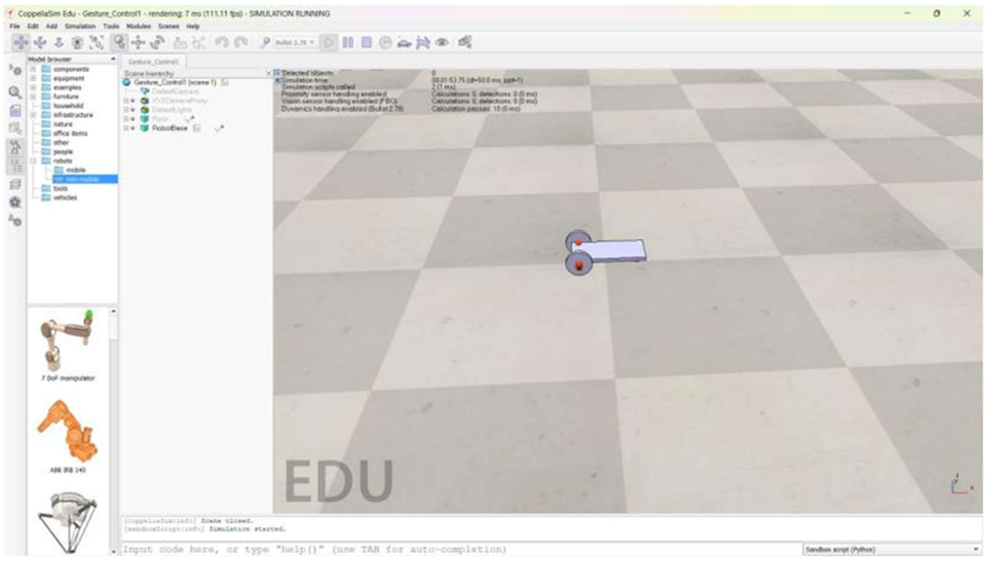
Algorithm:

- Import the required primitive shapes of size needed for the robot
- Add the revolute joint to the design as these are going to act as motors for the robot
- Build and position the caster wheel along with a force send
- Provide parent-child relationship to the components we have added
- Create a script for the built robot
- Open the script and initialize the motors
- Build the logic for the key press.
- Program it in LUA for the basic movements of the robot
- Play and check the simulation

Code Implementation:

```
function sysCall_init()
  RM=sim.getObject("../RM")
  LM=sim.getObject("../LM
end

function sysCall_actuation()
   message, data, data2 = sim.getSimulatorMessage()
   print(message, data)

      if(message==sim.message_keypress)then
      if(data[1]==2007)then
         -- Forward
         sim.setJointTargetVelocity(RM,-5)
         sim.setJointTargetVelocity(LM,-5)
      end

      if(data[1]==2008)then
         -- Backward
         sim.setJointTargetVelocity(RM,5)
         sim.setJointTargetVelocity(LM,5)
      end

      if(data[1]==2009)then
         -- LeftTurn
         sim.setJointTargetVelocity(RM,-5)
         sim.setJointTargetVelocity(LM,5)
      end

      if(data[1]==2010)then
         -- RightTurn
         sim.setJointTargetVelocity(RM,5)
         sim.setJointTargetVelocity(LM,-5)
      end

      else
         -- Stop
         sim.setJointTargetVelocity(RM,0)
         sim.setJointTargetVelocity(LM,0)
      end

end

function sysCall_sensing()
   -- put your sensing code here
end

function sysCall_cleanup()
   -- do some clean-up here
end

-- See the user manual or the available code snippets for additional callback functions and details
```

Output:



```
Selected objects:                               0
Simulation time:                                00:01:53.75 (dt=50.0 ms, ppf=1)
Simulation scripts called                       2 (1 ms)
Proximity sensor handling enabled               Calculations: 0, detections: 0 (0 ms)
Vision sensor handling enabled (FBO)            Calculations: 0, detections: 0 (0 ms)
Dynamics handling enabled (Bullet 2.78)         Calculation passes: 10 (0 ms)
```

```
[CoppeliaSim:info] Scene closed.
[sandboxScript:info] Simulation started.
Input code here, or type "help()" (use TAB for auto-completion)
```

Image Source - **Differential** Drive Robot using primitive shapes in Coppeliasim

Result:

The construction of the differential drive robot is completed, and the teleoperation via keyboard key press is programmed and validated through simulation in the environment.

Exp 02: Fundamentals Of Arduino

Aim:

To grasp the fundamentals of Arduino, program it using the IDE, and upload the code to verify its functionality.

Software & Hardware Required:

- Arduino UNO Board
- USB Cable
- Arduino IDE Software

Description:

Arduino: Arduino is an open-source electronics platform based on easy-to-use hardware and software.
Here are its key fundamentals:

- **Microcontroller:** Arduino boards are equipped with microcontrollers (usually from the Atmel AVR or ARM Cortex families) that act as the brains of the board. These microcontrollers can be programmed to perform a variety of tasks.
- **Arduino IDE:** The Arduino Integrated Development Environment (IDE) is a software application used to write and upload code to the Arduino board. It provides a simple interface for writing code in the Arduino programming language, which is based on C and C++.
- **Input/Output Pins:** Arduino boards have a number of digital and analog input/output pins that can be used to interface with external sensors, actuators, and other electronic components. These pins can be configured as inputs or outputs in software.
- **Power Supply:** Arduino boards can be powered via USB, a DC power jack, or an external power source. They typically operate at 5V, but some boards also support 3.3V operation.
- **Expansion Shields:** Arduino boards can be expanded using shields, which are additional circuit boards that plug into the main Arduino board. Shields can add functionality such as wireless communication, motor control, and sensor interfaces.
- **Programming Language:** Arduino uses its own programming language, which is similar to C and C++. It provides a set of libraries and functions that simplify common tasks such as reading analog sensors, controlling servos, and communicating over serial.

Overall, Arduino is a versatile platform that can be used for a wide range of projects, from simple blinking LED experiments to complex robotics and automation applications. Its ease of use, affordability, and large ecosystem of compatible hardware and software make it a popular choice for hobbyists, students, and professionals alike.

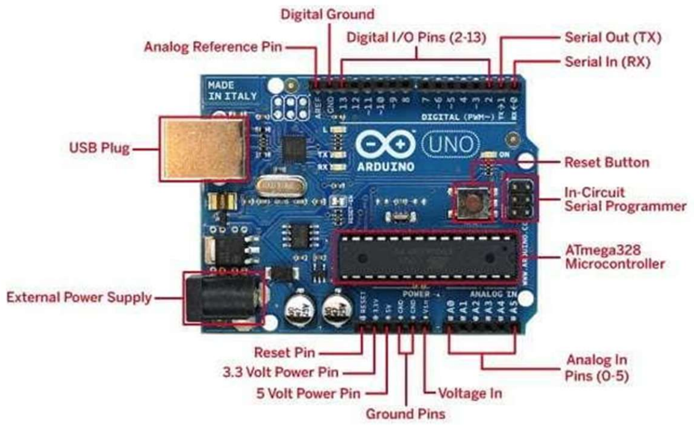


Image Source - https://robosynckits.in/wp-content/uploads/2020/12/arduino-uno-r3-pic3.j

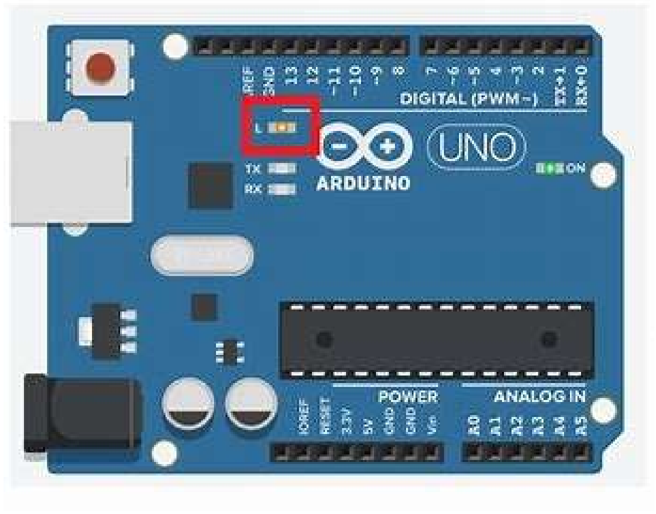
Steps:

- Download and install the Arduino IDE
- Get a new file
- Initialize the in-built LED
- Program the logic such that it blinks for one second
- Connect the Arduino UNO board to the computer
- Select the appropriate board and its corresponding port
- Upload the program
- Once upload is complete the in-built LED will blink

Code Implementation:

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT); // Initialize the in-built LED pin as an output
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // Turn the LED on
  delay(1000);                     // Wait for 1 second (1000 milliseconds)
  digitalWrite(LED_BUILTIN, LOW);  // Turn the LED off
  delay(1000);                     // Wait for 1 second
}
```

Output:



Result:

Therefore, the fundamental aspects of Arduino have been learned, and a basic program to blink the built-in LED has been developed, tested, and uploaded to the board for visualization.

Exp 03: Building an Obstacle Avoidance Robot

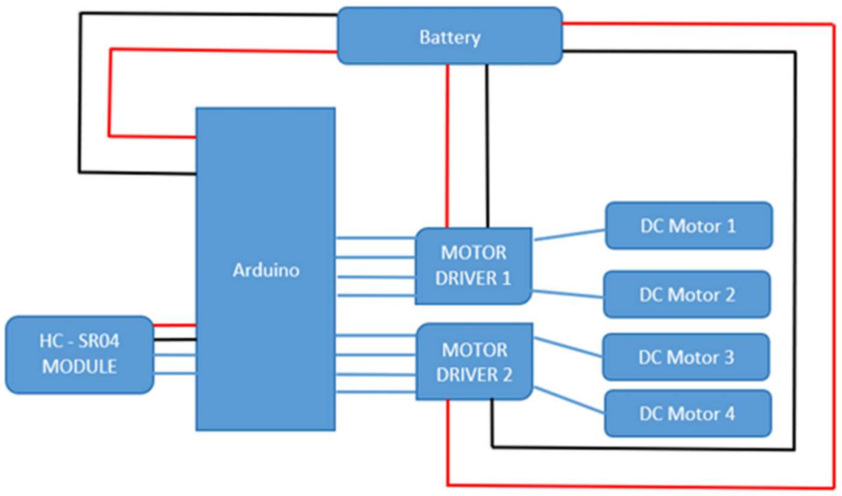
Aim:

To construct and program the robot to navigate an environment without collisions.

Software & Hardware Requirement:

- Google Colaboratory
- Arduino UNO Board
- USB Cable
- DC Motors
- Motor Driver
- HC-SR04 Module
- LiPo Battery
- Jumper Wires
- Robot Chassis
- Required Hardware
- Arduino IDE Software

Connection Diagram:



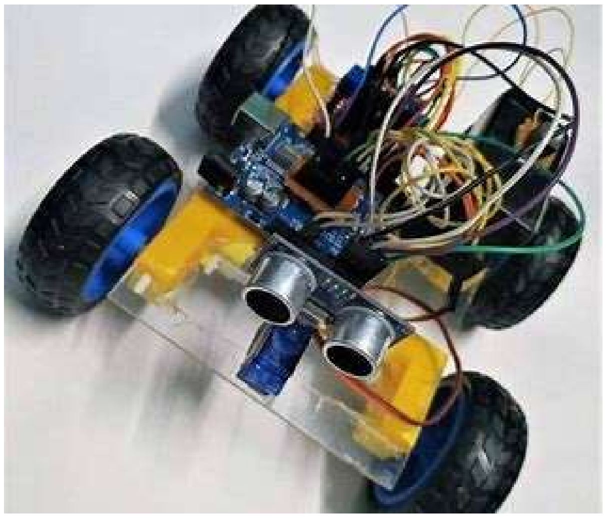
Steps:

- Build the robot using robot chassis and hardware
- Place the electronic components and wire them using jumper wires
- Get a new file
- Program the logic such that when a certain distance is reached from the sensor the robot has to stop or take a right turn.
- Once no obstacle is present the robot has to move in the way
- Connect the Arduino UNO board to the computer
- Select the appropriate board and its corresponding port
- Upload the program
- Once uploaded, connect the battery to turn on
- Now the robot moves and if a hand or object is kept in front of sensor and gets detected then the stop or right turn will happen
- The system is continuing and will stop only when the robot is switched off

Code Implementation:

```
// Motor Driver 1 Pins
#define IN1 2
#define IN2 3
#define IN3 4
#define IN4 5

// Motor Driver 2 Pins
#define IN5 6
#define IN6 7
#define IN7 8
#define IN8 9

// Ultrasonic Sensor Pins
#define trigPin 10
#define echoPin 11

// Distance threshold (in cm)
int distanceThreshold = 20;

void setup() {
  // Set motor pins as output
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(IN5, OUTPUT);
  pinMode(IN6, OUTPUT);
  pinMode(IN7, OUTPUT);
  pinMode(IN8, OUTPUT);
  // Set ultrasonic pins
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);

  // Begin Serial (optional, for debugging)
  Serial.begin(9600);
}

void loop() {
  int distance = readDistance();

  if (distance > 0 && distance < distanceThreshold) {
    // Obstacle detected
    stopRobot();
    delay(500);       // Pause
    turnRight();
    delay(600);        // Adjust turning time
  } else {
    // No obstacle
    moveForward();
  }
}

// Function to read distance from ultrasonic sensor
int readDistance() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  long duration = pulseIn(echoPin, HIGH);
  int distance = duration * 0.034 / 2;  // cm

  Serial.print("Distance: ");
  Serial.println(distance);

  return distance;
}

// Functions for Motor Movement
void moveForward() {
  // Motor Driver 1
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);

  // Motor Driver 2
  digitalWrite(IN5, HIGH);
  digitalWrite(IN6, LOW);
  digitalWrite(IN7, HIGH);
  digitalWrite(IN8, LOW);
}

void turnRight() {
  // Motor Driver 1
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);

  // Motor Driver 2
  digitalWrite(IN5, LOW);
  digitalWrite(IN6, HIGH);
  digitalWrite(IN7, HIGH);
  digitalWrite(IN8, LOW);
}

void stopRobot() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);

  digitalWrite(IN5, LOW);
  digitalWrite(IN6, LOW);
  digitalWrite(IN7, LOW);
  digitalWrite(IN8, LOW);
}
```

Output:



Result:

The robot is constructed with sensors and programmed to detect obstacles when they reach a certain distance. Finally, the robot is executed to move around without collisions.

Exp 04: Build A Simple Neural Network

Aim:

Constructing a simple neural network in TensorFlow to execute logistic regression and categorize the output.

Software Requirement:

- Google Colaboratory

Description:

Logistic regression is a statistical method used for binary classification tasks, where the goal is to predict the probability that an instance belongs to a particular class (usually denoted as class 1).

Here's an explanation of logistic regression and how it's built in TensorFlow 2

Logistic Regression:

- **Model:**

  - In logistic regression, the output of the model is obtained by applying the logistic function (also known as the sigmoid function) to a linear combination of the input features:
  = (w1x1 + w2x2 + ... + wnxn + b)
  - Here, y represents the predicted probability that the instance belongs to class 1, x1, x2, ..., xn are the input features, w1, w2, ..., wn are the weights, b is the bias term, and si is the sigmoid function.

- **Sigmoid Function:**

  - The sigmoid function transforms the linear combination of inputs into a value between 0 and 1, representing the probability of belonging to class 1:

  sigma(z) = frac11 + e^{-z}

- **Training:**

  - During training, the model learns the optimal values of weights w and bias b by minimizing a loss function (e.g., binary cross-entropy loss) using optimization algorithms like gradient descent.

- **Prediction:**

  - Once trained, the model can predict the probability of belonging to class 1 for new instances by passing their features through the trained logistic regression model.

Algorithm

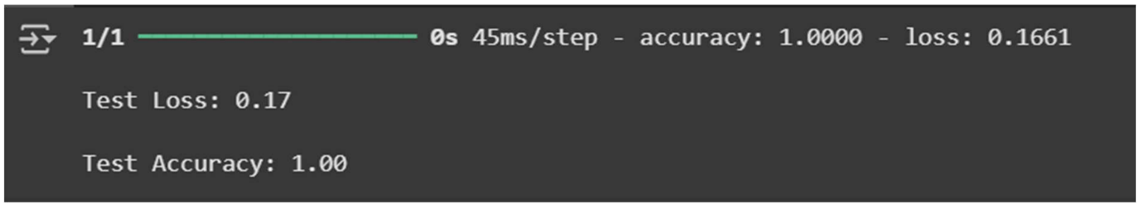
- Import necessary libraries and the dataset
- Create a TensorFlow sequential model and a sigmoid activation function
- Compile the model with appropriate optimizer and a suitable loss function
- Train the model using fit method y providing input and target
- Specify the number of epochs for training
- Predict using new data and evaluate the model's performance

Code Implementation:

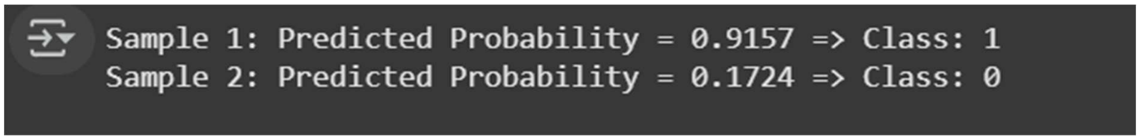
```
# Import necessary libraries
import tensorflow as tf
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels (0, 1, 2)

# Binary classification: Make it 0 (Setosa) vs 1 (Not Setosa)
y_binary = (y == 0).astype(int)  # Setosa -> 1, Others -> 0

# Split into training and testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y_binary, test_size=0.2, random_state=42
)

# Feature Scaling (important for faster training)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, activation='sigmoid', input_shape=(4,))
])

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

```
# Train the model
model.fit(X_train, y_train, epochs=100, verbose=1)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {accuracy:.2f}")

# Predict on some new data
sample_data = np.array([
    [5.1, 3.5, 1.4, 0.2],  # Setosa (should be 1)
    [6.7, 3.1, 4.4, 1.4]   # Not Setosa (should be 0)
])

sample_data = scaler.transform(sample_data)

predictions = model.predict(sample_data)

for i, pred in enumerate(predictions):
    print(f"Sample {i+1}: Predicted Probability = {pred[0]:.4f} => Class: {int(pred[0] > 0.5)}")
```

Output:



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step - accuracy: 1.0000 - loss: 0.1661

Test Loss: 0.17

Test Accuracy: 1.00
```



```
Sample 1: Predicted Probability = 0.9157 => Class: 1
Sample 2: Predicted Probability = 0.1724 => Class: 0
```

Result:

Thus, utilizing TensorFlow, a simple neural network is constructed to implement logistic regression with a dataset and classify the output.

Exp 05: Build A Multi-Layer Perceptron Model

Aim:
To Built a Multiple layer perceptron model in python using TensorFlow 2.

Software Requirement:

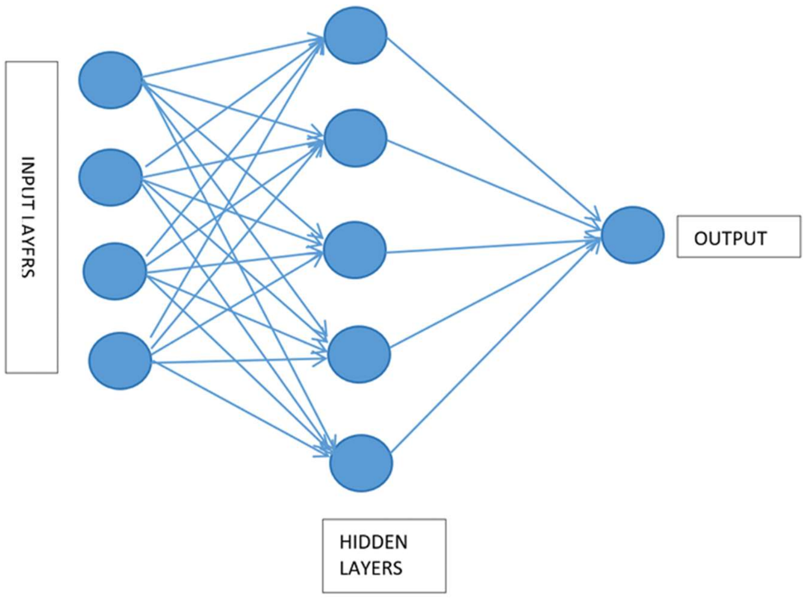
- Google Colaboratory

Description:

Multi-Layer Perceptron:

A multi-layer perceptron (MLP) is a type of feedforward artificial neural network that consists of multiple layers of nodes, each connected to the next in a sequential manner. MLPs are used for supervised learning tasks, including classification and regression.

- **Input Layer:** The input layer receives the features of the input data.
- **Hidden Layers:** One or more hidden layers process the input data through a series of weighted connections and activation functions. Each node in a hidden layer takes the weighted sum of its inputs, applies an activation function (such as ReLU or sigmoid), and passes the result to the next layer.
- **Output Layer:** The output layer produces the final predictions or outputs of the model. The number of nodes in the output layer depends on the task (e.g., binary classification, multi-class classification, regression).
- **Activation Functions:** Activation functions introduce non-linearity into the model, allowing it to learn complex patterns in the data. Common activation functions include ReLU, sigmoid, and tanh.
- **Training:** MLPs are trained using backpropagation, where the error between the predicted output and the actual output is calculated (using a loss function) and propagated back through the network to update the weights using an optimization algorithm (such as SGD or Adam).



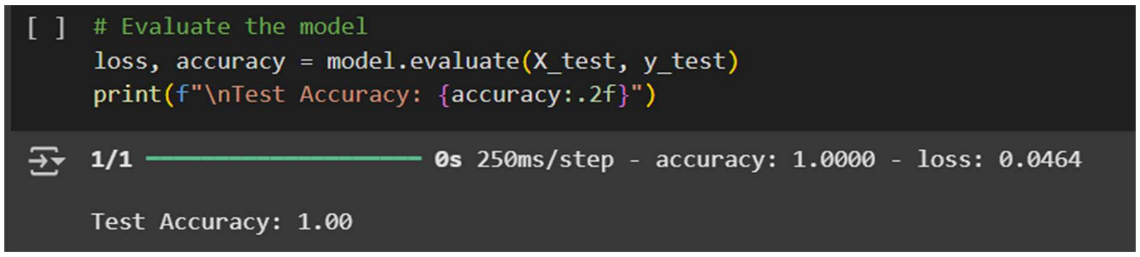
Algorithm:

- Import dataset
- Create the network architecture
- Build and compile the neural network
- Fit the values and train with set number of epochs and in batch.
- Predict the outcome with new values

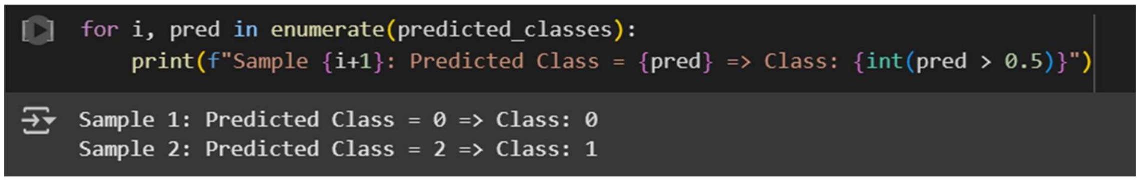
Code Implementation:

```
# Import necessary libraries
import tensorflow as tf
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
import numpy as np

# Load the Iris dataset
iris = load_iris()

X = iris.data  # Features
y = iris.target.reshape(-1, 1)  # Labels reshaped to column vector

# One-hot encode the labels (since there are 3 classes)
encoder = OneHotEncoder(sparse_output=False)

y_encoded = encoder.fit_transform(y)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42
)

# Scale features
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create the MLP model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(4,)),   # Hidden layer 1 with 10 neurons
    tf.keras.layers.Dense(8, activation='relu'),                      # Hidden layer 2 with 8 neurons
    tf.keras.layers.Dense(3, activation='softmax')                    # Output layer (3 classes)
])


# Compile the model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',  # Because we have multi-class classification
    metrics=['accuracy']
)

# Train (Fit) the model
model.fit(X_train, y_train, epochs=100, batch_size=8, verbose=1)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {accuracy:.2f}")

# Predict with new data
sample_data = np.array([
    [5.1, 3.5, 1.4, 0.2],  # Likely Setosa
    [6.2, 3.4, 5.4, 2.3]   # Likely Virginica
])

sample_data = scaler.transform(sample_data)  # Scale before prediction
predictions = model.predict(sample_data)

# Print predictions
predicted_classes = np.argmax(predictions, axis=1)

for i, pred in enumerate(predicted_classes):
    print(f"Sample {i+1}: Predicted Class = {pred} => Class: {int(pred > 0.5)}")
```

Output:



```
[ ] # Evaluate the model
    loss, accuracy = model.evaluate(X_test, y_test)
    print(f"\nTest Accuracy: {accuracy:.2f}")

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 250ms/step - accuracy: 1.0000 - loss: 0.0464

Test Accuracy: 1.00
```



```
for i, pred in enumerate(predicted_classes):
    print(f"Sample {i+1}: Predicted Class = {pred} => Class: {int(pred > 0.5)}")

Sample 1: Predicted Class = 0 => Class: 0
Sample 2: Predicted Class = 2 => Class: 1
```

Result:

A rudimentary multi-layer perceptron model is established with the Iris dataset, after which its evaluation score is computed.

Exp 06: Optimization Using Stochastic Gradient Descent Algorithm

Aim:
To implement stochastic gradient descent algorithm for optimization using colaboratory.

Software Requirement:

- Google Colaaboratory

Description:

Stochastic gradient descent (SGD) is an optimization algorithm commonly used for training machine learning models. Unlike batch gradient descent, which computes the gradient of the loss function using the entire dataset, SGD computes the gradient using a single randomly chosen data point (or a small subset of points) at each iteration. Here's how SGD works:

- **Initialization:** Start by initializing the model parameters (weights and biases) to some random values. Choose a Learning Rate: Select a learning rate (\ (\alpha \)), which controls the size of the step taken in the direction of the gradient.
- **Iterate Until Convergence:** Repeat the following steps until the algorithm converges:

  - Randomly shuffle the training data.
  - For each data point (x^(i), y^(i)) in the training data:
    - Compute the gradient of the loss function with respect to the parameters using only that data point:

$$\nabla J(\theta; x^{(i)}, y^{(i)})$$

  - Update the parameters using the computed gradient:

$$\theta = \theta - \alpha \cdot \nabla J(\theta; x^{(i)}, y^{(i)})$$

- **Stopping Criterion:** The algorithm stops when a stopping criterion is met, such as reaching a maximum number of iterations or when the change in the parameters is below a certain threshold.

SGD is particularly useful when working with large datasets, as it can be computationally more efficient than batch gradient descent. However, it can also be more sensitive to the learning rate and may require tuning of hyperparameters to achieve good performance. In practice, a variant of SGD called mini-batch gradient descent is often used, where the gradient is computed using a small batch of data points rather than a single point. This helps to reduce the variance in parameter updates while still being more computationally efficient than batch gradient descent.

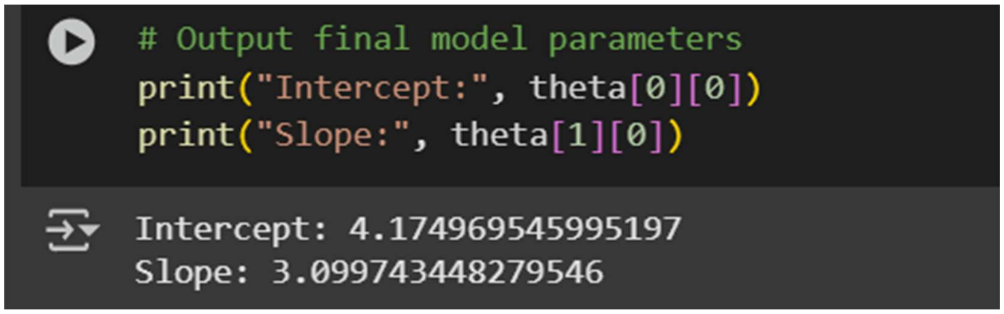
Algorithm:

- Initialize parameters
- Define model
- Set num of iterations, penalty
- Fit the values
- Get values of weights, intercept

Source Code :

```
import numpy as np

# Generate synthetic training data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add x0 = 1 to each instance for intercept term
X_b = np.c_[np.ones((100, 1)), X]

# Initialize parameters (theta) randomly
theta = np.random.randn(2, 1)

# Set hyperparameters
learning_rate = 0.01
n_epochs = 50
m = len(X_b)

# SGD Algorithm
for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        theta = theta - learning_rate * gradients

# Output final model parameters
print("Intercept:", theta[0][0])
print("Slope:", theta[1][0])
```

Output:



```
# Output final model parameters
print("Intercept:", theta[0][0])
print("Slope:", theta[1][0])
```

```
Intercept: 4.174969545995197
Slope: 3.099743448279546
```

Result:

Thus, a simple implementation of stochastic gradient descent is completed and executed.

Exp 07: Build An Image Recognition System Using CNN

Aim:
To Build a logistic regression model in tensorflow

Software Requirement:

- Google Colaboratory

Description:

CNN Model

A Convolutional Neural Network (CNN) is a type of deep learning model that is particularly effective for image recognition and classification tasks. CNNs are inspired by the organization of the animal visual cortex, and they have proven to be very successful in various computer vision tasks.
Here's how CNNs work and how they are built in TensorFlow 2:

Convolutional Layer:

- The core building block of a CNN is the convolutional layer.
- Each convolutional layer applies a set of filters (also called kernels) to the input image.
- These filters detect different features of the input, such as edges, textures, and shapes.
- The output of a convolutional layer is a set of feature maps, each representing the result of applying one filter to the input image.

Activation Function:

- After each convolution operation, an activation function like ReLU (Rectified Linear Unit) is applied element-wise to introduce non-linearity into the network.

Pooling Layers:

- Pooling layers are used to down sample the feature maps, reducing their spatial dimensions.
- Common pooling operations include max pooling and average pooling, which take the maximum or average value over a small region of the feature map, respectively.

Flattening:

- After several convolution and pooling layers, the remaining feature maps are flattened into a one dimensional vector.

Fully Connected Layers:

- The flattened vector is passed through one or more fully connected layers, which perform classification based on the learned features.
- The last fully connected layer typically has nodes corresponding to the number of classes in the classification task.

Output Layer:

- The final layer produces the output of the network, representing the predicted class probabilities for classification tasks.

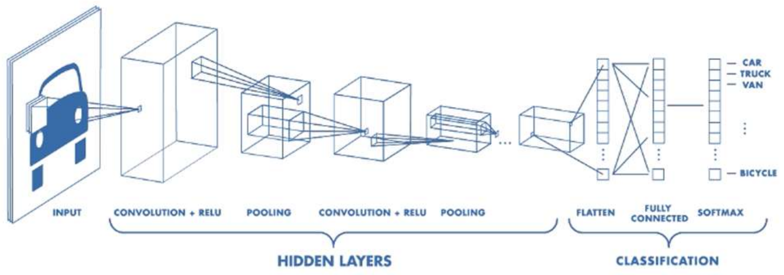


Image Source: https://www.mathworks.com/videos/introduction-to-deep-learning-what-are convolutional-neural-networks--1489512765771.html

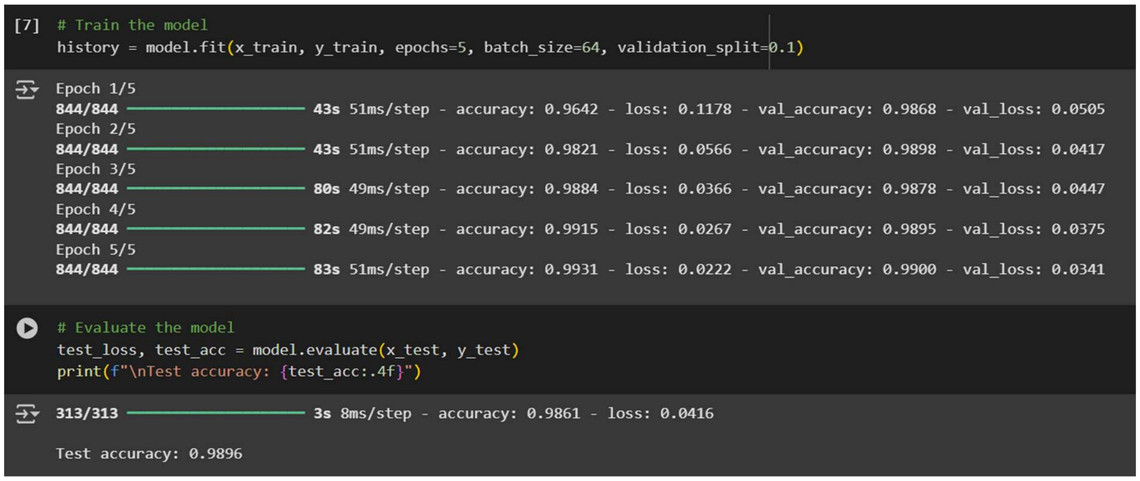
Algorithm:

- Import dataset
- Create the network model architecture
- Build the layers and compile the model
- Fit the values and train with set number of epochs and in batch.
- Predict the outcome using test data
- Check the performance metrics of the model

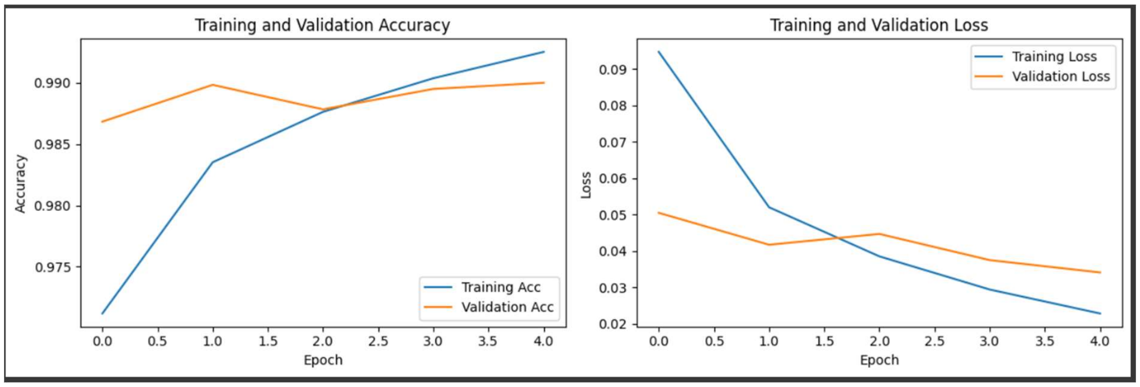
Source Code:

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data
x_train = x_train.reshape((60000, 28, 28, 1)).astype('float32') / 255
x_test = x_test.reshape((10000, 28, 28, 1)).astype('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Build the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes for digits 0-9
])

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

# Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.1)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"\nTest accuracy: {test_acc:.4f}")

# Plot training accuracy and loss
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Acc')
plt.plot(history.history['val_accuracy'], label='Validation Acc')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

Output:



```
[7] # Train the model
    history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.1)

    Epoch 1/5
    844/844 ━━━━━━━━━━━━━━━━━━━━ 43s 51ms/step - accuracy: 0.9642 - loss: 0.1178 - val_accuracy: 0.9868 - val_loss: 0.0505
    Epoch 2/5
    844/844 ━━━━━━━━━━━━━━━━━━━━ 43s 51ms/step - accuracy: 0.9821 - loss: 0.0566 - val_accuracy: 0.9898 - val_loss: 0.0417
    Epoch 3/5
    844/844 ━━━━━━━━━━━━━━━━━━━━ 80s 49ms/step - accuracy: 0.9884 - loss: 0.0366 - val_accuracy: 0.9878 - val_loss: 0.0447
    Epoch 4/5
    844/844 ━━━━━━━━━━━━━━━━━━━━ 82s 49ms/step - accuracy: 0.9915 - loss: 0.0267 - val_accuracy: 0.9895 - val_loss: 0.0375
    Epoch 5/5
    844/844 ━━━━━━━━━━━━━━━━━━━━ 83s 51ms/step - accuracy: 0.9931 - loss: 0.0222 - val_accuracy: 0.9900 - val_loss: 0.0341

    # Evaluate the model
    test_loss, test_acc = model.evaluate(x_test, y_test)
    print(f"\nTest accuracy: {test_acc:.4f}")

    313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9861 - loss: 0.0416

    Test accuracy: 0.9896
```



Result:

Therefore, the CNN model with its individual layers has been successfully created and tested for image recognition. This was achieved by training the model using a dataset and obtaining performance metrics for the model.

Exp 08: Stock Price Prediction Using LSTM

Aim:
To Construct the LSTM model using layers to predict stock prices.

Software Requirement:

- Google Colaboratory

Description:
LSTM Model
Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture designed to capture long-term dependencies in sequential data. It is particularly effective for tasks such as natural language processing (NLP), speech recognition, and time series forecasting. Here's how LSTM works and how it can be implemented in TensorFlow 2:
Cell State and Gates:

- The key to LSTM's ability to capture long-term dependencies is its cell state, which serves as a conveyor belt that can run along the entire sequence, with only minor linear interactions.
- LSTM has three gates that control the flow of information into and out of the cell state:
    - **Forget Gate:** Determines which information to discard from the cell state.
    - **Input Gate:** Determines which new information to store in the cell state.
    - **Output Gate:** Determines which information to output based on the cell state.

LSTM Operations:

- At each time step, the LSTM takes as input the current input xt, the previous hidden state h(t-1), and the previous cell state c(t-1).
- It then computes the new cell state ct and the new hidden state ht using the following equations:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{c}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \cdot \tanh(c_t)
\end{aligned}
$$

Here, ft, it, and ot are the forget, input, and output gate vectors, respectively. tilde{c}t is the candidate cell state vector, and ct is the new cell state. sigma is the sigmoid activation function, and tanh is the hyperbolic tangent activation function.

Algorithm

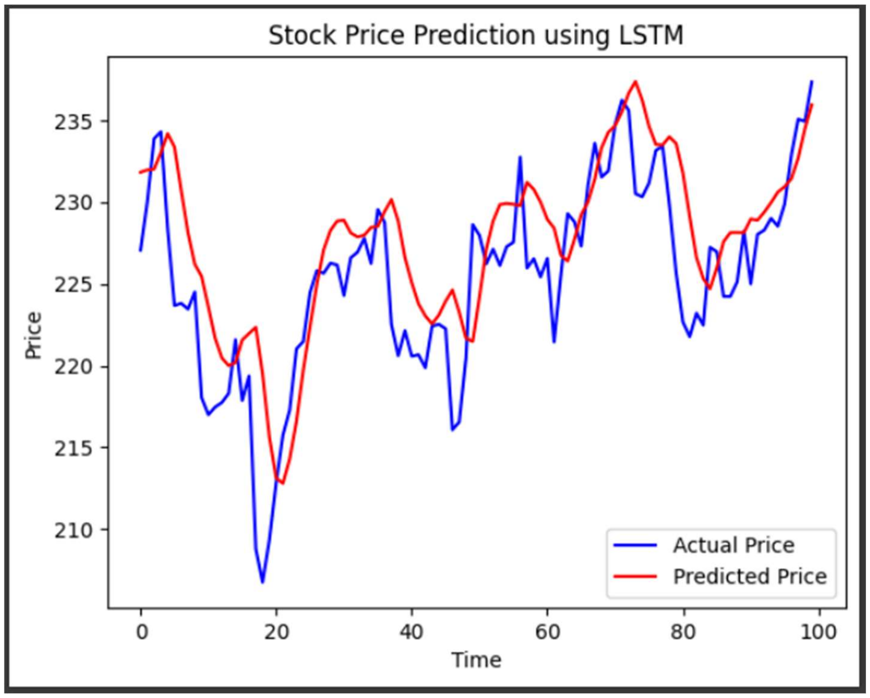
- Import dataset
- Feature scale the data if needed
- Create the network model architecture
- Build the layers and compile the model
- Fit the values and train with set number of epochs and in batch.
- Predict the outcome using test data

Code Implementation:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense


# Load your local CSV file (replace 'AAPL.csv' with your actual path)
data = pd.read_csv('AAPL.csv')
data = data[['Date', 'Close']]  # Ensure 'Close' column exists
data = data.dropna()


# Convert to datetime and sort
data['Date'] = pd.to_datetime(data['Date'])
data.sort_values('Date', inplace=True)


# Prepare the closing price
close_prices = data['Close'].values.reshape(-1, 1)


# Scaling
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(close_prices)


# Creating training data
X_train, y_train = [], []
window_size = 60
for i in range(window_size, len(scaled_data)):
    X_train.append(scaled_data[i-window_size:i, 0])
    y_train.append(scaled_data[i, 0])


X_train = np.array(X_train)
y_train = np.array(y_train)
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
```

```
# Building the model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(X_train.shape[1], 1)),
    LSTM(50),
    Dense(1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=10, batch_size=32)


# Prediction
test_input = scaled_data[-(window_size + 100):]
X_test = []
for i in range(window_size, len(test_input)):
    X_test.append(test_input[i-window_size:i, 0])
X_test = np.array(X_test)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))
predicted = model.predict(X_test)
predicted = scaler.inverse_transform(predicted)


# Plotting
plt.plot(close_prices[-len(predicted):], color='blue', label='Actual Price')
plt.plot(predicted, color='red', label='Predicted Price')
plt.title('Stock Price Prediction using LSTM')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.show()
```

Output:



Result:
Thus, the LSTM model generated predicts the stock price based on the provided data.

Exp 09: Neural Style Transfer Using Gan

Aim:
To Applying a style from one image to another using neural style transfer with Generative Adversarial Networks (GANs).

Software Requirement:

- Google Colaboratory

Description:

Neural Style Transfer:

Neural style transfer is a technique in deep learning that combines the content of one image with the style of another image to create a new image. This technique is based on the idea that the content of an image can be separated from its style, and these two aspects can be independently manipulated.

Content Image and Style Image:

- Neural style transfer requires two input images: a content image and a style image.
- The content image is the image whose content (objects and their arrangement) we want to preserve in the final image.
- The style image is the image whose style (textures, colors, and brush strokes) we want to apply to the final image.

Feature Extraction:

- Neural style transfer typically uses a pre-trained convolutional neural network (CNN) to extract features from the content and style images.
- The CNN is used to extract features at different layers of the network, capturing both low-level features (e.g., edges and textures) and high-level features (e.g., objects and their arrangements).

Loss Functions:

- Neural style transfer uses three loss functions to define the optimization problem:
    - **Content Loss:** Measures the difference between the content features of the generated image and the content image. It ensures that the generated image preserves the content of the content image. -
    - **Style Loss:** Measures the difference between the style features of the generated image and the style image. It ensures that the generated image has a similar style to the style image. -
    - **Total Variation Loss:** Encourages smoothness in the generated image by penalizing rapid changes in pixel values.

Optimization:

- Neural style transfer aims to minimize the total loss, which is a weighted sum of the content loss, style loss, and total variation loss.

- The optimization process involves updating the pixel values of the generated image to minimize the total loss using gradient descent or a similar optimization algorithm.

Output Image:

- The output of the neural style transfer algorithm is a new image that combines the content of the content image with the style of the style image.
- The generated image preserves the content and structure of the content image while incorporating the colors, textures, and brush strokes of the style image.

Neural style transfer is a powerful technique that can be used for artistic purposes, such as creating stylized images or videos, as well as for practical applications, such as image editing and content generation.

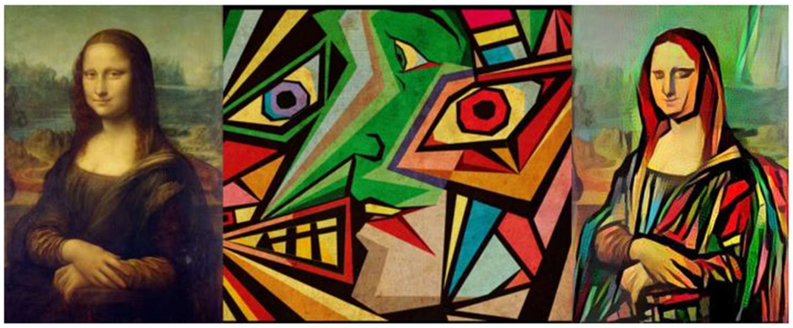


Image Source - https://chelseatroy.com/2019/03/01/neural-style-transfer-with-latte-art-part-1-layer-depth/

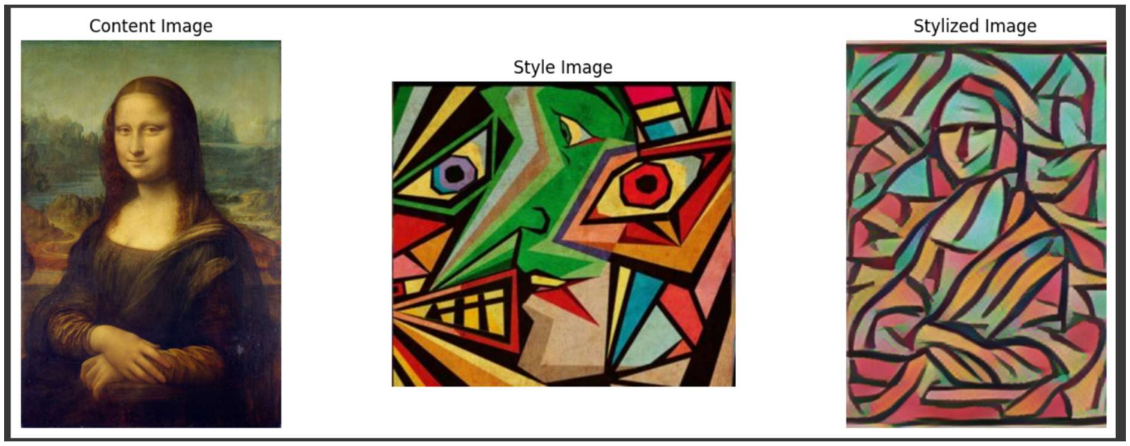
Algorithm:

- Import original image
- Check the size, shape, color channel and tensor shape
- Import Style image and perform the same set of checks
- Display and check the actual images
- For the model connect to the TF-Hub handle
- To the model now provide the original and style image
- Finally display the styled image

Code Implementation:

```
import tensorflow as tf
import tensorflow_hub as hub
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

# Function to load and preprocess image
def load_image(path_to_img, max_dim=512):
    img = Image.open(path_to_img)
    img = img.convert('RGB')
    long = max(img.size)
    scale = max_dim / long
    img = img.resize((round(img.size[0] * scale), round(img.size[1] * scale)))
    img = np.array(img)
    img = img.astype(np.float32)[np.newaxis, ...] / 255.
    return img

# Load content and style images
content_image = load_image("path_to_content_image.jpg")
style_image = load_image("path_to_style_image.jpg")

# Load TensorFlow Hub model
hub_model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')

# Stylize image
stylized_image = hub_model(tf.constant(content_image), tf.constant(style_image))[0]

# Display images
def show_image(img, title=''):
    plt.imshow(np.squeeze(img))
    plt.title(title)
    plt.axis('off')

plt.figure(figsize=(15,5))
plt.subplot(1, 3, 1)
show_image(content_image, 'Content Image')

plt.subplot(1, 3, 2)
show_image(style_image, 'Style Image')

plt.subplot(1, 3, 3)
show_image(stylized_image, 'Stylized Image')
plt.show()
```

Output:



Result:

Therefore, the styled image result is achieved through the utilization of neural style transfer with GANs.