INDEX

Practical: 1

Aim:- Implement and visualize different activation functions by plotting their graphs to understand their behavior and impact on neural networks.

Theory:-

1. **Sigmoid function:**

The Sigmoid function is a type of mathematical function that maps any real-valued number to a value between 0 and 1. It is commonly used in machine learning, especially in logistic regression and neural networks, to introduce non-linearity into models. The function is defined as:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Where:

- S(x) is the output of the sigmoid function,
- e is the base of the natural logarithm, and
- x is the input.

This function has the property of being differentiable, and its output is always between 0 and 1, which makes it useful for probabilities and binary classification tasks.
As $x \to \infty$, $S(x) \to 1$, and as $x \to -\infty$, $S(x) \to 0$. The function is symmetric around x=0, where S (0) = 0.5.
In the code provided, we visualize both the sigmoid function and its derivative using Matplotlib**.** The sigmoid function and its derivative are computed for a range of x-values from -6 to 6.

Here's the step-by-step explanation:

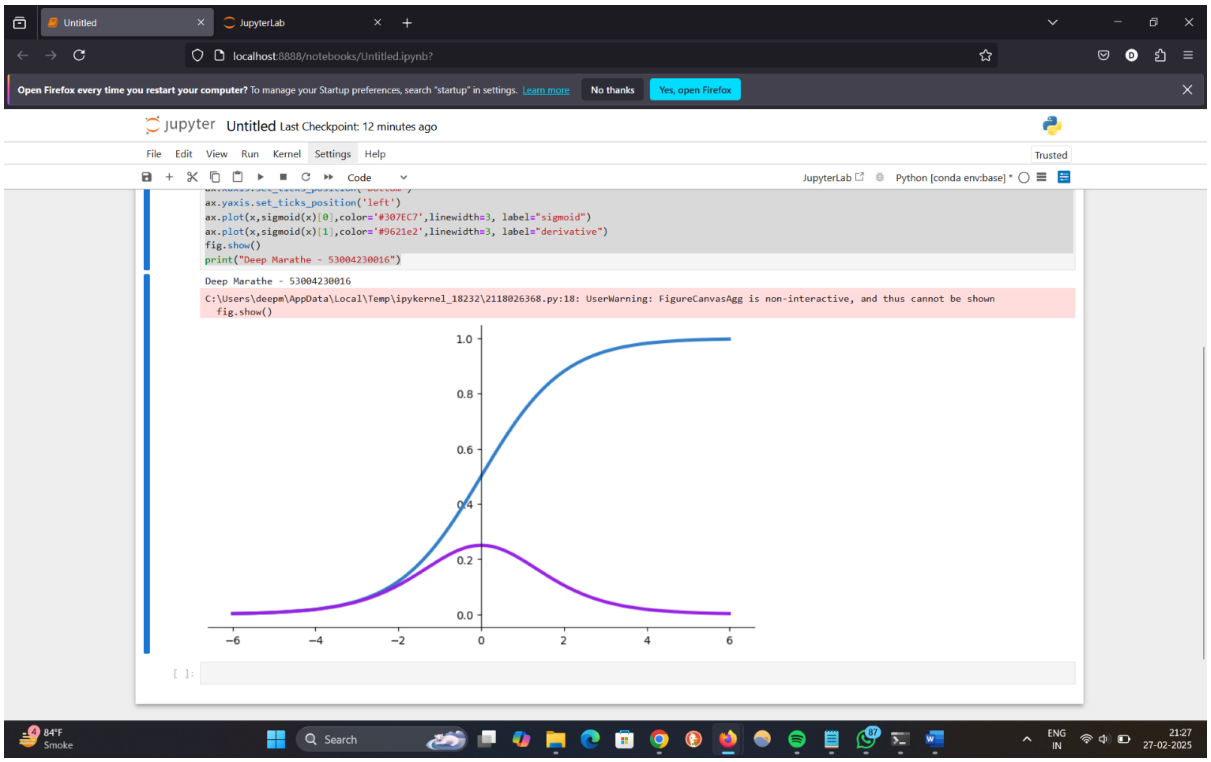
1. **Sigmoid Function Plot**: The blue curve represents the sigmoid function itself. It starts near 0 for very negative x, increases smoothly, and approaches 1 as x becomes more positive.
2. **Derivative Plot**: The purple curve represents the derivative of the sigmoid function. It shows how the rate of change of the sigmoid function behaves. As shown, the derivative has its peak near x=0and decreases as x moves away from the origin.

Code:-

```
#Sigmoid Function

import matplotlib.pyplot as plt

import numpy as np

def sigmoid(x):

  s=1/(1+np.exp(-x))

  ds=s*(1-s)
```

```
    return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x,sigmoid(x)[0],color='#307EC7',linewidth=3, label="sigmoid")
ax.plot(x,sigmoid(x)[1],color='#9621e2',linewidth=3, label="derivative")
fig.show()
print("Deep Marathe - 53004230016")
```

Output:-



2. TanH (Hyperbolic Tangent) Function:

The TanH (hyperbolic tangent) function is another activation function commonly used in neural networks and machine learning. It is similar to the sigmoid function, but it outputs values between -1 and 1 instead of 0 and 1. The TanH function is defined as:

$$\text{TanH}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Where:

- e is the base of the natural logarithm,
- x is the input.

The TanH function transforms input values into a range from -1 to 1, which can be useful when we need both negative and positive outputs. It is widely used in neural networks, particularly in hidden layers, because it provides a symmetric range and has a strong derivative for learning.

In the provided code, the TanH function and its derivative are visualized using Matplotlib. Here's a detailed explanation of the steps:

1. **TanH Function Plot**: The blue curve represents the TanH function. It is symmetric around the origin and smoothly maps input values into the range of -1 to 1. As $x \to \infty$, the output approaches 1, and as $x \to -\infty$, the output approaches -1. At x=0, the output is 0.
2. **Derivative Plot:** The purple curve represents the derivative of the TanH function. This curve shows how the rate of change of the TanH function behaves. The derivative is largest at x=0, where the slope of the TanH function is the steepest. As x moves away from 0 in either direction, the derivative becomes smaller, approaching 0 as x goes to infinity or negative infinity.

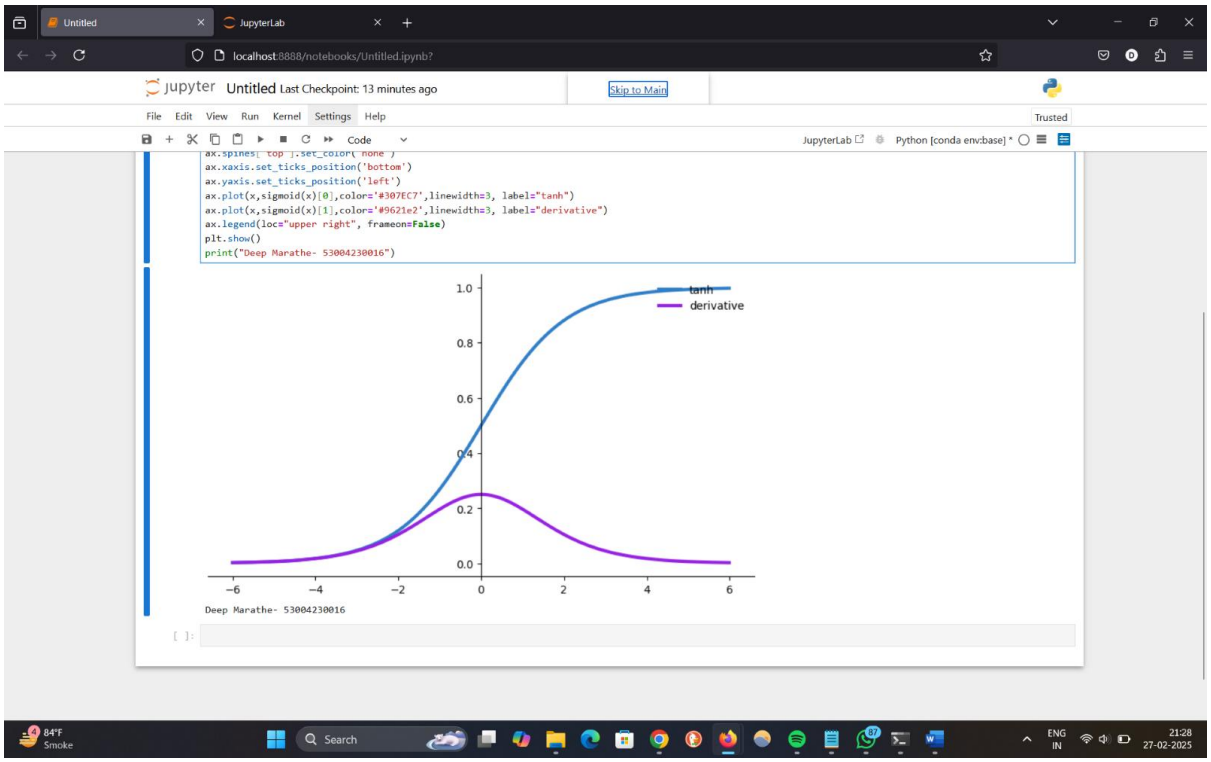
The plot is created using Matplotlib where the TanH function is plotted in blue, and the derivative is plotted in purple. The axes are adjusted to center at (0, 0), making the symmetry of the TanH function more visually apparent.

Code:-

```
#Tanh Function

import matplotlib.pyplot as plt

import numpy as np

def tanh(x):

   t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))

   dt=1-t**2

   return t,dt

z=np.arange(-4,4,0.01)

tanh(z)[0].size,tanh(z)[1].size
```

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x,sigmoid(x)[0],color='#307EC7',linewidth=3, label="tanh")
ax.plot(x,sigmoid(x)[1],color='#9621e2',linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
plt.show()
print("Deep Marathe- 53004230016")
```

Output:-



3. ReLU (Rectified Linear Unit) Function:

The **ReLU** (Rectified Linear Unit) function is one of the most widely used activation functions in deep learning models, especially in convolutional and fully connected layers. It is preferred due to its simplicity and the fact that it helps mitigate some of the issues seen with other activation functions like **Sigmoid** or **TanH** (such as the vanishing gradient problem). The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Where:

- x is the input value to the function, and
- The function outputs x if x is greater than 0, otherwise, it outputs 0.

This means that for any positive input, the output is the same as the input, while for negative inputs, the output is 0. The function is non-linear and piecewise linear at x=0, making it computationally efficient.

1. **ReLU Function**: The blue curve represents the ReLU activation function. It starts at 0 for negative values of x and increases linearly with a slope of 1 for positive values of x. This indicates that all negative inputs are squashed to 0, while positive inputs pass through unchanged.
2. **ReLU Derivative**: The purple curve represents the derivative of the ReLU function. For values of x greater than 0, the derivative is 1, indicating a constant slope of 1 in the positive region of the ReLU curve. For values of x less than or equal to 0, the derivative is 0, meaning that the function is flat, and no change occurs in this region.

Limitations of ReLU:

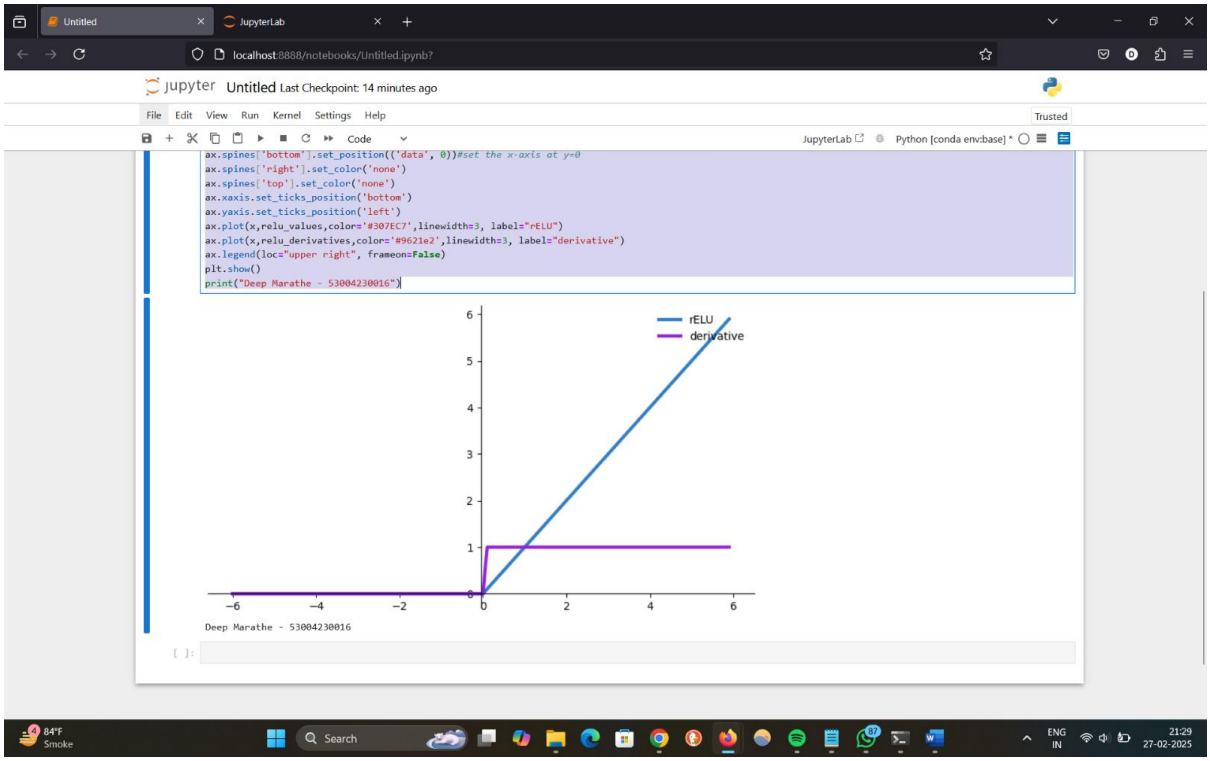
- Dying ReLU Problem**:** ReLU units can sometimes "die" during training, meaning that they stop learning entirely if they get stuck in the inactive region ($x \leq 0$), where the derivative is 0. Variants like Leaky ReLU and Parametric ReLU are introduced to address this issue.

Code:-

```
#Relu Function and its derivative

import matplotlib.pyplot as plt

import numpy as np

def relu(x):

  r = np.maximum(0, x)
```

```
    dr = np.where(x <=0, 0, 1)
    return r,dr
x=np.arange(-6, 6, 0.1)#Range for the x-axis
relu_values, relu_derivatives = relu(x) #Compute ReLU AND its derivative
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position(('data', 0))#set the x-axis at y=0
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x,relu_values,color='#307EC7',linewidth=3, label="rELU")
ax.plot(x,relu_derivatives,color='#9621e2',linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
plt.show()
print("Deep Marathe - 53004230016")
```

Output:-



4. **Leaky ReLU Activation Function:**

The Leaky ReLU (Leaky Rectified Linear Unit) function is a variation of the traditional ReLU function designed to address the problem of "dead neurons" encountered with ReLU. The problem with standard ReLU is that when the input is less than or equal to zero, the derivative is zero, and neurons can "die" and stop learning altogether. Leaky ReLU introduces a small slope for negative input values to prevent neurons from becoming inactive.
The Leaky ReLU function is defined as:

$$\text{Leaky ReLU}(x) = \max(\alpha x, x)$$

Where:

- x is the input value.
- α is a small positive constant (usually set to 0.01), which controls the slope for negative input values.
- For x≥0, the function behaves the same as ReLU, i.e., it outputs x.
- For x<0, the function outputs αx, where α is typically a small positive value.

This modification ensures that the function does not completely "saturate" for negative values, which prevents the "dying neuron" problem in neural networks.

1. **Leaky ReLU Function**: The blue curve represents the Leaky ReLU function. For values of x≥0, the function behaves just like the ReLU function, outputting the input value. However, for x<0, the function has a small slope defined by α (default is 0.01), ensuring that negative values do not cause the function to "die" (i.e., it does not output zero for negative inputs).
2. **Derivative of the Leaky ReLU Function**: The purple curve represents the derivative of the Leaky ReLU function. For x>0, the derivative is 1, indicating a constant slope. For x<0, the derivative is α\alphaα (default is 0.01), which ensures that the gradient is non-zero even for negative values of x, preventing dead neurons.

Advantages of Leaky ReLU:

- **Prevents Dying Neurons**: Unlike the standard ReLU, Leaky ReLU allows for a small gradient when x is negative, which helps avoid the "dying ReLU" problem where neurons stop learning.
- **Simplicity**: Leaky ReLU is still simple to compute and does not introduce significant computational overhead.
- **Improved Learning**: It allows the network to continue learning even when the input is negative, which can improve performance in some cases.

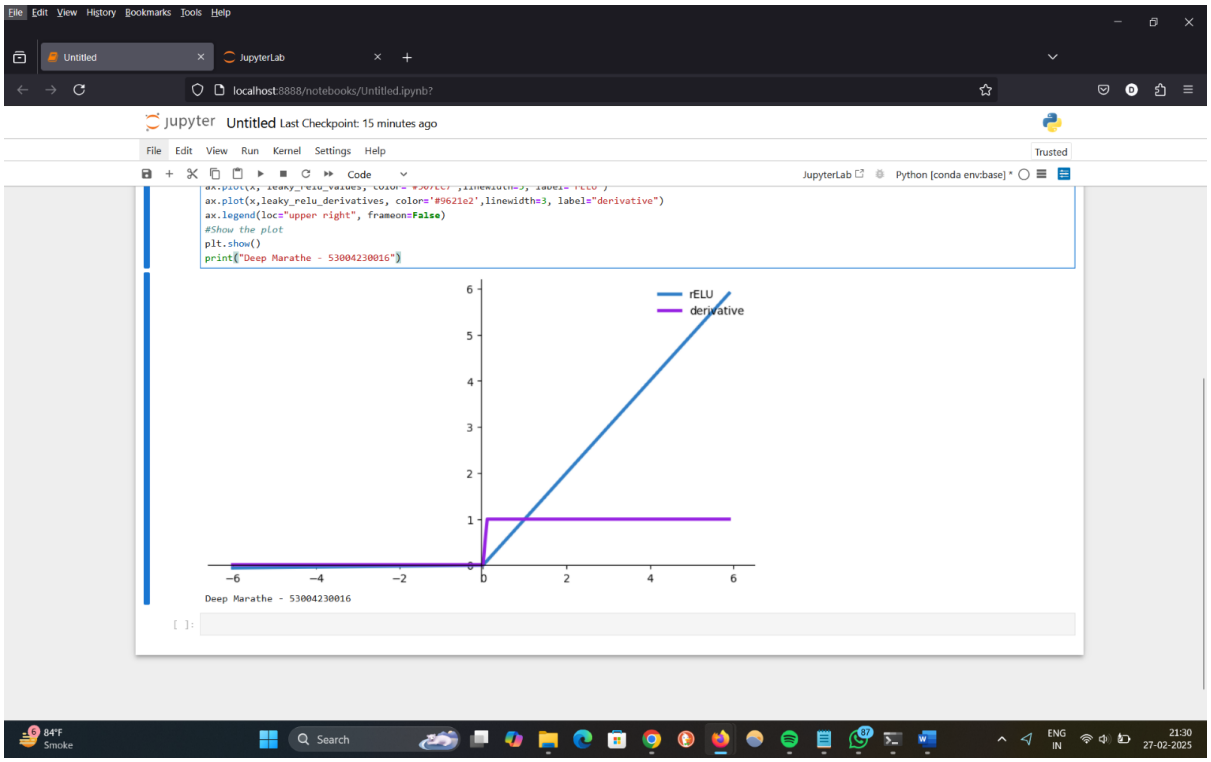
Disadvantages of Leaky ReLU:

- **Choosing the Best α**: The choice of α\alphaα is important. If α is too large, it may lead to a function that behaves too similarly to a linear activation, potentially losing the benefits of non-linearity.
- **Still prone to some issues**: While Leaky ReLU mitigates the "dying neuron" issue, it can still suffer from other problems, such as being overly sensitive to negative inputs.

Code:-

```
# lEAKY Rectified Linear Unit(Leaky Relu)


import matplotlib.pyplot as plt
import numpy as np
def leaky_relu(x, alpha=0.01):
   r = np.maximum(alpha * x, x)
   dr = np.where(x < 0, alpha, 1)
   return r,dr
#Generate x values
x=np.arange(-6, 6, 0.1)#Range for the x-axis
leaky_relu_values, leaky_relu_derivatives = leaky_relu(x)
# Create the plot
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position(('data', 0))#set the x-axis at y=0
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x, leaky_relu_values, color='#307EC7',linewidth=3, label="rELU")
ax.plot(x,leaky_relu_derivatives, color='#9621e2',linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
#Show the plot
plt.show()
print("Deep Marathe - 53004230016")
```

Output:-



5. <u>Parametric Rectified Linear Unit (PReLU) Activation Function:</u>

The Parametric Rectified Linear Unit (PReLU) is a variation of the Leaky ReLU function that introduces an additional learnable parameter, α, which controls the slope for negative inputs. Unlike Leaky ReLU, where α is a fixed constant, PReLU allows the model to learn the best value for α during training. This makes it more flexible and potentially more effective for different types of data.

The PReLU function is defined as:

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$$

Where:

- x is the input to the function.
- α is a learnable parameter that controls the slope of the function for negative values of x.
- For x≥0, the function behaves like the standard ReLU, i.e., it outputs x.
- For x<0, the function outputs αx, with α being a trainable parameter.

The advantage of this flexibility is that it allows the network to learn the best value of α\alphaα, improving the model's performance on tasks where different features may require different levels of activation for negative inputs.

1. **PReLU Function**: The blue curve represents the PReLU activation function. For $x \geq 0$, it behaves like a standard ReLU function, where the output is the same as the input. For $x<0$, the output is scaled by a factor of α, meaning that the function still provides a non-zero output even for negative values.
2. **PReLU Derivative**: The purple curve represents the derivative of the PReLU function. For positive values of x, the derivative is 1, just like ReLU. For negative values of x, the derivative is α, ensuring that the network can continue learning even for negative input values.

Advantages of PReLU

- **Learnable Slope**: Unlike Leaky ReLU, where α is fixed, PReLU allows the model to learn the optimal value of α during training, improving flexibility and adaptability.
- **Prevents Dead Neurons**: Just like Leaky ReLU, PReLU avoids the "dying neuron" problem by allowing negative inputs to contribute to the learning process.
- **Increased Flexibility**: The learnable parameter α allows the model to adjust how much negative information is allowed to propagate through the network, which can lead to better performance.

Disadvantages of PReLU

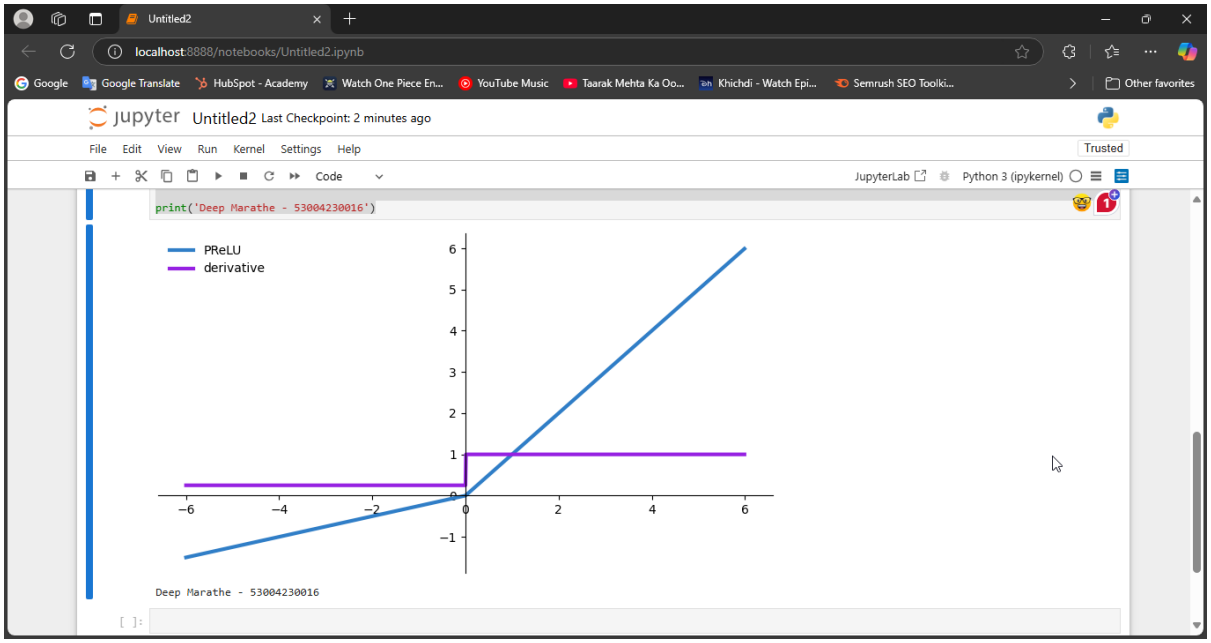
- **Risk of Overfitting**: Since α is a learnable parameter, it introduces more parameters into the model, which increases the risk of overfitting, especially in smaller datasets.
- **More Computationally Expensive**: Learning the value of α introduces additional computational complexity during training.

Code:-

```
Parametric Rectified Linear Unit (PReLU)

import matplotlib.pyplot as plt

import numpy as np

def prelu(x, alpha=0.25):

r = np.maximum(alpha*x, x)

dr = np.where(x < 0, alpha, 1)

return r, dr


x = np.arange(-6, 6, 0.01)

prelu_values, prelu_derivatives = prelu(x)

fig, ax = plt.subplots(figsize=(9,5))

ax.spines['left'].set_position('center')

ax.spines['bottom'].set_position(('data', 0))
```

```
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x, prelu_values, color="#307EC7", linewidth=3, label="PReLU")
ax.plot(x, prelu_derivatives, color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper left", frameon=False)
plt.show()
print('Deep Marathe - 53004230016')
```

Output:-



6. ELU Activation Function:

The **Exponential Linear Unit (ELU)** is a type of activation function that is used in neural networks to introduce non-linearity. Unlike traditional activation functions like ReLU, which outputs zero for negative inputs, ELU has a smoother transition for negative values, which helps avoid issues such as "dying neurons" and allows for better optimization. The mathematical form of the ELU function is given as:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

Where α is a constant that controls the smoothness of the function for negative values. By default, α=1.0.

The code defines a function elu(x, alpha=1.0) that computes both the ELU function and its derivative for a given input x.

- elu(x, alpha=1.0): This function accepts an array x and a parameter alpha to calculate the ELU output and its derivative.
- The ELU function (r) is computed using np.where(x >= 0, x, alpha*(np.exp(x) - 1)), which returns x for values of x >= 0 and alpha * (e^x - 1) for values of x < 0.
- The derivative (dr) is computed using np.where(x >= 0, 1, alpha * np.exp(x)), returning 1 for x >= 0 and alpha * e^x for x < 0.
- x = np.arange(-6, 6, 0.01): This line generates an array of x values from -6 to 6 with a step size of 0.01.

- **ELU Function**: The ELU curve (in blue) has a linear relationship for positive values of x, and for negative values of x, it follows an exponential curve, which smoothly saturates and avoids the flat zero region seen in ReLU.
- **Derivative**: The derivative curve (in purple) is 1 for positive values of x (as the ELU function is linear there), and for negative values, it follows α * exp(x) (exponential behavior). The derivative curve's behavior is significant for optimization as it provides the gradient for backpropagation.

The ELU activation function is considered beneficial because:

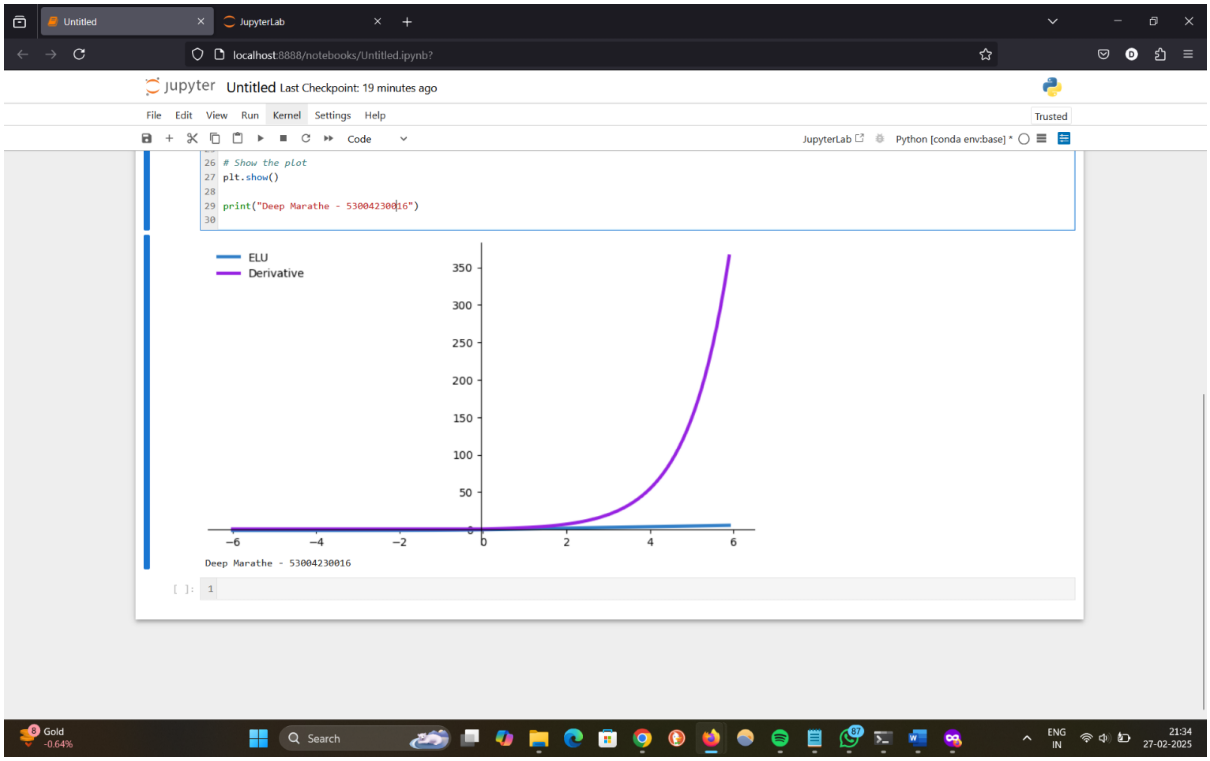
- It reduces the likelihood of "dead neurons" (common in ReLU, where neurons output zero and stop learning).
- It allows the network to have a smoother gradient flow, leading to potentially faster convergence during training.

Code:-

```
# Exponential Linear Unit (ELU)

import matplotlib.pyplot as plt
import numpy as np

def elu(x, alpha=1.0):
    r = np.where(x >= 0, x, alpha * (np.exp(x) - 1))
    dr = np.where(x <= 0, 1, alpha * np.exp(x))
    return r, dr

x = np.arange(-6, 6, 0.1)  # Range for the x-axis
elu_values, elu_derivatives = elu(x)

# Create the plot
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position(('data', 0))  # Set the x-axis at y=0
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

# Plotting the ELU and its derivative
ax.plot(x, elu_values, color='#307EC7', linewidth=3, label="ELU")
ax.plot(x, elu_derivatives, color='#9621e2', linewidth=3, label="Derivative")
ax.legend(loc="upper left", frameon=False)

# Show the plot
plt.show()
```

```
print("Deep Marathe - 53004230016")
```

Output:-



7. SoftPlus Activation Function:

The **SoftPlus** activation function is a smooth approximation of the ReLU (Rectified Linear Unit) function. It can be described by the equation:

$$f(x) = \log(1 + e^x)$$

The function is differentiable and smooth, unlike ReLU, which has a sharp transition at x=0. This smooth nature makes it less likely to cause issues like "dead neurons" that can occur with ReLU.

- For large positive values of x, SoftPlus behaves similarly to ReLU, since $\log(1+e^x) \approx x$ for large x.
- For large negative values of x, the SoftPlus function approaches zero, but smoothly, as opposed to the hard zero cut-off of ReLU.

➢ SoftPlus(x): This function takes an array x and computes both the SoftPlus function (r) and its derivative (dr).
- The SoftPlus function is computed as np.log(1 + np.exp(x)), which applies the logarithmic function to the exponential of x and adds 1.
- The derivative of SoftPlus is computed as 1 / (1 + np.exp(-x)), which is the sigmoid function.

➢ x = np.arange(-6, 6, 0.01): This creates an array of x values ranging from -6 to 6 with a step size of 0.01, providing the input for the SoftPlus function and its derivative.

- **SoftPlus Function:** The SoftPlus curve (in blue) starts from approximately zero for negative values of x and smoothly increases for positive x. As x grows larger, the curve approaches a straight line with a slope of 1 (similar to the ReLU function).
- **Derivative (Sigmoid):** The derivative (in purple) follows the sigmoid curve, with values between 0 and 1. It approaches 0 for large negative x and approaches 1 for large positive x. This derivative describes the rate of change of the SoftPlus function at each point.

The SoftPlus function is commonly used as an activation function in neural networks because:

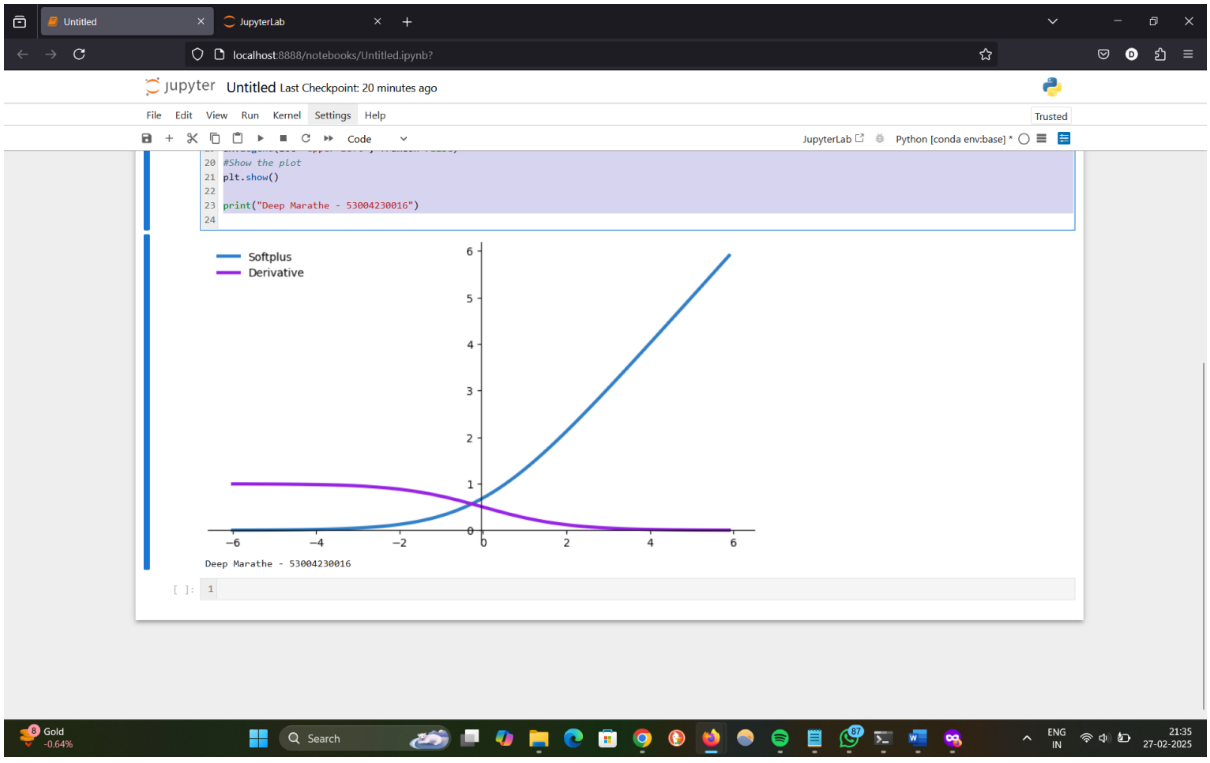
- It is a smooth, continuous function, which makes it differentiable everywhere, preventing problems with optimization, such as issues that arise with non-differentiable functions like ReLU at x=0.
- The derivative of SoftPlus is the sigmoid function, which can help stabilize the gradients during training by avoiding extremely small or large gradient values.
- For large positive values of x, it behaves similarly to ReLU, providing a non-zero output, while for negative values of x, the output is closer to zero, maintaining a smooth gradient.

Code:-

```
import matplotlib.pyplot as plt

import numpy as np

def softplus(x):

   r = np.log(1 + np.exp(x))

   dr = 1/(1 + np.exp(x))

   return r,dr

x=np.arange(-6, 6, 0.1)#Range for the x-axis

softplus_values, softplus_derivatives = softplus(x)

fig, ax = plt.subplots(figsize=(9, 5))

ax.spines['left'].set_position('center')

ax.spines['bottom'].set_position(('data', 0))#set the x-axis at y=0

ax.spines['right'].set_color('none')

ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')

ax.yaxis.set_ticks_position('left')

#Plotting the PRelu and its derivative

ax.plot(x, softplus_values, color='#307EC7',linewidth=3, label="Softplus")
```

```
ax.plot(x,softplus_derivatives, color='#9621e2',linewidth=3, label="Derivative")
ax.legend(loc="upper left", frameon=False)
#Show the plot
plt.show()

print("Deep Marathe - 53004230016")
```

Output:-



8. ArcTan (Arctangent) Activation Function:

The **ArcTan (arctangent)** function is the inverse of the tangent function. It maps real values to an output in the range of $-\pi/2 \leq \text{ArcTan}(x) \leq \pi/2$. The function is defined mathematically as:

$$f(x) = \arctan(x)$$

It is used as an activation function in some neural networks due to its smooth and continuous nature. The ArcTan function:

- Has an output that is bounded, with asymptotes at $\pm\pi/2$ as x approaches infinity.
- The output smoothly transitions from negative to positive values, which is desirable for many optimization tasks in neural networks.

The code defines a function arctan(x) that computes both the ArcTan function and its derivative.

- **arctan(x)**: This function takes an array x and computes both the ArcTan function (r) and its derivative (dr):
- The ArcTan function is computed using np.arctan(x), which calculates the arctangent of each element in the input array x.
- The derivative of ArcTan is computed using the formula d/dx arctan(x)=1/1+x^2 which is implemented as 1 / (1 + x**2).
- **x = np.arange(-6, 6, 0.01)**: This creates an array of x values ranging from -6 to 6, with a step size of 0.01. This array serves as the input to the ArcTan function and its derivative.

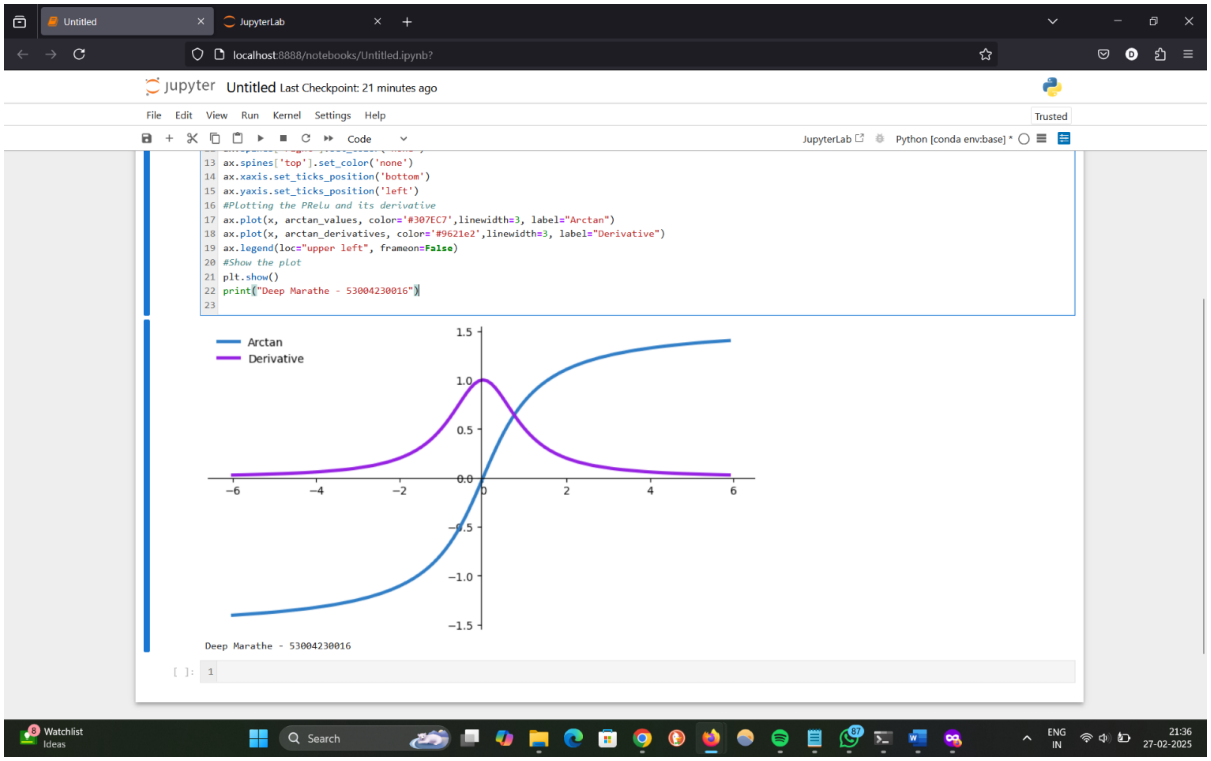
- **ArcTan Function**: The ArcTan curve (in blue) smoothly transitions from negative to positive values. As x approaches large positive or negative values, the function asymptotically approaches $\pm\pi/2$. For x=0, the ArcTan function crosses through 0, with a smooth curve across the entire range.
- **Derivative**: The derivative (in purple) is given by 1/ {1 + x^2} . It is a smooth curve that starts at 1 when x=0, and as x becomes large (positive or negative), the derivative approaches 0. The slope of the ArcTan function is steepest around x=0 and becomes flatter as |x| increases.

Code:-

```
import matplotlib.pyplot as plt

import numpy as np
```

```
def arctan(x):
    r = np.arctan(x)
    dr = 1/(1 + x**2)
    return r,dr
x=np.arange(-6, 6, 0.1)#Range for the x-axis
arctan_values, arctan_derivatives = arctan(x)
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position(('data', 0))#set the x-axis at y=0
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
#Plotting the PRelu and its derivative
ax.plot(x, arctan_values, color='#307EC7',linewidth=3, label="Arctan")
ax.plot(x, arctan_derivatives, color='#9621e2',linewidth=3, label="Derivative")
ax.legend(loc="upper left", frameon=False)
#Show the plot
plt.show()
print("Deep Marathe - 53004230016")
```

Output:



Activation functions and derivatives

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

from sklearn.datasets import make_moons

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

import numpy as np

import matplotlib.pyplot as plt

Data preparation

X, y = make_moons(n_samples=1000, noise=0.2, random_state=42) # Fixed parameters

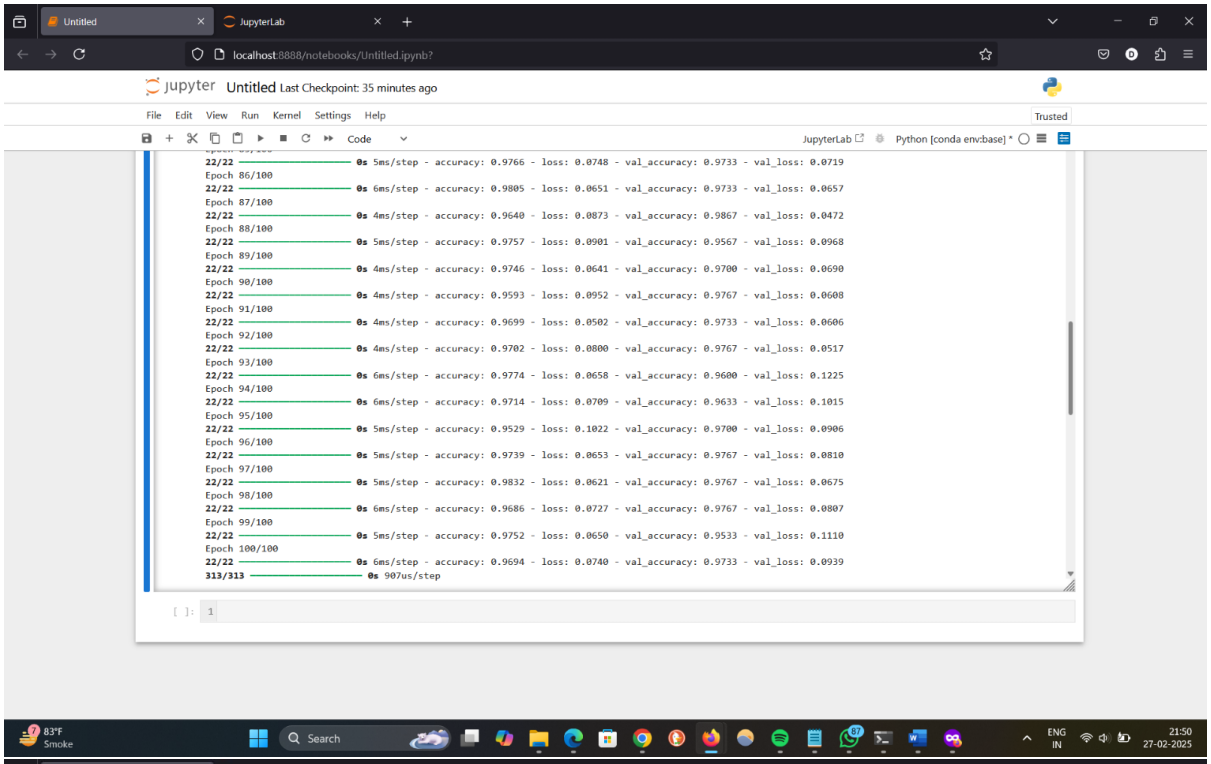
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42) # Corrected assignment

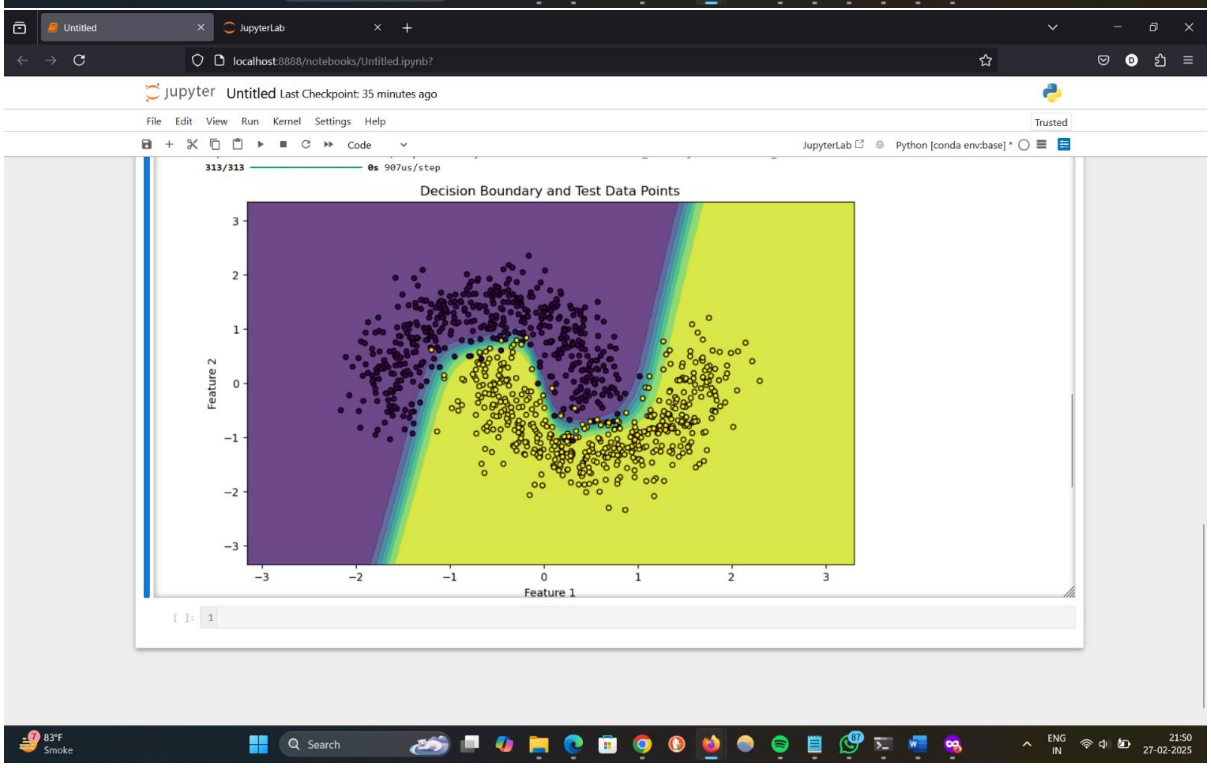
scaler = StandardScaler() # Fixed typo in scaler name

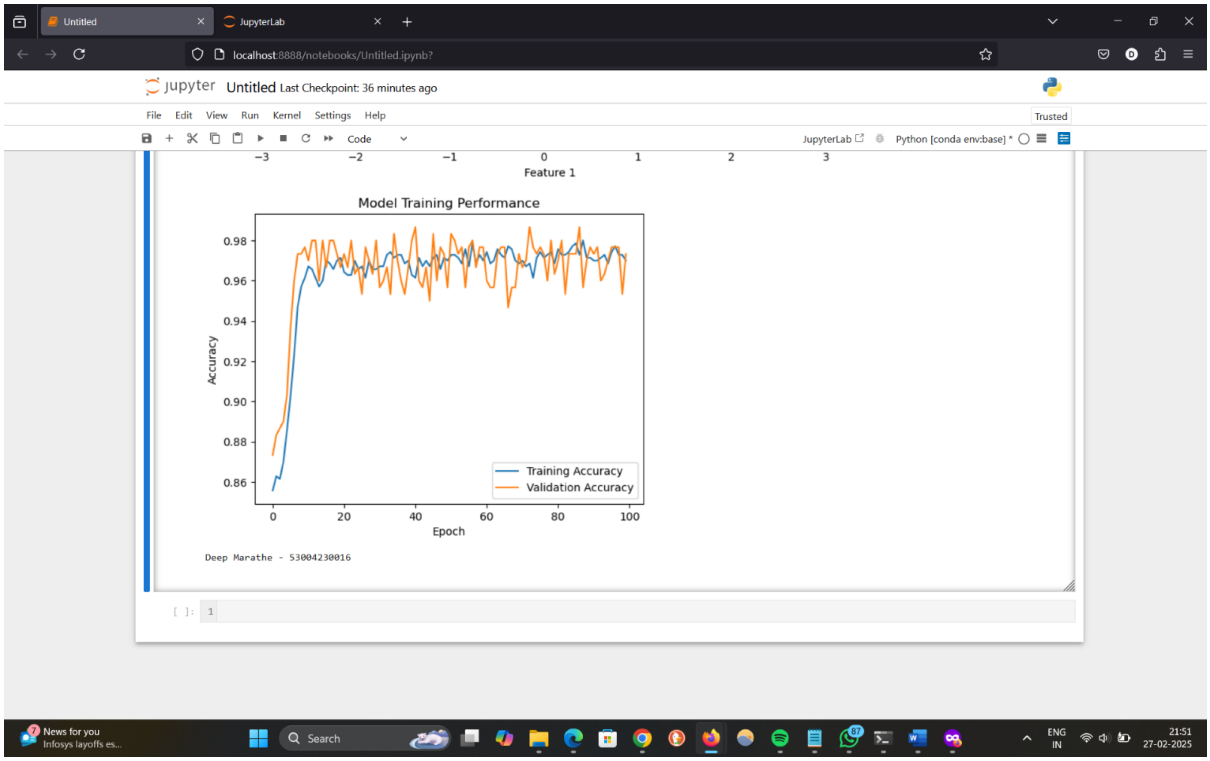
```
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)  # Fixed typo in transform method

# Model definition
model = Sequential([
    Dense(18, input_shape=(2,), activation='tanh'),
    Dense(10, activation='elu'),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(learning_rate=0.01), loss="binary_crossentropy",
metrics=['accuracy'])

history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, verbose=1)  #
Fixed typo in 'verbose'

def plot_decision_boundary(model, X, y):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=20, edgecolor='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Decision Boundary and Test Data Points')

plt.figure(figsize=(10, 6))
plot_decision_boundary(model, np.vstack((X_train, X_test)), np.hstack((y_train, y_test)))
plt.show()
```

```
# Plot the training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Training Performance')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()

print("Deep Marathe - 53004230016")
```

Output:-


Untitled
JupyterLab
localhost:8888/notebooks/Untitled.ipynb?
jupyter Untitled Last Checkpoint: 35 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python [conda env:base] *
22/22 0s 5ms/step - accuracy: 0.9766 - loss: 0.0748 - val_accuracy: 0.9733 - val_loss: 0.0719
Epoch 86/100
22/22 0s 6ms/step - accuracy: 0.9805 - loss: 0.0651 - val_accuracy: 0.9733 - val_loss: 0.0657
Epoch 87/100
22/22 0s 4ms/step - accuracy: 0.9640 - loss: 0.0873 - val_accuracy: 0.9867 - val_loss: 0.0472
Epoch 88/100
22/22 0s 5ms/step - accuracy: 0.9757 - loss: 0.0901 - val_accuracy: 0.9567 - val_loss: 0.0968
Epoch 89/100
22/22 0s 4ms/step - accuracy: 0.9746 - loss: 0.0641 - val_accuracy: 0.9700 - val_loss: 0.0690
Epoch 90/100
22/22 0s 4ms/step - accuracy: 0.9593 - loss: 0.0952 - val_accuracy: 0.9767 - val_loss: 0.0608
Epoch 91/100
22/22 0s 4ms/step - accuracy: 0.9699 - loss: 0.0502 - val_accuracy: 0.9733 - val_loss: 0.0606
Epoch 92/100
22/22 0s 4ms/step - accuracy: 0.9702 - loss: 0.0800 - val_accuracy: 0.9767 - val_loss: 0.0517
Epoch 93/100
22/22 0s 6ms/step - accuracy: 0.9774 - loss: 0.0658 - val_accuracy: 0.9600 - val_loss: 0.1225
Epoch 94/100
22/22 0s 6ms/step - accuracy: 0.9714 - loss: 0.0709 - val_accuracy: 0.9633 - val_loss: 0.1015
Epoch 95/100
22/22 0s 5ms/step - accuracy: 0.9529 - loss: 0.1022 - val_accuracy: 0.9700 - val_loss: 0.0906
Epoch 96/100
22/22 0s 5ms/step - accuracy: 0.9739 - loss: 0.0653 - val_accuracy: 0.9767 - val_loss: 0.0810
Epoch 97/100
22/22 0s 5ms/step - accuracy: 0.9832 - loss: 0.0621 - val_accuracy: 0.9767 - val_loss: 0.0675
Epoch 98/100
22/22 0s 6ms/step - accuracy: 0.9686 - loss: 0.0727 - val_accuracy: 0.9767 - val_loss: 0.0807
Epoch 99/100
22/22 0s 5ms/step - accuracy: 0.9752 - loss: 0.0650 - val_accuracy: 0.9533 - val_loss: 0.1110
Epoch 100/100
22/22 0s 6ms/step - accuracy: 0.9694 - loss: 0.0740 - val_accuracy: 0.9733 - val_loss: 0.0939
313/313 0s 907us/step
[ ]: 1
83°F Smoke
Search
ENG IN
21:50 27-02-2025


Untitled
JupyterLab
localhost:8888/notebooks/Untitled.ipynb?
jupyter Untitled Last Checkpoint: 35 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python [conda env:base] *
313/313 0s 907us/step
Decision Boundary and Test Data Points
Feature 2
Feature 1
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
[ ]: 1
83°F Smoke
Search
ENG IN
21:50 27-02-2025



Practical: 2

Aim:- Implement gradient-based optimization to minimize the loss function and visualize loss reduction over iterations.

Theory:-

Gradient descent optimization process to minimize a simple quadratic loss function, (x−3) ^ 2, using TensorFlow.

Gradient descent works by iteratively adjusting the value of x to minimize the given loss function. The loss function here is a simple quadratic function, (x - 3) ^ 2, which has a global minimum at x=3.

- **TensorFlow** is used for automatic differentiation and optimization.
- **Matplotlib** is used for plotting the loss reduction curve over time.
- The **loss function** is a simple quadratic function: f(x)=(x−3) ^ 2.
- The function calculates the square of the difference between x and 3. The minimum value of this function occurs when x=3, where the loss is 0.
- x is initialized as a TensorFlow **trainable variable** with an initial value of 5.0.
- **learning_rate** is set to 0.1, which determines the size of the steps we take in each iteration towards the minimum.
- steps and **loss_values** are lists to store the values of the step index and corresponding loss values for plotting.
- Gradient **Tape**: TensorFlow's **GradientTape** records the operations on the TensorFlow variables to compute the gradients (derivatives) during backpropagation.
- **tape.gradient(loss, [x])** computes the derivative of the loss function with respect to x.
- Since the loss function is f(x) = (x - 3) ^ 2, its derivative is 2(x−3). This gradient is used to update x towards the minimum.
- Gradient **Update**: The **assign_sub** method is used to update x by subtracting the gradient scaled by the **learning rate** (0.1). This is a typical gradient descent update rule:

$$x = x - \eta \cdot \frac{d}{dx}\text{Loss}(x)$$

where η is the learning rate, and d/dx Loss(x) is the gradient.

The updated value of x is printed at each step along with the corresponding loss.

- After the optimization loop, a plot is generated that shows how the loss decreases over the 20 steps. The x-axis represents the **steps** (iterations), and the y-axis shows the **loss values** at each step.
- The plot demonstrates how the optimization process gradually reduces the loss value over time, converging towards the minimum at x=3.

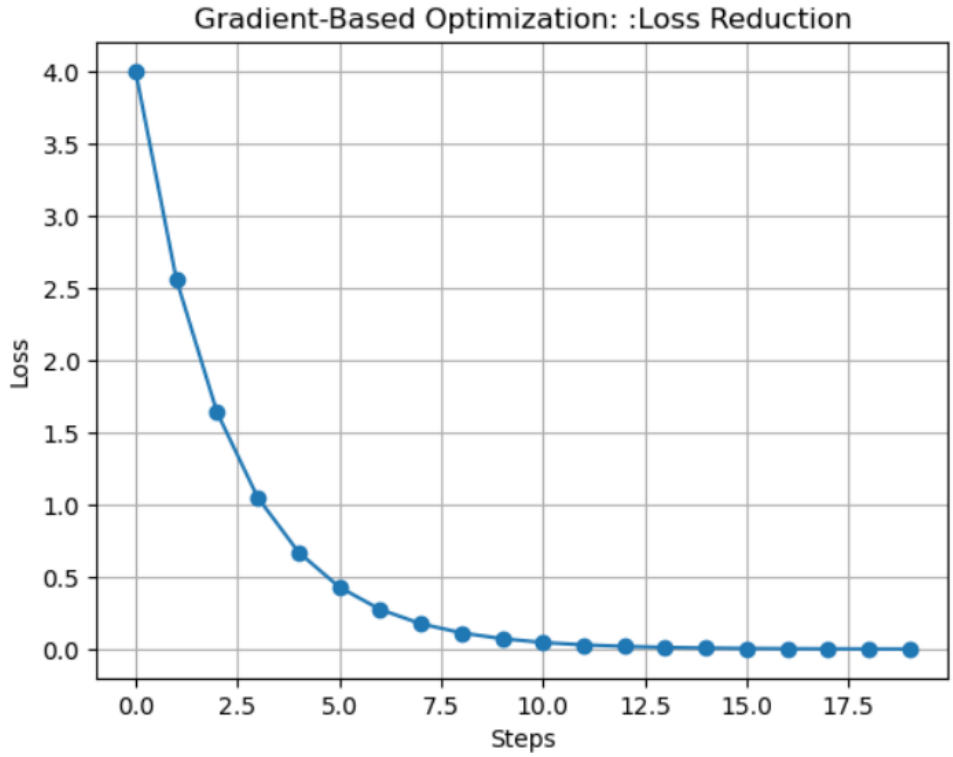
Code:-

```
# Gradient Based Optimization
import tensorflow as tf
import matplotlib.pyplot as plt
def loss_function(x):
    return (x-3)**2
x = tf.Variable(initial_value = 5.0, trainable = True, dtype = tf.float32)
learning_rate = 0.1
steps = []
loss_values = []
for step in range(20):
    with tf.GradientTape() as tape:
        loss = loss_function(x)
    gradients = tape.gradient(loss, [x])
    x.assign_sub(learning_rate*gradients[0])
    steps.append(step)
    loss_values.append(loss.numpy())
    print(f"Step {step+1}: x = {x.numpy():.4f}, Loss = {loss.numpy():.4f}")
plt.plot(steps, loss_values, marker='o')
plt.title('Gradient-Based Optimization: :Loss Reduction')
plt.xlabel('Steps')
```

```
plt.ylabel('Loss')

plt.grid(True)

plt.show()
```

Output:-

```
Step 1: x = 4.6000, Loss = 4.0000
Step 2: x = 4.2800, Loss = 2.5600
Step 3: x = 4.0240, Loss = 1.6384
Step 4: x = 3.8192, Loss = 1.0486
Step 5: x = 3.6554, Loss = 0.6711
Step 6: x = 3.5243, Loss = 0.4295
Step 7: x = 3.4194, Loss = 0.2749
Step 8: x = 3.3355, Loss = 0.1759
Step 9: x = 3.2684, Loss = 0.1126
Step 10: x = 3.2147, Loss = 0.0721
Step 11: x = 3.1718, Loss = 0.0461
Step 12: x = 3.1374, Loss = 0.0295
Step 13: x = 3.1100, Loss = 0.0189
Step 14: x = 3.0880, Loss = 0.0121
Step 15: x = 3.0704, Loss = 0.0077
Step 16: x = 3.0563, Loss = 0.0050
Step 17: x = 3.0450, Loss = 0.0032
Step 18: x = 3.0360, Loss = 0.0020
Step 19: x = 3.0288, Loss = 0.0013
Step 20: x = 3.0231, Loss = 0.0008
```



The code demonstrates the use of the **Adam optimizer** in TensorFlow for minimizing the same quadratic loss function (x - 3) ^ 2 through optimization. The Adam optimizer is an advanced gradient descent technique that adjusts the learning rate for each parameter dynamically.

The Adam (short for Adaptive Moment Estimation) optimizer combines ideas from two other popular methods:

- **Momentum:** Uses a moving average of past gradients to smooth updates.
- **RMSprop**: Uses a moving average of the squared gradients to scale the learning rate

Adam computes adaptive learning rates for each parameter based on both first-order (mean) and second-order (variance) moments. It is generally faster and more efficient compared to traditional gradient descent and is particularly well-suited for training deep learning models.

- x: This is a TensorFlow variable initialized at x=5.0. This variable will be optimized to minimize the loss function.
- optimizer = **tf.keras.optimizers.Adam(learning_rate=0.1)**: This creates an **Adam optimizer** with a learning rate of 0.1. The Adam optimizer will be responsible for applying gradients to the variable x during optimization.
- tf**.GradientTape()**: TensorFlow's GradientTape is used to record operations on the x variable, which will allow us to compute gradients later. It watches the operations that happen inside the with block.
- loss = loss_function(x): The loss function (x - 3) ^ 2is computed based on the current value of x.

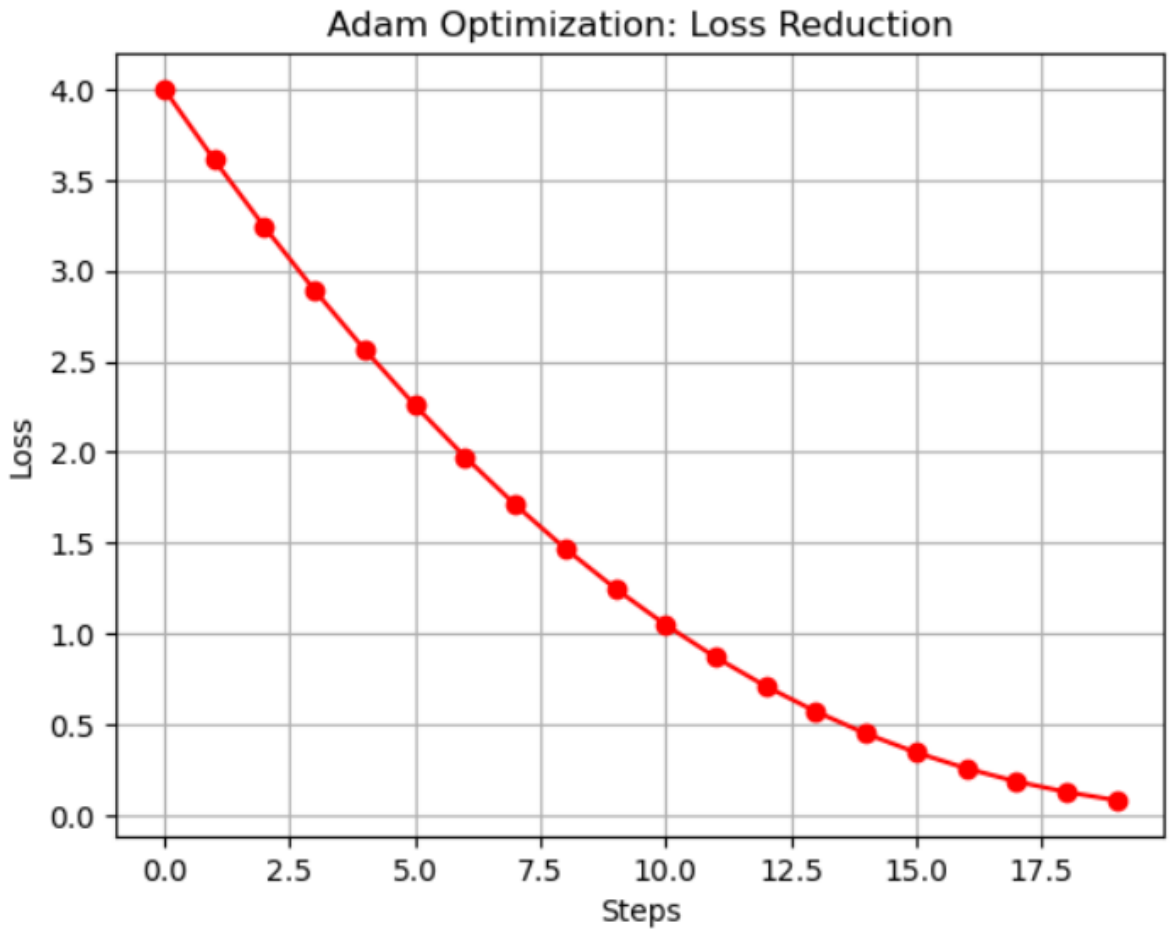
- gradients = tape.gradient(loss, [x]): The gradient of the loss function with respect to x is computed. The gradient of (x - 3) ^ 2 is 2(x−3), which tells us how to adjust x to reduce the loss.
- optimizer.apply_gradients(zip(gradients, [x])): The Adam optimizer is used to update x by applying the gradients. The optimizer uses both the first-order moment (mean) and second-order moment (variance) to adjust x more effectively than traditional gradient descent.
- Storing Loss and Step: The current step number and the corresponding loss value are stored in adam_steps and adam_loss_values for later plotting.
- print(): After each step, the current value of x and the loss are printed, showing how the optimization progresses.

```
# Using Tensorflow Optimizers

x = tf.Variable(initial_value = 5.0, trainable = True, dtype = tf.float32)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.1)

adam_steps = []

adam_loss_values = []

for step in range(20):

   with tf.GradientTape() as tape:

      loss = loss_function(x)
```

```
    gradients = tape.gradient(loss, [x])
    optimizer.apply_gradients(zip(gradients, [x]))
    adam_steps.append(step)
    adam_loss_values.append(loss.numpy())
    print(f"Step {step+1} (Adam): x = {x.numpy():.4f}, Loss = {loss.numpy():.4f}")
plt.plot(adam_steps, adam_loss_values, marker='o', color='red')
plt.title('Adam Optimization: Loss Reduction')
plt.xlabel('Steps')
plt.ylabel('Loss')
plt.grid('True')
plt.show()
print("Deep Marathe - 53004230016")
```

Output:-

```
Step 1 (Adam): x = 4.9000, Loss = 4.0000
Step 2 (Adam): x = 4.8002, Loss = 3.6100
Step 3 (Adam): x = 4.7006, Loss = 3.2406
Step 4 (Adam): x = 4.6015, Loss = 2.8921
Step 5 (Adam): x = 4.5030, Loss = 2.5648
Step 6 (Adam): x = 4.4051, Loss = 2.2589
Step 7 (Adam): x = 4.3082, Loss = 1.9744
Step 8 (Adam): x = 4.2123, Loss = 1.7114
Step 9 (Adam): x = 4.1177, Loss = 1.4698
Step 10 (Adam): x = 4.0246, Loss = 1.2493
Step 11 (Adam): x = 3.9331, Loss = 1.0498
Step 12 (Adam): x = 3.8435, Loss = 0.8707
Step 13 (Adam): x = 3.7560, Loss = 0.7115
Step 14 (Adam): x = 3.6709, Loss = 0.5716
Step 15 (Adam): x = 3.5884, Loss = 0.4501
Step 16 (Adam): x = 3.5086, Loss = 0.3462
Step 17 (Adam): x = 3.4319, Loss = 0.2587
Step 18 (Adam): x = 3.3585, Loss = 0.1866
Step 19 (Adam): x = 3.2886, Loss = 0.1285
Step 20 (Adam): x = 3.2225, Loss = 0.0833
```



Deep Marathe - 53004230016

Practical: 3

Aim:- Implement a CNN for MNIST digit classification and analyze its performance using accuracy and loss visualization.

Theory:-

Applying convolutional neural network (CNN) on the MNIST dataset for handwritten digit classification using TensorFlow. It also performs some data augmentation and visualizes the model's performance during training.

- **MNIST Dataset**: This code loads the MNIST dataset using TensorFlow's tf.keras.datasets.mnist.load_data() function. The dataset contains 60,000 training images and 10,000 test images of handwritten digits (0-9).
- **Normalization**: The pixel values are in the range 0 to 255. The code normalizes the pixel values by dividing by 255, so the values are now between 0 and 1.
- **Adding Channel Dimension**: The MNIST images are grayscale, and their shape is initially (28, 28). To work with convolutional layers, we add a new axis at the end to represent the channel (which is 1 for grayscale images). This results in shapes (28, 28, 1).
- **Printing Shape**: This line prints the shape of the train and test datasets. After adding the channel dimension, each image has the shape (28, 28, 1).
- **Creating TensorFlow Datasets**:
    - tf.data.Dataset.from_tensor_slices() is used to convert the mnist_train and mnist_test data into tf.data.Dataset objects. This provides an efficient way to handle data in batches.
    - **Shuffling**: The training dataset is shuffled with a buffer size of 10,000. Shuffling helps prevent the model from learning the order of the data and improves generalization.
    - **Batching**: Both the training and test datasets are batched into chunks of 32 samples (batch_size=32), which is a standard size for training in deep learning.
- **Data Augmentation**: The augment() function is defined to perform random data augmentation on the images during training:
    - **Random Horizontal Flip**: The image is randomly flipped horizontally, which helps the model learn more robust features.
    - **Random Brightness**: Randomly adjusts the brightness of the image by a small value (up to 0.1).
    - These transformations are applied only to the training data, and they help increase the diversity of the training dataset and improve the model's ability to generalize.

- **Augmented Dataset**: The training dataset is augmented using map(augment), meaning each image in the training set will undergo the transformations defined in augment()
- **Sequential Model**: A Sequential model is created where layers are stacked one after another. The model consists of:
    - **Conv2D Layer**: The first convolutional layer applies 32 filters of size 3×33 \times 33×3 to the input image, followed by ReLU activation.
    - **MaxPooling2D Layer**: The first pooling layer reduces the spatial dimensions by applying a 2×22 \times 22×2 pooling window.
    - **Second Conv2D Layer**: The second convolutional layer applies 64 filters of size 3×33 \times 33×3, again followed by ReLU activation.
    - **Second MaxPooling2D Layer**: Another pooling layer reduces the spatial dimensions.

    - **Flatten Layer**: The output of the convolutional layers is flattened into a 1D vector to be passed to fully connected layers.
    - **Dense Layer**: A fully connected layer with 128 neurons, using ReLU activation.
    - **Final Dense Layer**: The output layer has 10 neurons (one for each digit from 0 to 9), using the softmax activation function to produce a probability distribution over the classes.

- **Optimizer**: The Adam optimizer is used for training. Adam adapts the learning rate dynamically and is widely used in deep learning for faster convergence.
- **Loss Function**: The loss function used is sparse_categorical_crossentropy, which is appropriate for multi-class classification where the target labels are integers (0-9 in this case).
- **Metrics**: Accuracy is tracked as a performance metric.
- **Training**: The model is trained for 10 epochs using the training dataset. The validation dataset is passed during training so that the model's performance on unseen data can be evaluated after each epoch.
- **Training Curves**: After training, the accuracy and loss for both the training and validation datasets are plotted for each epoch.
    - **Left Plot (Accuracy)**: Shows how the model's accuracy on both the training and validation sets evolves over time.
    - **Right Plot (Loss)**: Shows the model's loss for both datasets during training.

Code:-

```
import tensorflow as tf

from tensorflow.keras import layers, models

import matplotlib.pyplot as plt

#-----------------part 3

print("=====part3:Dataset Loading and preprocessing===")

#3.1 load datasets (eg. MNIST, CIFAR-10)

#load mnist datasets

(mnist_train, mnist_train_labels), (mnist_test, mnist_test_labels) =
tf.keras.datasets.mnist.load_data()

#normalise

mnist_train = mnist_train/255.0

mnist_test = mnist_test/255.0

#expand dimensu=ions (eg. MNIST images are grayscale)

mnist_train = mnist_train[..., tf.newaxis]

print(f"MINST Train Shape: {mnist_train.shape}, MNIST Test Shape: {mnist_test.shape}")
```

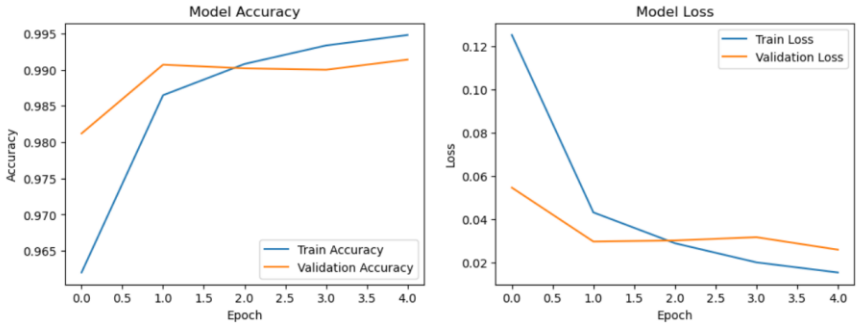
```
#3.2 Create a custom dataset using tf.data Datasets
batch_size = 32
train_dataset = tf.data.Dataset.from_tensor_slices((mnist_train, mnist_train_labels))
train_dataset = train_dataset.shuffle(buffer_size=10000).batch(batch_size)
test_dataset = tf.data.Dataset.from_tensor_slices((mnist_test, mnist_test_labels))
test_dataset = test_dataset.batch(batch_size)
#3.3 perform Data Augmentation using TensorFlow operations
def augment(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_brightness(image, max_delta=0.1)
    return image, label
augment_train_dataset = train_dataset.map(augment)
#---------------------------------- Part 4:Build and train simple model----------#
print("¥n== Part 4:Build and Train a Simple Model==")
#4.1 Create a Neural Network to Classify MNIST using the Sequential API
model = models.Sequential([
layers.Input(shape=(28, 28, 1)),
layers.Conv2D(64, kernel_size=(3,3), activation='relu'),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(3,3), activation='relu'),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Flatten(),
layers.Dense(128, activation='relu'),
layers.Dense(10, activation='softmax') # 10 classes for MNIST digits
])
#Compile the model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
# 4.2 Train the Model
history = model.fit(train_dataset, epochs=5, validation_data=test_dataset)
```

#4.3 Evaluate any Visualise Model Performance

```
plt.figure(figsize=(12, 4))
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
print("Deep Marathe - 53004230016")
```

Output:-

```
=====part3:Dataset Loading and preprocessing===
MINST Train Shape: (60000, 28, 28, 1), MNIST Test Shape: (10000, 28, 28)
¥n== Part 4:Build and Train a Simple Model==
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 14ms/step - accuracy: 0.9152 - loss: 0.2748 - val_accuracy: 0.9812 - val_loss: 0.0547
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9852 - loss: 0.0487 - val_accuracy: 0.9907 - val_loss: 0.0298
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9907 - loss: 0.0301 - val_accuracy: 0.9902 - val_loss: 0.0303
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9929 - loss: 0.0204 - val_accuracy: 0.9900 - val_loss: 0.0318
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9950 - loss: 0.0151 - val_accuracy: 0.9914 - val_loss: 0.0261
```




```
Deep Marathe - 53004230016
```

Practical: 4

Aim:- Perform regularization to prevent model from over-fitting.

Theory:-

Regularization is a technique used in machine learning to prevent overfitting by adding some form of penalty to the loss function. The goal is to reduce the complexity of the model, ensuring that it generalizes well to new, unseen data rather than fitting too closely to the training data. Regularization helps combat overfitting, which occurs when a model learns to perform very well on the training data but fails to generalize to unseen data (test data). Regularization techniques like **L2 regularization (Ridge regression)** penalize the size of the weights, which prevents the model from becoming too complex and overfitting the training data.

In this case, **L2 regularization** is applied to the first two Dense layers of the model:

- The regularization term added to the loss function is proportional to the square of the weights. By penalizing large weights, the model is encouraged to learn simpler, more generalized patterns rather than memorizing the training data.
- **Impact on Training and Validation Loss**: With regularization, you might observe that the gap between the training loss and validation loss is smaller, and the model is less likely to overfit, especially if you train it for many epochs.

- **Loading the MNIST dataset**: The MNIST dataset contains images of handwritten digits. The data is split into training (60,000 images) and test (10,000 images) sets.
- **Normalization**: The pixel values range from 0 to 255. By dividing by 255.0, the pixel values are normalized to the range [0, 1], which is a common preprocessing step in neural networks to improve convergence.
- **Reshaping**: The images, originally 28x28 pixels, are flattened into 784-dimensional vectors (28 x 28 = 784). This flattening is required for fully connected (Dense) layers in a feed-forward neural network.
- **One-Hot Encoding**: The MNIST labels are integers (0-9), but neural networks require categorical labels as a one-hot encoded vector (i.e., a vector where only the index corresponding to the class is 1, and all others are 0). For example, the label 2 is converted to [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].
- **Model Architecture**: A simple feed-forward neural network is built using the Sequential model:
    - **First Dense Layer**: A fully connected layer with 128 units and ReLU activation. The input_shape=(784,) indicates the input dimension (784 features from the flattened MNIST image). The kernel_regularizer=tf.keras.regularizers.l2(0.01) applies L2 regularization with a weight decay factor of 0.01. This penalizes large weights during training and helps prevent overfitting by adding a term proportional to the sum of the squared weights to the loss function.
    - **Second Dense Layer**: Similar to the first layer, but with 64 units and the same L2 regularization.
    - **Output Layer**: A softmax output layer with 10 units (one for each digit class).
- **Optimizer:** The Adam optimizer is used with a learning rate of 0.001. Adam is an adaptive optimization algorithm that adjusts the learning rate during training based on the gradient information.

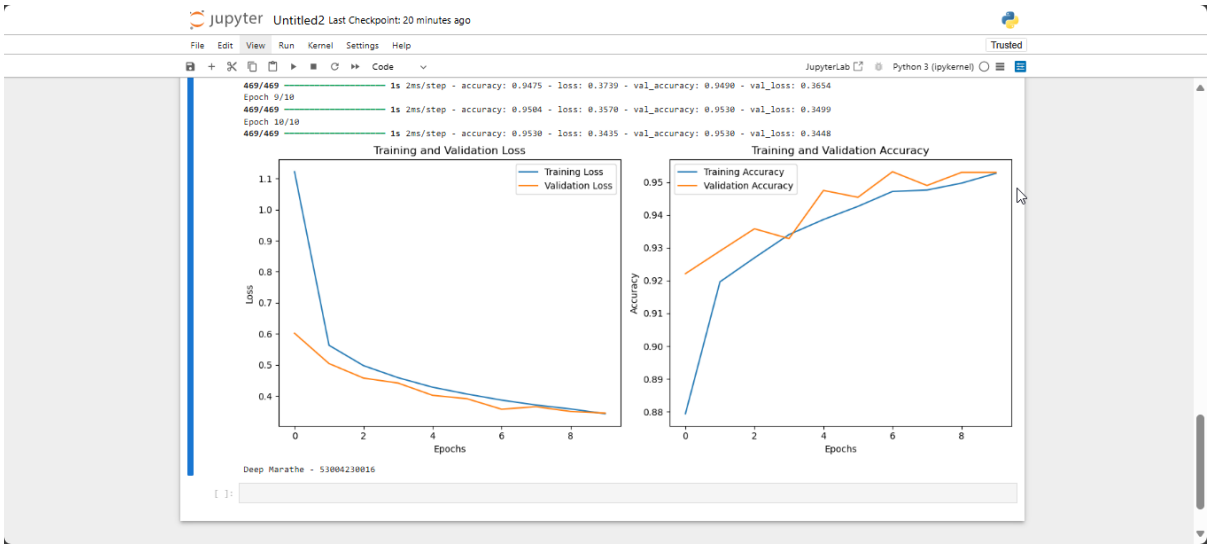
- **Loss Function:** The loss function is categorical cross-entropy, which is appropriate for multi-class classification tasks like MNIST. This loss function measures how well the predicted class probabilities match the true labels.
- **Metrics:** The accuracy of the model is tracked during training to evaluate how well the model performs.
- **Training:** The model is trained for 10 epochs with a batch size of 128. During training, the model learns from the training data (train_data) and updates its weights to minimize the loss. The validation data (test_data) is used to evaluate the model after each epoch, allowing us to check for overfitting by comparing training and validation performance.
- **Left Plot (Loss)**: The training and validation loss are plotted over the epochs. Ideally, the training loss should decrease over time, and the validation loss should decrease or stabilize (it should not increase significantly). A significant gap between training and validation loss could indicate overfitting.
- **Right Plot (Accuracy)**: Similarly, the training and validation accuracy are plotted. The training accuracy should increase, and the validation accuracy should ideally increase as well, although it may fluctuate more.

Code:-

```
import tensorflow as tf

##Load MNIST data

(train_data, train_labels), (test_data, test_labels) = tf.keras.datasets.mnist.load_data()

##Normalize and flatten the images

train_data = train_data.reshape((60000, 784)) / 255.0

test_data = test_data.reshape((10000, 784)) / 255.0

One-hot encode labels

train_labels = tf.keras.utils.to_categorical(train_labels, 10)

test_labels = tf.keras.utils.to_categorical(test_labels, 10)

##Build the model

model = tf.keras.models.Sequential([

tf.keras.layers.Dense(128, activation='relu', input_shape=(784,),
kernel_regularizer=tf.keras.regularizers.l2(0.01)),

tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),

tf.keras.layers.Dense(10, activation='softmax')


])

##Compile the model

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
```

```
        loss='categorical_crossentropy',
        metrics=['accuracy'])

##Train the model
history = model.fit(train_data, train_labels, epochs=10, batch_size=128, validation_data=(test_data, test_labels))
import matplotlib.pyplot as plt
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
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('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.tight_layout()
plt.show()
print('Deep Marathe - 53004230016')
```

Output:-


jupyter Untitled2 Last Checkpoint: 20 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)
469/469 1s 2ms/step - accuracy: 0.9475 - loss: 0.3739 - val_accuracy: 0.9490 - val_loss: 0.3654
Epoch 9/10
469/469 1s 2ms/step - accuracy: 0.9504 - loss: 0.3570 - val_accuracy: 0.9530 - val_loss: 0.3499
Epoch 10/10
469/469 1s 2ms/step - accuracy: 0.9530 - loss: 0.3435 - val_accuracy: 0.9530 - val_loss: 0.3448
Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs
Deep Marathe - 53004230016

Practical: 5

Aim:- Implement a CNN for MNIST digit classification, train and evaluate the model, and predict sample images.

Theory:-

The task at hand is to implement a Convolutional Neural Network (CNN) for recognizing handwritten digits from the **MNIST dataset**. The MNIST (Modified National Institute of Standards and Technology) dataset is a large collection of 28x28 grayscale images of handwritten digits (0 through 9). This dataset is often used as a benchmark for evaluating machine learning models, especially for image classification tasks.

In this case, we are using **Convolutional Neural Networks (CNNs)**, which are a class of deep neural networks that are particularly effective for processing images due to their ability to detect patterns such as edges, textures, and shapes through convolution operations.

- **Loading the MNIST Dataset:**
  - The MNIST dataset is loaded using keras.datasets.mnist.load_data(). This function returns training and test data as tuples of image arrays and corresponding labels.
  - x_train and x_test contain the images of the digits (28x28 pixel grayscale images).
  - y_train and y_test contain the corresponding labels (the actual digit in each image).

- **Preprocessing the Data:**
  - The pixel values of the images are normalized to the range [0, 1] by dividing by 255. This is common practice to ensure that the model learns efficiently without large scale values.
  - The images are also reshaped to have an additional channel dimension, as CNNs expect 3D input: height, width, and channels (grayscale has 1 channel). So, the shape of the images becomes (28, 28, 1) instead of just (28, 28).

- **Building the CNN Model:**
  - **Conv2D Layer**: A 2D convolution layer with 32 filters of size (3, 3) is applied. This layer is responsible for extracting features from the input image. The relu activation function introduces non-linearity to the model.
  - **MaxPooling2D Layer**: After each convolution layer, max pooling is applied with a pool size of (2, 2). This helps reduce the spatial dimensions (height and width) of the feature maps while retaining the important information.
  - **Conv2D and MaxPooling2D Layers**: A second set of convolution and pooling layers is applied with 64 filters. This further extracts high-level features from the image.
  - **Flatten Layer**: After feature extraction, the 2D feature maps are flattened into a 1D vector to feed into the fully connected layers.
  - **Dense Layers**: These are fully connected layers. The first dense layer has 64 neurons with relu activation, and the second dense layer has 10 neurons with softmax activation.

The 10 output neurons correspond to the 10 digit classes (0-9), and the softmax activation ensures the output is a probability distribution over the classes.

- **Compiling the Model:**
  - The model is compiled with the **Adam optimizer**, which is an adaptive learning rate optimization algorithm.
  - The loss function used is **sparse categorical crossentropy**, which is appropriate for multi-class classification problems where the target labels are integers (as in the case of the MNIST labels).
  - We track accuracy as a performance metric.

- **Training the Model:**
  - The model is trained using the training data (x_train and y_train) for **15 epochs** with a batch size of 128.
  - The validation data (x_test and y_test) is used to evaluate the model's performance after each epoch.
  - The model's training history, including loss and accuracy over epochs, is saved in history.

- **Evaluating the Model:**
  - After training, the model is evaluated on the test data (x_test and y_test) to check its accuracy on unseen data.
  - The accuracy is printed out.

- **Making Predictions on a Sample Image:**
  - A sample image from the test set is chosen (x_test[0]), and its true label is obtained (y_test[0]).
  - The model makes a prediction on this sample image, and the predicted label is extracted by taking the index of the highest probability (using np.argmax).

- **Visualizing the Sample Image:**
  - The sample image is displayed using Matplotlib. The. squeeze() method is used to remove the extra dimensions added earlier (such as the batch dimension).

Key Concepts in CNNs:

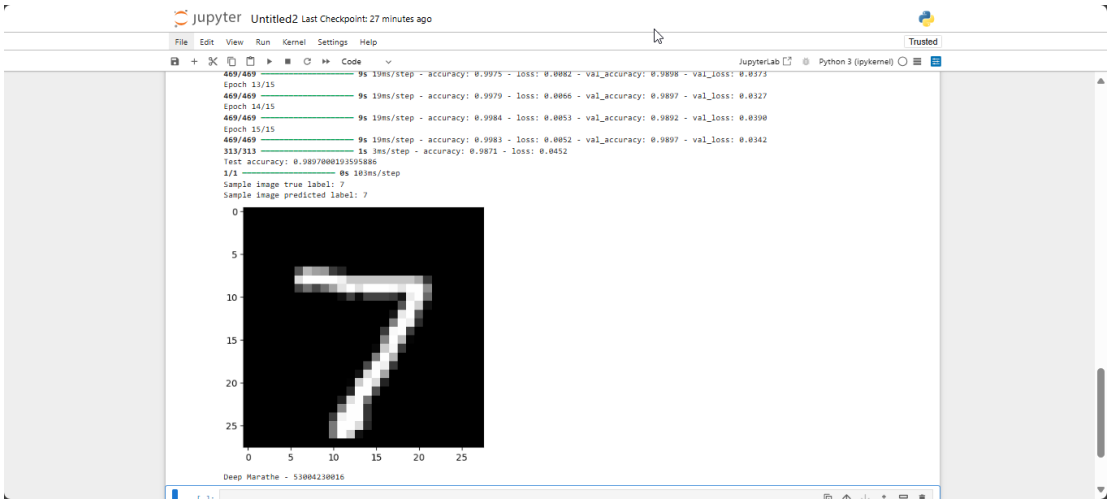
- **Convolution:** This is the process where the network scans the image with a filter (kernel) to extract important features such as edges and textures. The output of the convolution layer is a feature map that contains learned patterns from the image.
- **Max Pooling:** This operation reduces the spatial dimensions (height and width) of the feature map, reducing the number of parameters and computations while retaining the most important information.
- **ReLU Activation:** The **Rectified Linear Unit (ReLU)** is a non-linear activation function that introduces non-linearity into the network, allowing it to learn complex patterns.
- **Softmax Activation:** The **softmax function** is used in the final layer of classification models to output probabilities for each class, ensuring the predicted values sum to 1.

Code:-

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.astype("float32")/255.0
x_test = x_test.astype("float32")/255.0
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
model = keras.models.Sequential([
keras.layers.Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1)),
keras.layers.MaxPooling2D((2,2)),
keras.layers.Conv2D(64, (3,3), activation="relu"),
keras.layers.MaxPooling2D((2,2)),
keras.layers.Flatten(),
keras.layers.Dense(64, activation="relu"),
keras.layers.Dense(10, activation="softmax")

])
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=['accuracy'])
history = model.fit(x_train, y_train, epochs=15, batch_size=128, validation_data=(x_test, y_test))
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
sample_img = x_test[0]
sample_label = y_test[0]
sample_img = np.expand_dims(sample_img, 0)
pred = model.predict(sample_img)
pred_label = np.argmax(pred)
```

```
print("Sample image true label:", sample_label)

print("Sample image predicted label:", pred_label)

plt.imshow(sample_img.squeeze(), cmap='gray')

plt.show()

print('Deep Marathe - 53004230016')
```

Output:-



Practical: 6

Aim:- Implement time series forecasting using RNN and ARIMA models to predict airline passenger trends and compare their performance.

Theory:-

Implementing **Recurrent Neural Network (RNN)** for predicting airline passenger traffic based on historical data. The dataset we are using contains monthly airline passenger counts from 1949 to 1960. We will compare the performance of the RNN model with the **ARIMA** (AutoRegressive Integrated Moving Average) model, which is a traditional statistical model for time series forecasting. Time series forecasting refers to predicting future values based on previously observed values in the series. **RNNs**, particularly variants like **LSTM (Long Short-Term Memory)**, are very effective for time series data because they have the ability to learn patterns over time by maintaining memory of previous states.

- **RNNs for Time Series Forecasting**: RNNs are used for sequential data and time series problems due to their ability to "remember" past inputs. They are suitable for tasks like predicting future values based on historical data.

- **ARIMA Model**: ARIMA is a classical statistical model used for time series forecasting. It combines three components:

  - **AR (AutoRegressive)**: Uses the relationship between an observation and a number of lagged observations.
  - **I (Integrated)**: Represents the differencing of raw observations to make the time series stationary.
  - **MA (Moving Average)**: Models the relationship between an observation and a residual error from a moving average model applied to lagged observations.
  - **RMSE**: A measure of the differences between predicted and actual values, giving an indication of the model's prediction accuracy.

- **Building the RNN Model:**
  - **SimpleRNN Layers**: A simple RNN model is built using two RNN layers, each with 50 units. The return_sequences=True argument in the first RNN layer ensures that the output of each time step is passed on to the next layer. The second RNN layer doesn't return sequences (return_sequences=False), which means it will output the final state after processing all time steps.
  - **Dense Layers**: A dense layer with 25 neurons and another dense output layer with a single neuron that predicts the future value (passenger count for the next month).
  - **Compilation**: The model is compiled using the **Adam optimizer** and **mean squared error** as the loss function, which is appropriate for regression tasks.

- **Making Predictions and Inverse Transforming the Data:**
  - After training, predictions are made on both the training and test datasets.
  - The predictions are then **inverse transformed** to the original scale of the passenger data using scaler.inverse_transform. This step is necessary since the data was scaled earlier.

- **Calculating RMSE (Root Mean Squared Error):**
  - **RMSE** is calculated for both the training and test predictions. RMSE is a common metric to evaluate regression models, as it represents the square root of the average squared differences between predicted and actual values. A lower RMSE indicates better performance.
- **Comparing ARIMA Model:**
  - An **ARIMA model** is fit to the original dataset (df) with parameters (5,1,0). These parameters represent the order of the AR, I, and MA components, respectively.
  - Predictions are made for the same length as the test set and RMSE is calculated to evaluate the ARIMA model's performance.

Code:-

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler
```

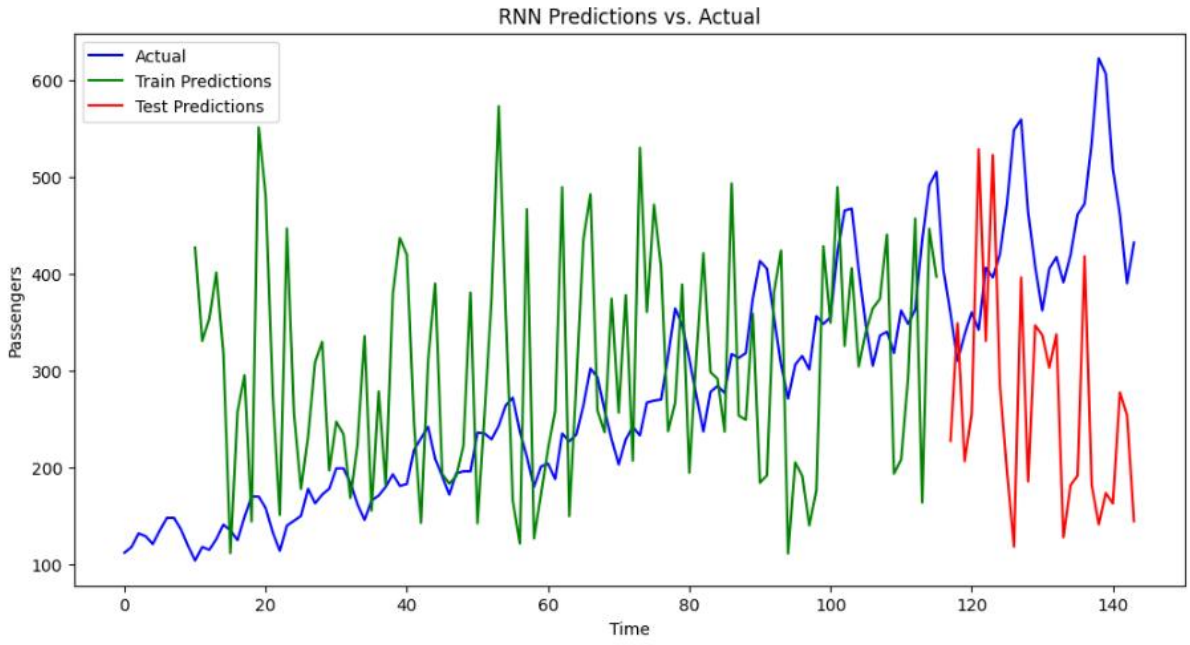
```
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, LSTM
from sklearn.model_selection import train_test_split
from statsmodels.tsa.arima.model import ARIMA
import seaborn as sns
url= 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
df = pd.read_csv(url,usecols=['Passengers'])
df.head()
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(df)

def create_dataset(data, time_step=1):
    X, Y = [], []
    for i in range(len(data)-time_step-1):
        a = data[i:(i+time_step),0]
        X.append(a)
        Y.append(data[i+time_step,0])
    return np.array(X), np.array(Y)

time_step=10
X, Y = create_dataset(scaled_data, time_step)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1],1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1],1)

model = Sequential()
model.add(SimpleRNN(50, return_sequences=True, input_shape=(time_step,1)))
model.add(SimpleRNN(50, return_sequences=False))
model.add(Dense(25))
```

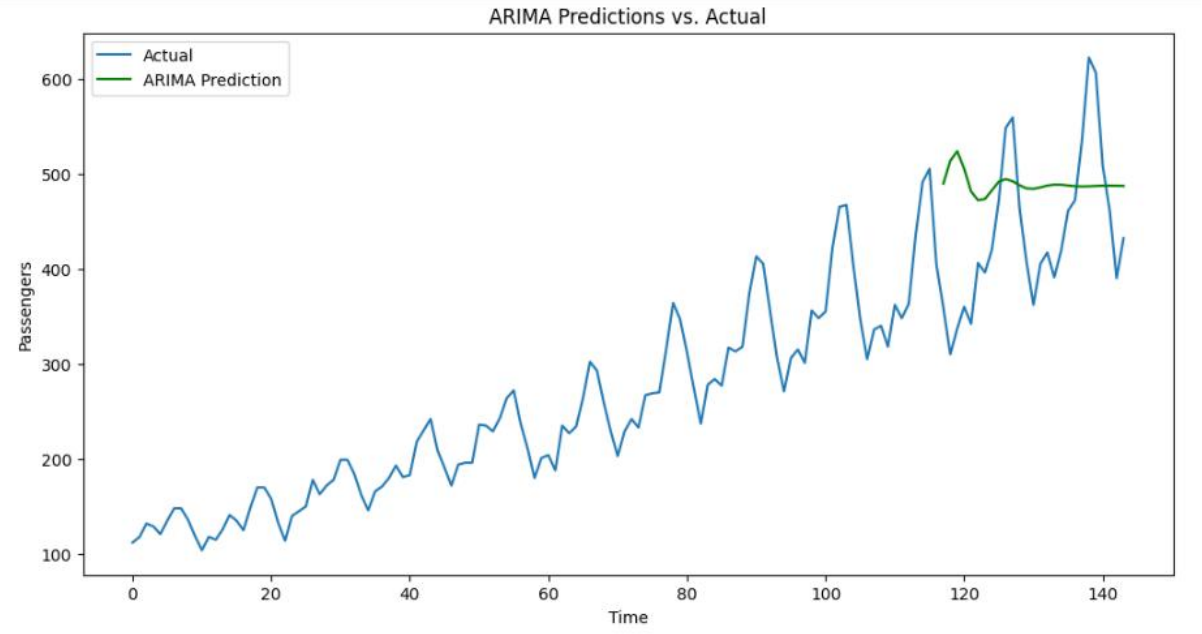
```
model.add(Dense(1))

model.compile(optimizer='adam', loss='mean_squared_error')

model.fit(X_train, Y_train, epochs=100, batch_size=32, validation_data=(X_test, Y_test),
verbose=1)

train_predict = model.predict(X_train)

test_predict = model.predict(X_test)

train_predict = scaler.inverse_transform(train_predict)

test_predict = scaler.inverse_transform(test_predict)

Y_train = scaler.inverse_transform([Y_train])

Y_test = scaler.inverse_transform([Y_test])

train_rmse = np.sqrt(mean_squared_error(Y_train[0], train_predict[:,0]))

test_rmse = np.sqrt(mean_squared_error(Y_test[0], test_predict[:,0]))

print(f'Train RMSE: {train_rmse}')

print(f'Test RMSE: {test_rmse}')

model_arima = ARIMA(df, order=(5,1,0))

model_arima_fit = model_arima.fit()

arima_pred = model_arima_fit.forecast(steps=len(X_test))

arima_rmse = np.sqrt(mean_squared_error(df[-len(arima_pred):], arima_pred))

print(f'ARIMA RMSE: {arima_rmse}')

plt.figure(figsize=(12,6))

plt.plot(df, label='Actual', color='blue')

plt.plot(range(time_step, time_step+len(train_predict)), train_predict, color='green',
label='Train Predictions')

plt.plot(range(len(df)-len(test_predict), len(df)), test_predict, color='red', label='Test
Predictions')

plt.legend()

plt.title('RNN Predictions vs. Actual')

plt.xlabel('Time')

plt.ylabel('Passengers')

plt.show()
```

```
plt.figure(figsize=(12,6))

plt.plot(df, label='Actual')

plt.plot(range(len(df)-len(arima_pred), len(df)), arima_pred, color='green', label='ARIMA Prediction')

plt.legend()

plt.title('ARIMA Predictions vs. Actual')

plt.xlabel('Time')

plt.ylabel('Passengers')

plt.show()

print('Viraj Joshi - 53004230033')
```

Output:-

```
Epoch 1/100
C:\Users\DELL\anaconda3\envs\python39\lib\site-packages\keras\src\layers\
t to a layer. When using Sequential models, prefer using an `Input(shape)
  super().__init__(**kwargs)
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 254ms/step - loss: 0.7344 - val_loss: 0.3384
Epoch 2/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step - loss: 0.3061 - val_loss: 0.0238
Epoch 3/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 0.0243 - val_loss: 0.0662
Epoch 4/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step - loss: 0.0947 - val_loss: 0.0187
Epoch 5/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - loss: 0.0199 - val_loss: 0.0199
Epoch 6/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - loss: 0.0278 - val_loss: 0.0238
Epoch 7/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 0.0229 - val_loss: 0.0061
Epoch 8/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 0.0067 - val_loss: 0.0134
Epoch 9/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step - loss: 0.0144 - val_loss: 0.0076
Epoch 10/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - loss: 0.0059 - val_loss: 0.0068

Epoch 95/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - loss: 0.0020 - val_loss: 0.0019
Epoch 96/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - loss: 0.0023 - val_loss: 0.0019
Epoch 97/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - loss: 0.0023 - val_loss: 0.0018
Epoch 98/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 0.0023 - val_loss: 0.0017
Epoch 99/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - loss: 0.0021 - val_loss: 0.0017
Epoch 100/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step - loss: 0.0020 - val_loss: 0.0018
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 355ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
Train RMSE: 23.23939814270194
Test RMSE: 21.82204568843397
ARIMA RMSE: 96.63626313278345
```





Viraj Joshi - 53004230033

Practical: 7

Aim:- Perform stock price prediction using LSTM and GRU models, compare their performance, and forecast future prices.

Theory:-

Two types of **Recurrent Neural Networks (RNNs)** — **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Units)** — to predict the stock prices of **Apple Inc. (AAPL)**. These models are popular for time series forecasting because they can handle sequential data, which makes them suitable for stock price predictions, weather forecasting, etc.

1. **LSTM (Long Short-Term Memory):**
    - **LSTM** is a type of RNN designed to solve the problem of vanishing gradients, which can occur in traditional RNNs when dealing with long sequences of data.
    - LSTM units have memory cells that can store information over long periods. They utilize a combination of **gates** (input, output, and forget) that control the flow of information. This makes LSTM particularly suitable for problems where long-term dependencies exist, such as time series forecasting.
    - The gates in LSTM regulate how information is remembered and how much influence past states should have on future predictions.

2. **GRU (Gated Recurrent Units):**
    - **GRU** is a simpler variant of LSTM and also solves the vanishing gradient problem. However, it combines the input and forget gates into a single update gate and merges the cell state and hidden state.
    - GRUs are computationally more efficient than LSTMs since they use fewer parameters, while still achieving similar performance for many tasks.

Both models are used for sequential tasks, but LSTMs are generally more powerful for learning complex patterns in long sequences, whereas GRUs can be faster and more efficient in practice.

- **Building the LSTM and GRU Models:**
    - **LSTM Model:**
        - A **Sequential** model is created, and an **LSTM layer** with 50 units is added.
        - The output of the LSTM is connected to a **Dense layer** with a single neuron that outputs the predicted stock price.
        - The model is compiled using the **Adam optimizer** and **mean squared error** as the loss function, which is typical for regression problems.
    - **GRU Model:**
        - The **GRU layer** works similarly to the LSTM layer but uses fewer parameters. It is also followed by a **Dense layer** to output the predicted stock price.

Code:-

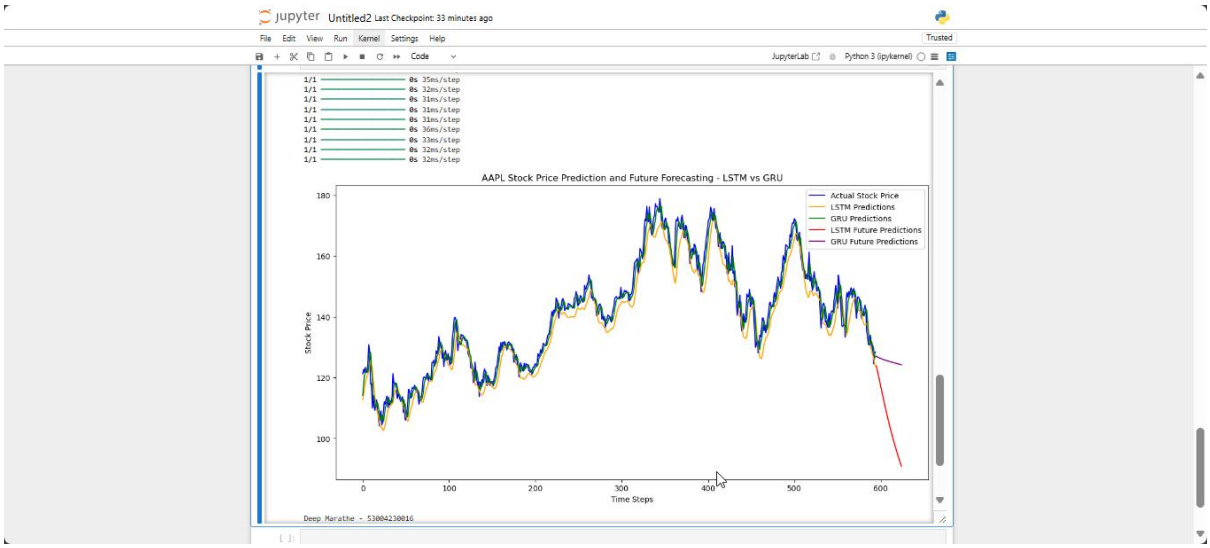
```
import yfinance as yf

import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, GRU, Dense
stock_symbol = 'AAPL'
df = yf.download(stock_symbol, start='2010-01-01', end='2023-01-01')
data = df[['Close']].values
scaler = MinMaxScaler(feature_range=(0,1))
data_scaled = scaler.fit_transform(data)
train_size = int(len(data_scaled)*0.8)
train, test = data_scaled[0:train_size], data_scaled[train_size:]
def create_dataset(dataset, look_back=60):
X, Y = [], []
for i in range(len(dataset)-look_back):
   X.append(dataset[i:i+look_back, 0])
   Y.append(dataset[i + look_back, 0])
return np.array(X), np.array(Y)

look_back = 60
X_train, Y_train = create_dataset(train, look_back)
X_test, Y_test = create_dataset(test, look_back)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
model_lstm = Sequential()
model_lstm.add(LSTM(50, input_shape=(look_back, 1)))
model_lstm.add(Dense(1))
model_lstm.compile(loss='mean_squared_error', optimizer='adam')
model_gru = Sequential()
model_gru.add(GRU(50, input_shape=(look_back, 1)))
```

```
model_gru.add(Dense(1))
model_gru.compile(loss='mean_squared_error', optimizer='adam')
model_lstm.fit(X_train, Y_train, epochs=20, batch_size=32, verbose=1)
model_gru.fit(X_train, Y_train, epochs=20, batch_size=32, verbose=1)
lstm_predictions = model_lstm.predict(X_test)
gru_predictions = model_gru.predict(X_test)
lstm_predictions = scaler.inverse_transform(lstm_predictions)
gru_predictions = scaler.inverse_transform(gru_predictions)
Y_test_actual = scaler.inverse_transform([Y_test])
future_steps = 30
last_sequence = X_test[-1]
lstm_future_predictions = []
for _ in range(future_steps):
prediction = model_lstm.predict(np.reshape(last_sequence, (1, look_back, 1)))
lstm_future_predictions.append(prediction[0][0])
last_sequence = np.append(last_sequence[1:], prediction[0])

last_sequence = X_test[-1]
gru_future_predictions = []
for _ in range(future_steps):
prediction = model_gru.predict(np.reshape(last_sequence, (1, look_back, 1)))
gru_future_predictions.append(prediction[0][0])
last_sequence = np.append(last_sequence[1:], prediction[0])

lstm_future_predictions = scaler.inverse_transform(np.array(lstm_future_predictions).reshape(-1,
1))
gru_future_predictions = scaler.inverse_transform(np.array(gru_future_predictions).reshape(-1, 1))
future_range = np.arange(len(Y_test_actual[0]), len(Y_test_actual[0])+ future_steps)
plt.figure(figsize=(14,7))
plt.plot(Y_test_actual[0], label='Actual Stock Price', color='blue')
plt.plot(lstm_predictions, label='LSTM Predictions', color='orange')
```

```
plt.plot(gru_predictions, label='GRU Predictions', color='green')
plt.plot(future_range, lstm_future_predictions, label='LSTM Future Predictions', color='red')
plt.plot(future_range, gru_future_predictions, label='GRU Future Predictions', color='purple')
plt.title(f'{stock_symbol} Stock Price Prediction and Future Forecasting - LSTM vs GRU')
plt.xlabel('Time Steps')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
print('Deep Marathe - 53004230016')
```

Output:-



Practical: 8

Aim:- Perform classification of cats and dogs using a pre-trained VGG16 model with transfer learning, data augmentation, and evaluation on validation data.

Theory:-

Transfer Learning is a deep learning technique where a pre-trained model (trained on a large dataset) is reused for a new, but related, task. Instead of training a model from scratch, TL allows leveraging learned features from a powerful neural network, reducing training time and improving accuracy with limited data.

Key Concepts:

1. **Feature Extraction** – Use the pre-trained model as a fixed feature extractor by removing its top layers and adding new task-specific layers. The pre-trained weights remain unchanged.
2. **Fine-Tuning** – Unfreeze some layers of the pre-trained model and retrain them on the new dataset, adapting to specific features of the new task.
3. **Benefits:**
    - Faster convergence and reduced computational cost.
    - Requires less labelled data.

- Improves accuracy by utilizing knowledge from large datasets.

Common Pre-Trained Models:

- **VGG16/VGG19** – Simple but large models.
- **ResNet** – Efficient deep networks with skip connections.
- **Inception, EfficientNet** – Optimized for high accuracy with fewer parameters.

In **image classification**, TL is widely used, where models trained on **ImageNet** (millions of images) are adapted for tasks like medical imaging, object detection, and more.

Code:-

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os
import zipfile
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16

# Define cache directory and filename
cache_dir = os.getcwd()  # Set the current working directory as the cache directory
filename = "cats_and_dogs_filtered.zip"  # Name of the file to be downloaded
url = "https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip"
# Download the file
file_path = tf.keras.utils.get_file(filename, url, cache_dir=cache_dir)
# Extract the file
with zipfile.ZipFile(file_path, "r") as zip_ref:
    zip_ref.extractall()
# Define directories
train_dir = os.path.join(os.getcwd(), "cats_and_dogs_filtered", "train")
validation_dir = os.path.join(os.getcwd(), "cats_and_dogs_filtered", "validation")
# Data augmentation and rescaling for training and validation data
train_datagen = ImageDataGenerator(rescale=1./255,
```

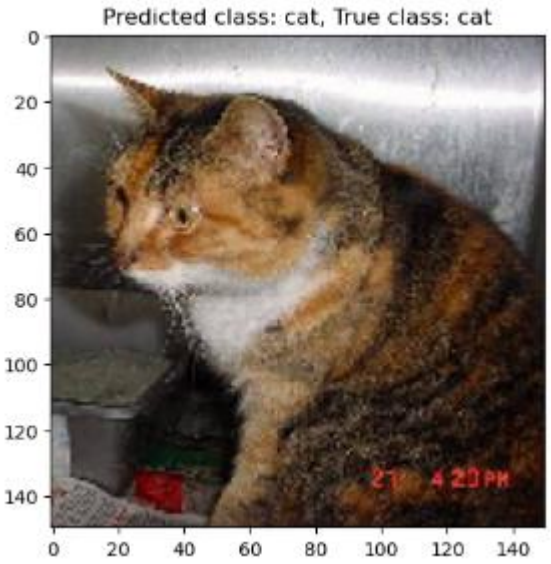
```
                    rotation_range=20,
                    width_shift_range=0.2,
                    height_shift_range=0.2,
                    shear_range=0.2,
                    zoom_range=0.2,
                    horizontal_flip=True)
validation_datagen = ImageDataGenerator(rescale=1./255)
# Image generators
train_generator = train_datagen.flow_from_directory(train_dir,
                                    target_size=(150,150),
                                    batch_size=20,
                                    class_mode="binary")
validation_generator = validation_datagen.flow_from_directory(validation_dir,
                                    target_size=(150,150),
                                    batch_size=20,
                                    class_mode="binary")

# Load pre-trained VGG16 model
conv_base = VGG16(weights="imagenet",
            include_top=False,
             input_shape=(150, 150, 3))
# Freeze the convolutional base
conv_base.trainable = False
# Build the model
model = tf.keras.models.Sequential()
model.add(conv_base)
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

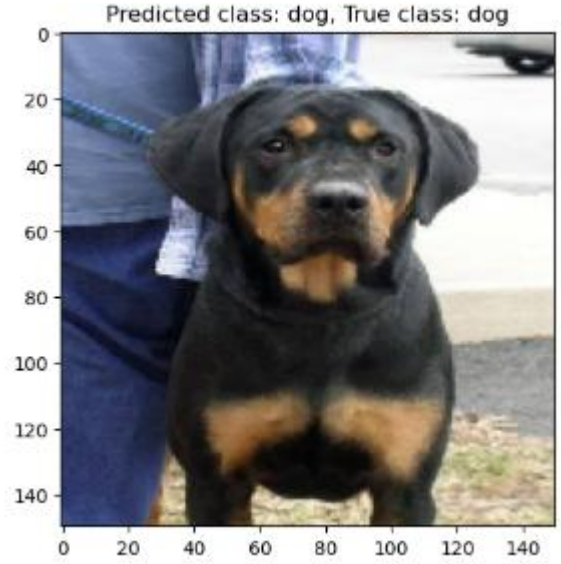
```
# Compile the model
model.compile(loss="binary_crossentropy",
        optimizer=tf.keras.optimizers.RMSprop(learning_rate=2e-5),
        metrics=["accuracy"])


# Train the model
history = model.fit(train_generator,
            steps_per_epoch=10,
            epochs=30,
            validation_data=validation_generator,
            validation_steps=50)
# Display predictions for some validation images
x, y_true = next(validation_generator)
y_pred = model.predict(x)
class_names = ['cat', 'dog']
for i in range(len(x)):
   plt.imshow(x[i])
   plt.title(f'Predicted class: {class_names[int(round(y_pred[i][0]))]}, True class:
{class_names[int(y_true[i])]}')
   plt.show()
# Plot training and validation accuracy
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(acc) + 1)
# Accuracy plot
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.legend()
```

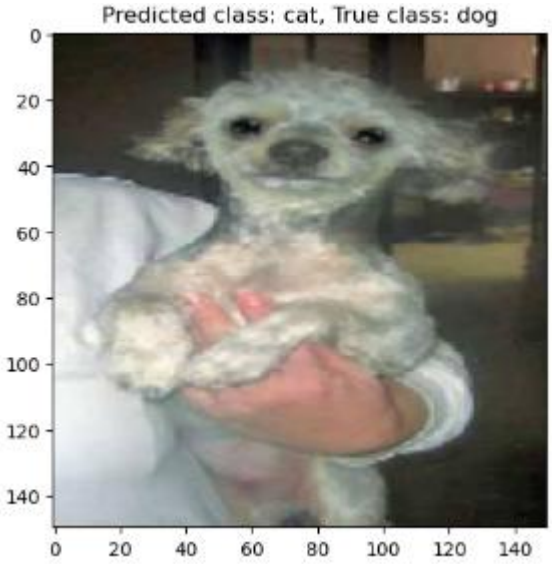
```
# Loss plot
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
print('Viraj Joshi - 53004230033')
```

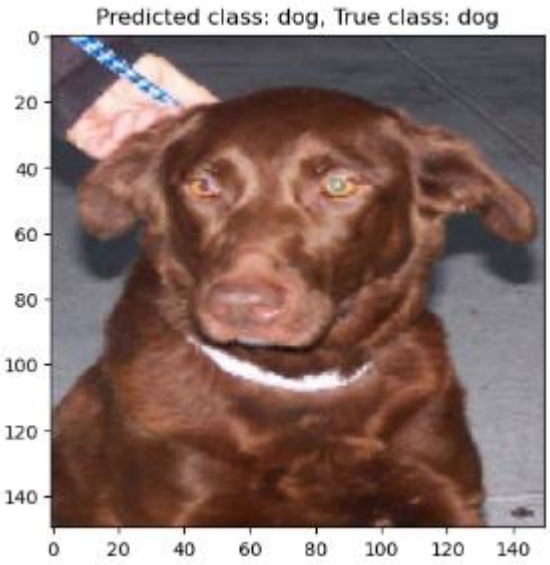
Output:-

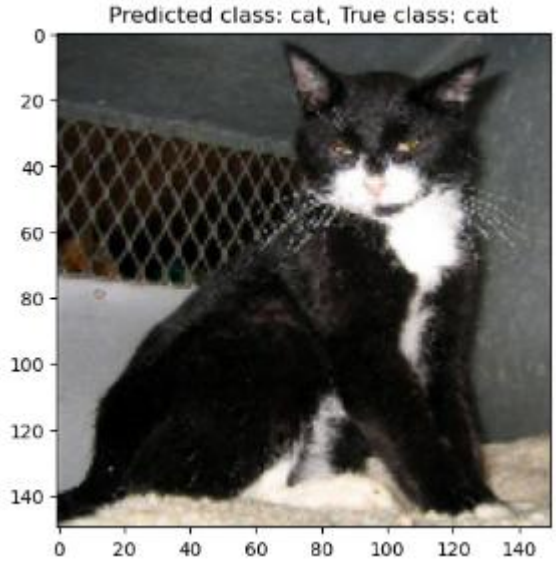
```
Epoch 27/30
10/10 ━━━━━━━━━━━━━━━━━━━━ 36s 4s/step - accuracy: 0.7695 - loss: 0.4551 - val_accuracy: 0.8160 - val_loss: 0.3909
Epoch 28/30
10/10 ━━━━━━━━━━━━━━━━━━━━ 36s 4s/step - accuracy: 0.7697 - loss: 0.5041 - val_accuracy: 0.8330 - val_loss: 0.3738
Epoch 29/30
10/10 ━━━━━━━━━━━━━━━━━━━━ 37s 4s/step - accuracy: 0.7778 - loss: 0.4364 - val_accuracy: 0.8390 - val_loss: 0.3697
Epoch 30/30
10/10 ━━━━━━━━━━━━━━━━━━━━ 36s 4s/step - accuracy: 0.7636 - loss: 0.5126 - val_accuracy: 0.8320 - val_loss: 0.3653
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 757ms/step
```


Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140

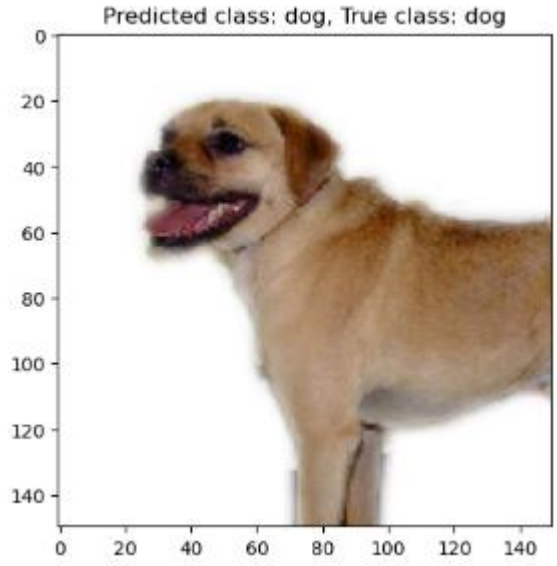

Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140

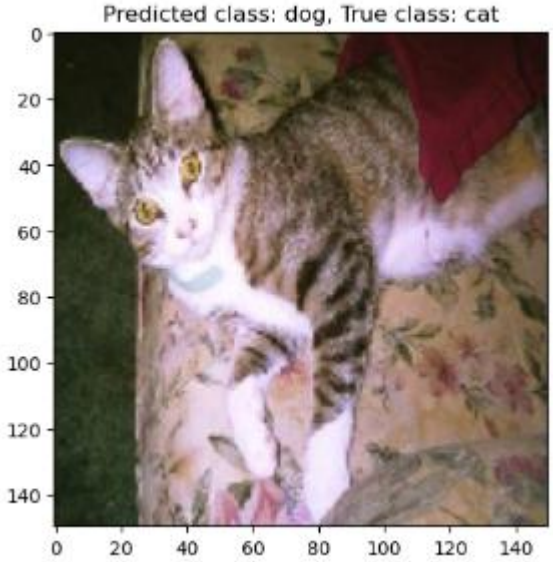

Predicted class: cat, True class: dog
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120 140

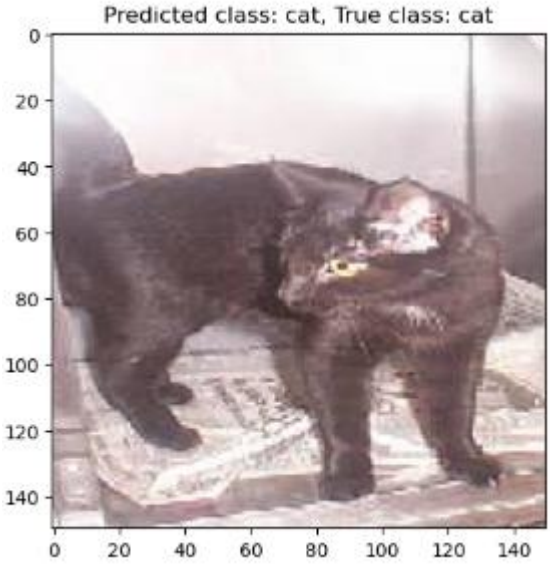

Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

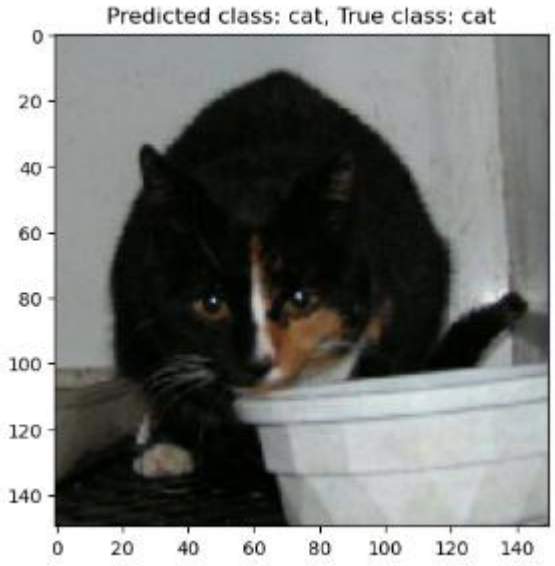

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

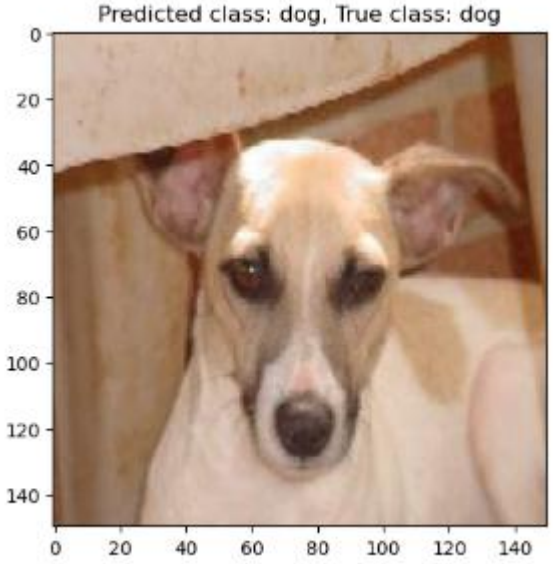

Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

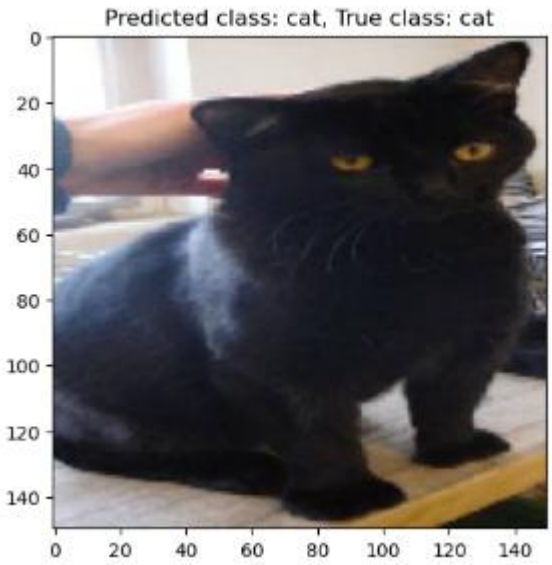

Predicted class: dog, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

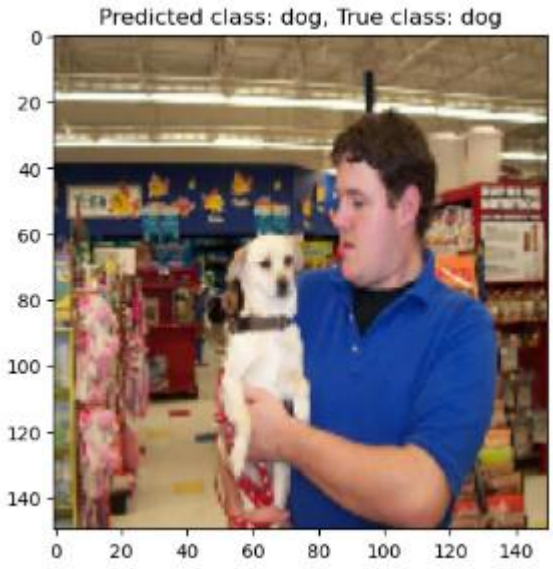

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

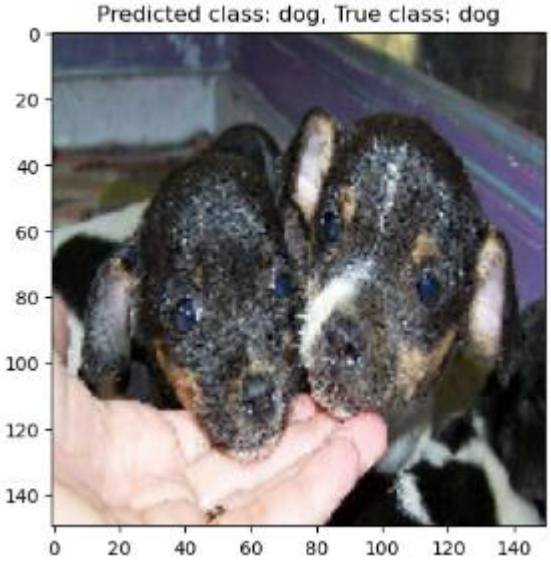

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

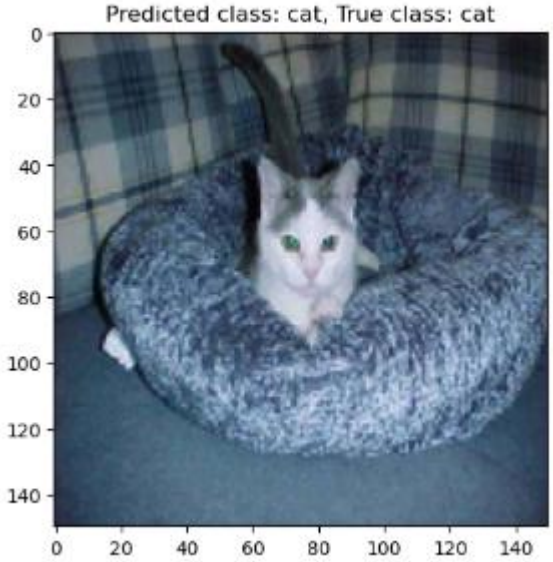

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

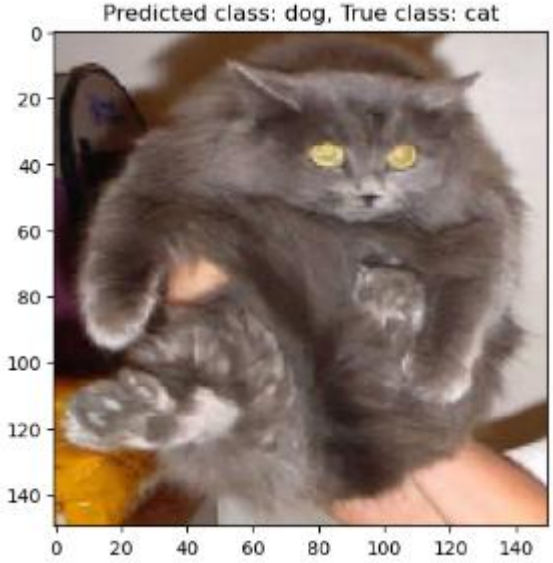

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

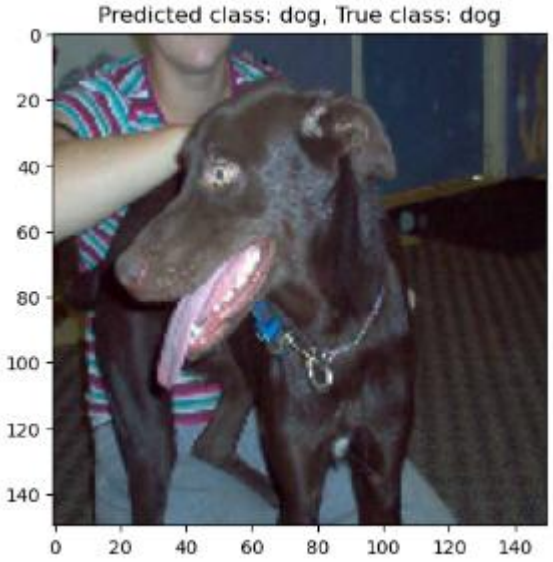

Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

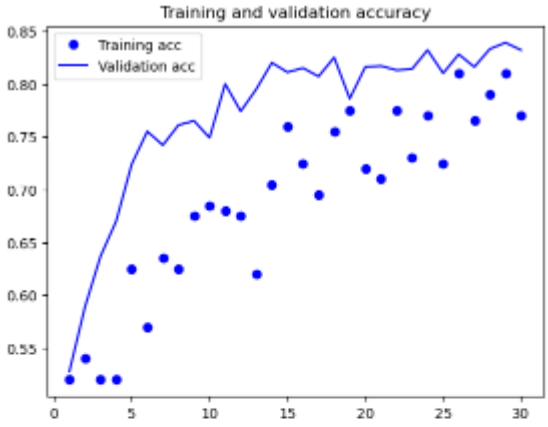

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

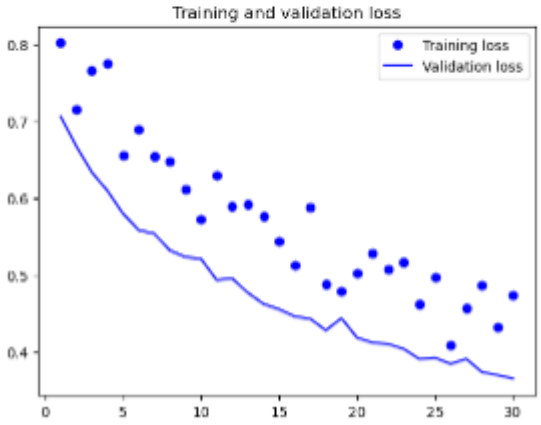

Predicted class: cat, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: dog, True class: cat
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: dog, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140


Predicted class: cat, True class: dog
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140





Deep Marathe - 53004230016

Practical: 9

Aim:- Build an autoencoder for the MNIST dataset to encode and decode digit images, reducing dimensionality while reconstructing the original data.

Theory:-

An **autoencoder** is a type of neural network used for **unsupervised learning** that learns to encode input data into a compressed representation and then reconstruct it back to the original form. It consists of two main parts:

1. **Encoder** – Compresses input data into a lower-dimensional latent space.
2. **Decoder** – Reconstructs the input from this compressed representation.

Key Concepts:

- **Dimensionality Reduction** – Learns efficient representations, similar to PCA but non-linear.
- **Noise Reduction (Denoising Autoencoders)** – Learns to remove noise from data.
- **Anomaly Detection** – Identifies unusual patterns by detecting high reconstruction errors.

Types of Autoencoders:

- **Vanilla Autoencoder** – Basic encoder-decoder structure.
- **Denoising Autoencoder** – Trained to remove noise from input.
- **Sparse Autoencoder** – Uses sparsity constraints for efficient feature learning.
- **Variational Autoencoder (VAE)** – Learns probabilistic latent space for generating new data.

Autoencoders are widely used in **image compression, feature extraction, anomaly detection, and generative modeling**.

Code:-

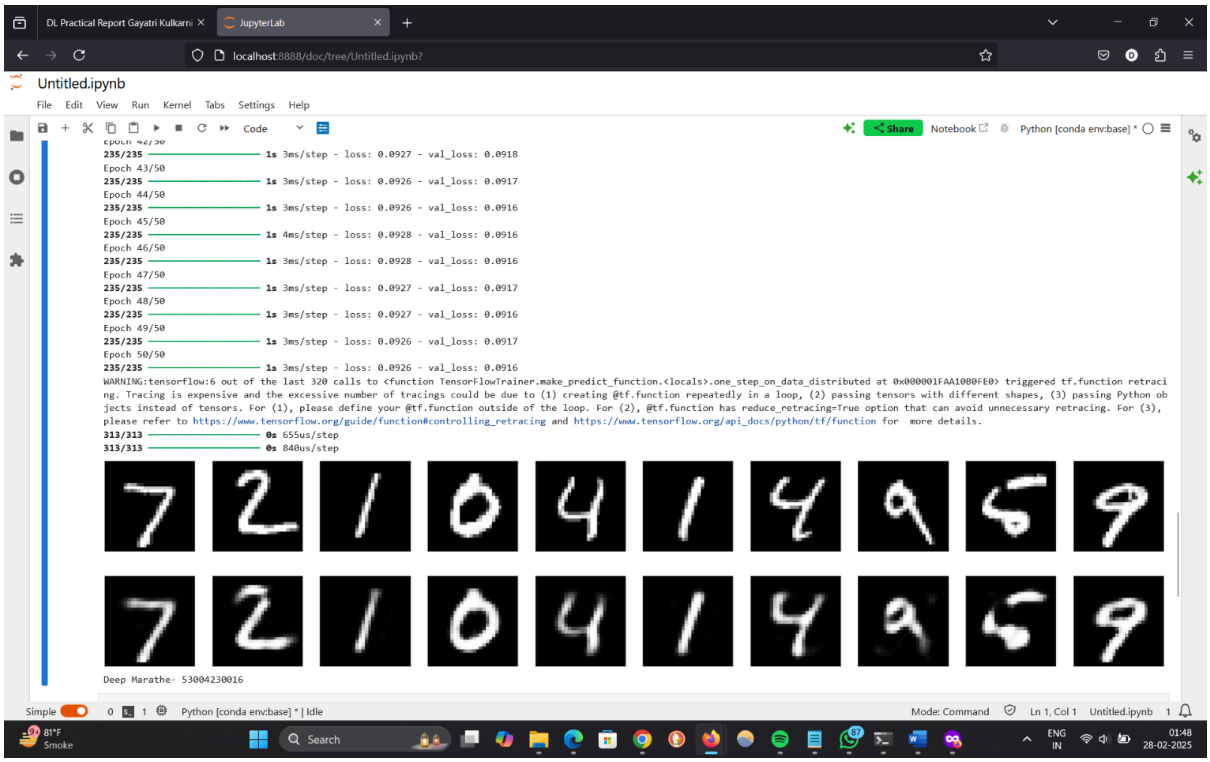
```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist


# Load dataset
```

```
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize the data
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Define the encoding dimension
encoding_dim = 32

# Input layer
input_img = Input(shape=(784,))

# Encoder
encoded = Dense(encoding_dim, activation='relu')(input_img)

# Decoder
decoded = Dense(784, activation='sigmoid')(encoded)

# Autoencoder model (input to output)
autoencoder = Model(input_img, decoded)

# Encoder model (input to encoded representation)
encoder = Model(input_img, encoded)

# Decoder model (encoded input to decoded output)
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

```
# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the model
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

# Encode the test set
encoded_imgs = encoder.predict(x_test)

# Decode the encoded images
decoded_imgs = decoder.predict(encoded_imgs)

# Visualize the original and decoded images
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))

for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display decoded images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
```

```
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()

print("Deep Marathe- 53004230016")
```

Output:-



Practical: 10

Aim:- Implement a Generative Adversarial Network (GAN) to generate realistic handwritten digits based on the MNIST dataset.

Theory:-

A **Generative Adversarial Network (GAN)** is a deep learning framework that consists of two neural networks competing against each other:

1. **Generator (G)** – Creates synthetic data resembling real data.
2. **Discriminator (D)** – Distinguishes between real and fake data.

These networks are trained in an **adversarial process**, where the **generator tries to fool the discriminator**, and the **discriminator improves at detecting fakes**. Over time, the generator produces highly realistic data.

Key Concepts:

- **Adversarial Training** – Both networks improve iteratively through competition.
- **Loss Function** – Uses a minimax game where the generator minimizes the discriminator's accuracy while the discriminator maximizes it.
- **Latent Space** – Random noise is transformed into realistic outputs.

Types of GANs:

- **Vanilla GAN** – Basic GAN architecture.
- **DCGAN (Deep Convolutional GAN)** – Uses CNNs for better image generation.
- **WGAN (Wasserstein GAN)** – Improves training stability with Wasserstein loss.
- **StyleGAN** – Generates high-quality images with style control.

Applications:

- Image Generation (e.g., DeepFake, AI Art)
- Super-Resolution (Enhancing low-res images)
- Data Augmentation
- Drug Discovery

GANs are powerful for generating realistic data but are challenging to train due to issues like **mode collapse** and **instability**.

Code:-

```
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt
```

```
##Load the MNIST dataset

mnist = tf.keras.datasets.mnist

(train_images, train_labels), (_, _) = mnist.load_data()

Normalize the images to [-1, 1]

train_images = (train_images.astype('float32') - 127.5) / 127.5

Reshape the images to (28, 28, 1) and add a channel dimension

train_images = np.expand_dims(train_images, axis=-1)

BUFFER_SIZE = 60000

BATCH_SIZE = 256

train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

##Create the generator model

def make_generator_model():

model = tf.keras.Sequential()

model.add(tf.keras.layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))

model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Reshape((7, 7, 256)))

assert model.output_shape == (None, 7, 7, 256)

model.add(tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
use_bias=False))

assert model.output_shape == (None, 7, 7, 128)

model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.LeakyReLU())



model.add(tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))

assert model.output_shape == (None, 14, 14, 64)
```

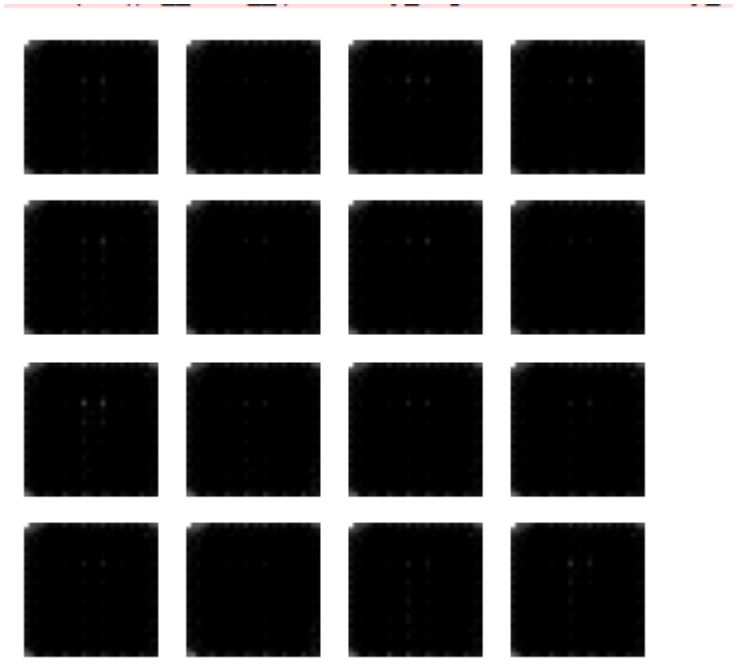
```
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation="tanh"))

assert model.output_shape == (None, 28, 28, 1)

return model


##Create the discriminator model

def make_discriminator_model():

model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))

model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Dropout(0.3))

model.add(tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))

model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Dropout(0.3))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(1))

return model


##Loss functions

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):

real_loss = cross_entropy(tf.ones_like(real_output), real_output)

fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)

total_loss = real_loss + fake_loss
```

```
return total_loss

def generator_loss(fake_output):
return cross_entropy(tf.ones_like(fake_output), fake_output)

##Define the models
generator = make_generator_model()
discriminator = make_discriminator_model()
##Define the optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
##Training loop settings
EPOCHS = 5
noise_dim = 100
num_examples_to_generate = 16
##Training step function
@tf.function
def train_step(images):
noise = tf.random.normal([BATCH_SIZE, noise_dim])
with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
   generated_images = generator(noise, training=True)


   real_output = discriminator(images, training=True)
   fake_output = discriminator(generated_images, training=True)


   gen_loss = generator_loss(fake_output)
   disc_loss = discriminator_loss(real_output, fake_output)


gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
```

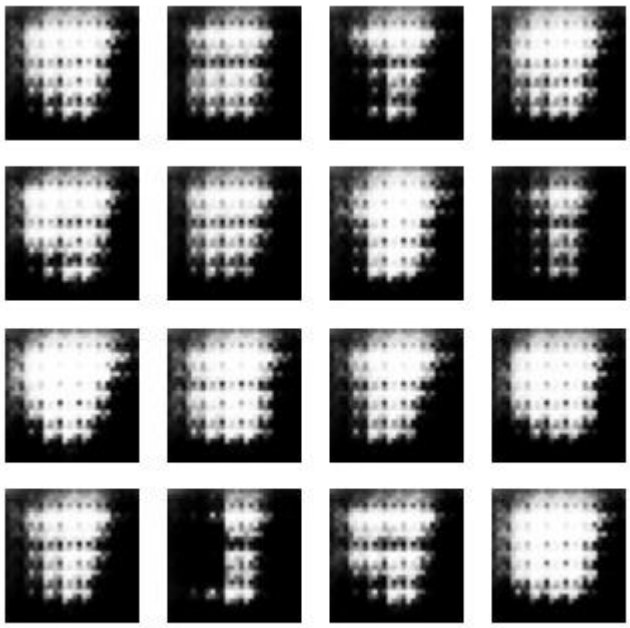
```
gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)


generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))

discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

##Image generation function

def generate_and_save_images(model, epoch, test_input):

predictions = model(test_input, training=False)

predictions = (predictions + 1) / 2.0  # Rescale to [0, 1]


fig = plt.figure(figsize=(4, 4))

for i in range(predictions.shape[0]):

   plt.subplot(4, 4, i+1)

   plt.imshow(predictions[i, :, :, 0], cmap='gray')

   plt.axis('off')


plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))

plt.show()

##Generate a fixed set of noise for evaluating the model during training

fixed_noise = tf.random.normal([num_examples_to_generate, noise_dim])

##Train the model

for epoch in range(EPOCHS):

for image_batch in train_dataset:

   train_step(image_batch)

generate_and_save_images(generator, epoch + 1, fixed_noise)  # Now saving images at every epoch
```

```
print(f'Epoch {epoch + 1} completed')

##Generate final images after training
print("Training completed. Generating final images...")
generate_and_save_images(generator, EPOCHS, fixed_noise)
print('Deep Marathe - 53004230016')
```

Output:-

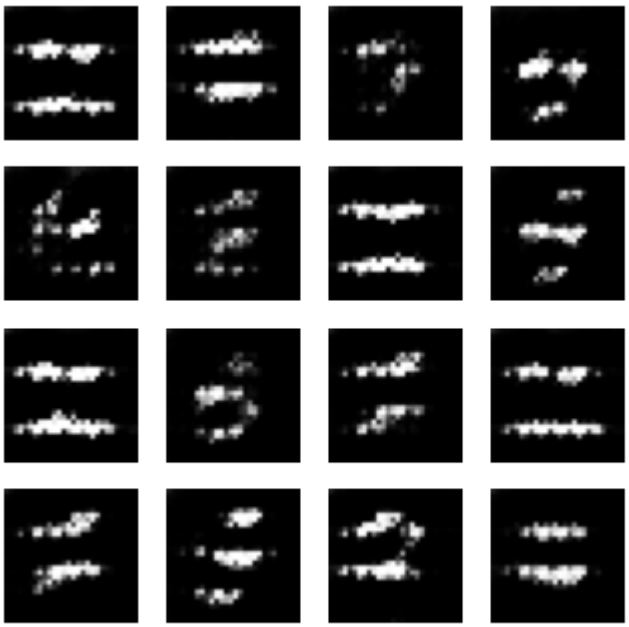


```
Epoch 1 completed
```

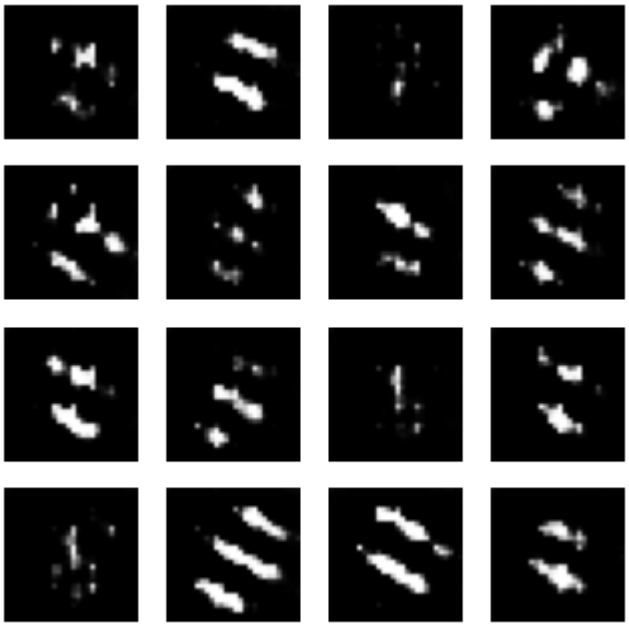


Epoch 2 completed

Epoch 3 completed



Epoch 4 completed



Epoch 5 completed

```
Training completed. Generating final images...
```