**PROGRAM 7**

**AIM-** Implementation of transfer learning using a pre-trained model(VGG-16) for image classification in Python.(Use any Image dataset)

**THEORY-**

Overview of VGG-16:

* VGG-16 is a deep convolutional neural network (CNN) architecture developed by the Visual Geometry Group at Oxford University. It was proposed in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition” by K. Simonyan and A. Zisserman in 2014.
* VGG-16 was designed to improve upon previous architectures by emphasizing simplicity in design while increasing depth. It achieved high performance on the ImageNet dataset, making it one of the most influential models in computer vision.

Key Features of VGG-16:

* Depth: VGG-16 has 16 weight layers, with 13 convolutional layers and 3 fully connected (dense) layers.
* Simplicity: The architecture uses only 3x3 convolutional filters with a stride and padding of 1, which makes it easier to understand and implement.
* Uniform Design: Every convolutional layer has the same 3x3 filter size, and every max-pooling layer has a 2x2 filter with a stride of 2.
* Parameter Consistency: By fixing the convolutional filter size and stacking these layers, the model was able to achieve more parameter efficiency and avoid overfitting compared to previous architectures.

Detailed Architecture of VGG-16:

* The input to VGG-16 is a 224x224 RGB image. It undergoes several layers of convolution, each followed by a ReLU activation.
* Convolutional Layers: VGG-16 has five blocks of convolutional layers. Each block has a specific number of layers:
  + Block 1: Two 3x3 convolutional layers with 64 filters.
  + Block 2: Two 3x3 convolutional layers with 128 filters.
  + Block 3: Three 3x3 convolutional layers with 256 filters.
  + Block 4: Three 3x3 convolutional layers with 512 filters.
  + Block 5: Three 3x3 convolutional layers with 512 filters.
* Pooling Layers: After each convolutional block, there is a max-pooling layer with a 2x2 filter and a stride of 2, which reduces the spatial dimensions.
* Fully Connected Layers: The output of the convolutional layers is flattened and passed through three fully connected layers. The first two dense layers have 4096 neurons each, followed by a final output layer with 1000 neurons (for 1000 ImageNet classes).
* Softmax Layer: A softmax activation function is applied at the output layer to get probabilities for each class.

Advantages of VGG-16:

* Better Feature Representation: The uniform use of small 3x3 filters allows VGG-16 to capture fine-grained features effectively.
* High Performance on ImageNet: VGG-16 achieved excellent results on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), making it a go-to choice for transfer learning.
* Ease of Transfer Learning: The architecture is versatile and well-suited for transfer learning applications due to its simplicity and generalizability.

Limitations of VGG-16:

* Large Memory Requirement: With around 138 million parameters, VGG-16 is memory-intensive, making it difficult to deploy on devices with limited resources.
* High Computation Cost: The depth and number of parameters result in a high computational cost, which can be slow without specialized hardware (like GPUs).
* No Skip Connections: Unlike more modern architectures (e.g., ResNet), VGG-16 doesn’t use skip connections, which can improve gradient flow and training efficiency in deeper networks.

Applications of VGG-16:

* Image Classification: VGG-16 remains popular for classification tasks and is widely used as a baseline for performance.
* Feature Extraction for Transfer Learning: The convolutional layers of VGG-16 can serve as effective feature extractors, enabling it to generalize well to various new datasets.
* Object Detection and Segmentation: The feature maps generated by VGG-16’s convolutional layers are utilized in object detection architectures like Faster R-CNN and semantic segmentation models like Fully Convolutional Networks (FCNs).

CODE AND OUTPUT-

import pandas as pd

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

import warnings

import os

os.environ['TF\_ENABLE\_ONEDNN\_OPTS'] = '0'

warnings.filterwarnings('ignore')

tf.test.gpu\_device\_name()

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_dir=r'output/train'

test\_dir=r'output/val'

train\_data\_genrator=ImageDataGenerator(rescale=1./255)

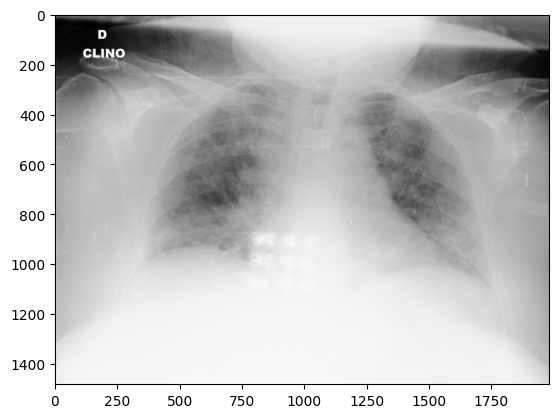
train\_data=train\_data\_genrator.flow\_from\_directory(train\_dir,target\_size=(224,224),batch\_size=32,class\_mode='categorical')

test\_data\_generator=ImageDataGenerator(rescale=1./255)

test\_data=test\_data\_generator.flow\_from\_directory(test\_dir,target\_size=(224,224),batch\_size=32,class\_mode='categorical')

num\_classes=train\_data.num\_classes

plt.imshow(Image.open(r"archive (6)/Data/train/COVID19/COVID19(1).jpg"))



from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.layers import \*

import tensorflow

from tensorflow.keras import Model

vgg\_model\_load = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

vgg\_model=vgg\_model\_load.output

x = tensorflow.keras.layers.GlobalAveragePooling2D()(vgg\_model)

x = tensorflow.keras.layers.Dense(1024, activation='relu')(x)

predictions = tensorflow.keras.layers.Dense(num\_classes, activation='softmax')(x)

vgg\_model=Model(inputs=vgg\_model\_load.input,outputs=predictions)

early\_stopping=EarlyStopping(patience=3,monitor="val\_loss")

vgg\_model.compile(optimizer="adam",loss="categorical\_crossentropy",metrics=['accuracy'])

history=vgg\_model.fit(x=train\_data,validation\_data=test\_data,epochs=15,callbacks=[early\_stopping],verbose=1) 

def plot\_curves(history):

    plt.plot(history.history['accuracy'])

    plt.plot(history.history['val\_accuracy'])

    plt.title('model accuracy')

    plt.ylabel('accuracy')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

    # "Loss"

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

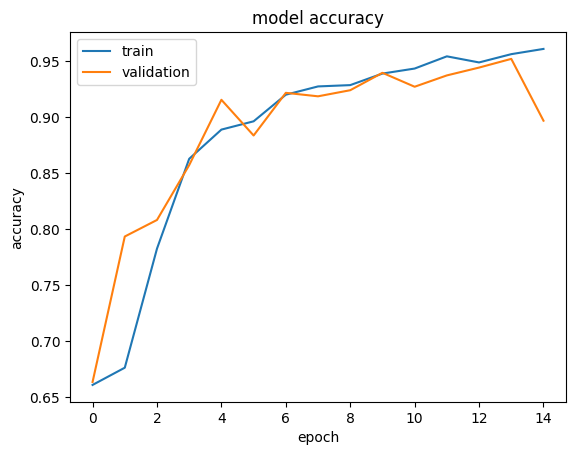
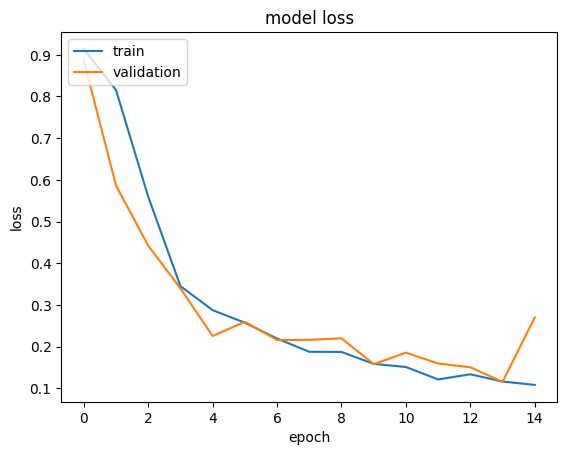
    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

plot\_curves(history)



vgg\_model.save("vgg16.keras")

**VIVA VOICE-**

1. What is Transfer Learning?

Transfer Learning is the process of leveraging a pre-trained model on a new, related task. It allows the model to apply learned patterns from a large dataset to a smaller one, thus reducing the time and computational resources needed.

1. Why use VGG-16?

VGG-16 is a deep neural network trained on the ImageNet dataset with millions of images across thousands of classes, making it an excellent candidate for transfer learning, especially in image classification tasks. Its architecture is known for its simplicity and effectiveness.

1. How does freezing layers help?

Freezing layers helps retain the learned features from the pre-trained model without updating the weights. This is particularly useful when the new dataset is small, as it prevents overfitting and ensures that the model’s high-level features, such as edges and shapes, remain intact.

1. What changes did you make to VGG-16 for CIFAR-10 classification?

Since VGG-16 was initially trained on 1000 classes for ImageNet, we removed the fully connected layers at the top and added custom dense layers suited for the 10-class CIFAR-10 dataset. This new top section of the model is trainable and will adapt to the new dataset.

1. What is the role of Flatten and Dense layers?

The Flatten layer converts the 3D output of the convolutional layers into a 1D vector, which is then fed to the fully connected (Dense) layers. Dense layers at the top add non-linearity, allowing the network to make class-specific predictions.

**PROGRAM 9**

**AIM**- Implementation of RNN Model for Stock Price Prediction in Python.(Stock Price Dataset)

**THEORY-**

Overview of Recurrent Neural Networks (RNNs):

* Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data where previous inputs inform later ones. Unlike feedforward networks, RNNs have connections that form cycles, enabling them to maintain a "memory" of previous information.
* RNNs are widely used in applications involving sequential data, such as natural language processing (NLP), time-series prediction, speech recognition, and video analysis.

Key Characteristics of RNNs:

* Sequential Data Processing: RNNs process data sequentially, which allows them to remember and utilize information from previous time steps. This makes them particularly suitable for tasks where the sequence order matters.
* Hidden State (Memory): At each time step, RNNs maintain a hidden state that stores information from previous steps. This hidden state acts as the network's memory, capturing context that can influence future predictions.
* Weight Sharing: RNNs use the same set of weights (parameters) across all time steps, making them efficient and capable of handling variable-length sequences.

RNN Architecture and Forward Propagation:

* Input and Hidden State: At each time step ttt, the RNN receives an input vector xtx\_txt​ and an updated hidden state ht−1h\_{t-1}ht−1​ from the previous time step.
* Updating the Hidden State: The hidden state hth\_tht​ at time ttt is computed as: ht=f(Wxh⋅xt+Whh⋅ht−1+bh)h\_t = f(W\_{xh} \cdot x\_t + W\_{hh} \cdot h\_{t-1} + b\_h)ht​=f(Wxh​⋅xt​+Whh​⋅ht−1​+bh​) where:
  + WxhW\_{xh}Wxh​ is the weight matrix for the input.
  + WhhW\_{hh}Whh​ is the weight matrix for the hidden state.
  + bhb\_hbh​ is the bias term.
  + fff is typically a non-linear activation function (e.g., tanh or ReLU).
* Output at Each Time Step: The output yty\_tyt​ at each time step can be computed as: yt=g(Why⋅ht+by)y\_t = g(W\_{hy} \cdot h\_t + b\_y)yt​=g(Why​⋅ht​+by​) where:
  + WhyW\_{hy}Why​ is the output weight matrix.
  + byb\_yby​ is the bias for the output layer.
  + ggg is often the softmax function for classification tasks.

Types of RNNs:

* One-to-One (Vanilla RNN): A standard RNN used for traditional feedforward tasks like image classification.
* One-to-Many: Used in applications like image captioning, where a single image (one input) generates a sequence of words (many outputs).
* Many-to-One: Common in sentiment analysis, where a sequence of words (many inputs) results in a single sentiment score (one output).
* Many-to-Many: Used in tasks like machine translation and video classification, where a sequence of inputs maps to a sequence of outputs.

Challenges with Standard RNNs:

* Vanishing and Exploding Gradients: During training, the gradients can either become very small (vanish) or very large (explode). This happens due to the recurrent nature of RNNs, which multiply the gradients at each time step, causing instability.
* Limited Long-Term Memory: Standard RNNs struggle to retain information over long sequences, as their hidden states get overwhelmed by recent inputs, which leads to poor performance on tasks requiring long-range dependencies.

CODE AND OUTPUT-

import pandas as pd

import numpy as np

import tensorflow as tf

import tensorflow.keras as keras

import matplotlib.pyplot as plt

df=pd.read\_csv("NSE-Tata-Global-Beverages-Limited.csv")

df.head()



df['Date'] = pd.to\_datetime(df['Date'])

df = df.sort\_values('Date')

df=df.iloc[:,[0,1,2,3,4,6,7,5]]

x=df.iloc[:,:-1].values

y=df.iloc[:,-1].values

from sklearn.preprocessing import StandardScaler,LabelEncoder

encoder=LabelEncoder()

x[:,0]=encoder.fit\_transform(x[:,0])

scaler\_1=StandardScaler()

y=scaler\_1.fit\_transform(y.reshape(-1, 1))

from sklearn.model\_selection import train\_test\_split

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,random\_state=0,test\_size=0.2)

x\_train.shape,x\_test.shape,y\_train.shape,y\_test.shape

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))

from tensorflow.keras.layers import \*

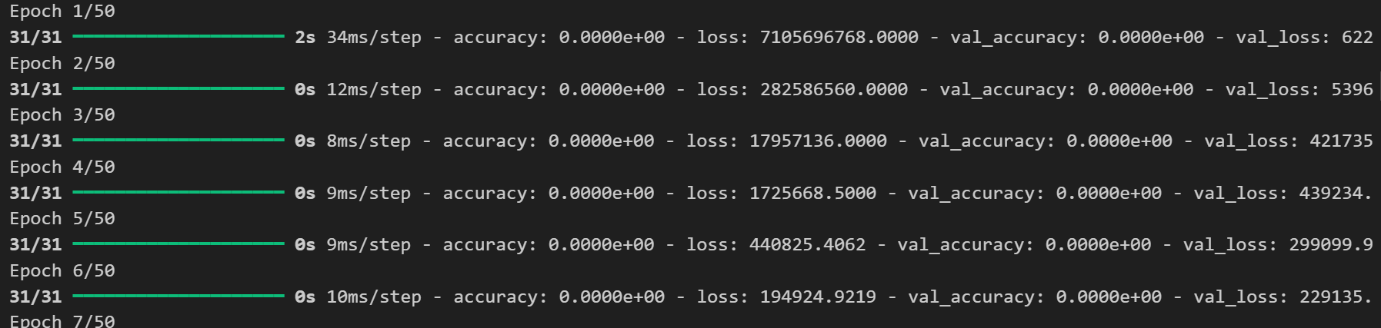
model=tf.keras.Sequential()

model.add(SimpleRNN(units=50,activation='relu'))

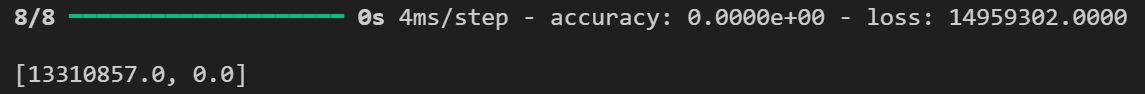
model.add(Dense(1))

model.compile(optimizer='adam', loss='mean\_squared\_error',metrics=['accuracy'])

history=model.fit(x=tf.convert\_to\_tensor(x\_train,np.float32),y=tf.convert\_to\_tensor(y\_train,np.float32),validation\_data=[tf.convert\_to\_tensor(x\_test,np.float32),tf.convert\_to\_tensor(y\_test,np.float32)],batch\_size=32,epochs=50)



model.evaluate(tf.convert\_to\_tensor(x\_test,np.float32),tf.convert\_to\_tensor(y\_test,np.float32))



model.save('rnn.keras')

def plot\_curves(history):

    plt.plot(history.history['accuracy'])

    plt.plot(history.history['val\_accuracy'])

    plt.title('model accuracy')

    plt.ylabel('accuracy')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

    # "Loss"

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

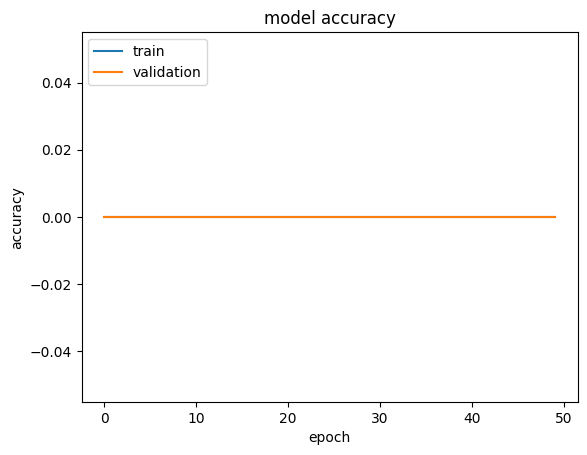
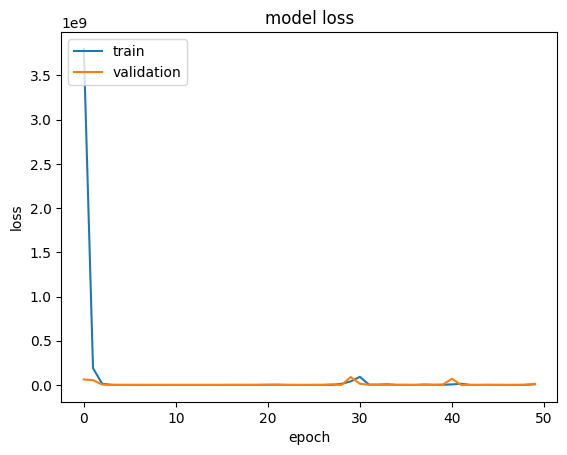
    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

plot\_curves(history)



**VIVA VOICE-**

1. What is a Recurrent Neural Network (RNN)?

Answer: An RNN is a type of neural network designed to handle sequential data, where the output depends on previous inputs. It has connections that form cycles, allowing it to retain memory of previous time steps, which is essential for tasks where context or order is important.

2. How does an RNN differ from a traditional feedforward neural network?

Answer: In an RNN, connections between nodes form cycles, allowing information to persist over time. Unlike feedforward networks, where inputs and outputs are processed independently, RNNs have a hidden state that carries information from previous steps, making them suitable for sequential data.

3. What is the vanishing gradient problem in RNNs?

Answer: The vanishing gradient problem occurs during training when gradients become very small as they propagate back through time, leading to poor learning. This makes it difficult for RNNs to capture long-term dependencies in sequences, as the influence of earlier time steps fades over time.

4. What are LSTM and GRU, and how do they address the vanishing gradient problem?

Answer: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) are types of RNNs that introduce gating mechanisms to control information flow. They selectively retain or forget information, allowing them to capture long-term dependencies more effectively and reduce the impact of vanishing gradients.

5. Explain the purpose of gates in LSTM networks.

Answer: Gates in LSTM networks control the flow of information. The input gate decides what new information to store, the forget gate determines what to discard from the cell state, and the output gate controls the output at each time step. These gates help manage and preserve relevant information over longer sequences.

**PROGRAM 10**

**AIM-** Implementation of Autoencoders on Image Dataset.(Use MNIST Dataset)

**THEORY-**

Overview of Autoencoders:

* An Autoencoder is a type of artificial neural network designed to learn efficient representations of data, typically for dimensionality reduction or feature extraction. It learns to map input data to a compressed representation (encoding) and then reconstruct the input from this compressed version (decoding).
* Autoencoders are used in unsupervised learning, as they do not require labeled data for training and can identify important structures in the data.

Architecture of Autoencoders:

* Encoder: The encoder compresses the input data into a lower-dimensional representation (latent space) by reducing the number of neurons in each successive layer. This part of the network captures essential information from the input while discarding redundant details.
* Latent Space (Code): The encoded representation, or "code," is a compact form of the input data, capturing its most important features.
* Decoder: The decoder reconstructs the original input from the compressed latent space representation. It has a mirror architecture to the encoder and aims to minimize the difference between the original input and the reconstruction.

Loss Function:

* Autoencoders are trained to minimize reconstruction loss, which is the difference between the original input and its reconstructed output. Common loss functions include Mean Squared Error (MSE) for continuous data or Binary Cross-Entropy for binary data.

Types of Autoencoders:

* Vanilla Autoencoder: A basic structure with a single encoder and decoder, trained to reconstruct the input.
* Denoising Autoencoder: Trained to reconstruct the original input from a noisy version of it, useful for noise reduction in data.
* Sparse Autoencoder: Encourages sparsity in the latent representation by adding a penalty term, ensuring that only a few neurons are active at any time, making it suitable for feature extraction.
* Variational Autoencoder (VAE): A probabilistic autoencoder that learns a distribution over the latent space, allowing for the generation of new data samples. It has applications in generative modeling.

CODE AND OUTPUT-

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist

import tensorflow as tf

from tensorflow.keras.layers import \*

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train.shape,x\_test.shape



# Build the Autoencoder model

input\_img = Input(shape=(28,28))

x=tf.keras.layers.Rescaling(1./255)(input\_img)

x=tf.keras.layers.Flatten()(x)# 28x28 = 784

#! Encoder

encoded = Dense(64, activation='relu')(x)

#! Decoder

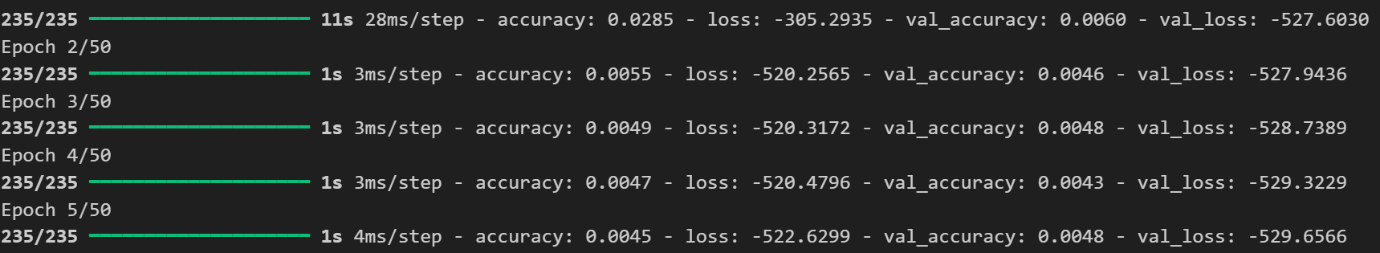
decoded = Dense(784, activation='sigmoid')(encoded)

decoded = Reshape((28,28))(decoded)

autoencoder = Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy',metrics=['accuracy'])

autoencoder.fit(x\_train, x\_train, epochs=50, batch\_size=256, shuffle=True, validation\_data=(x\_test, x\_test))



# ! Visualize the results

n = 10

plt.figure(figsize=(20, 4))

for i in range(n):

    #! Original images

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i].reshape(28, 28), cmap='gray')

    plt.title("Original")

    plt.axis('off')

    #! Reconstructed images

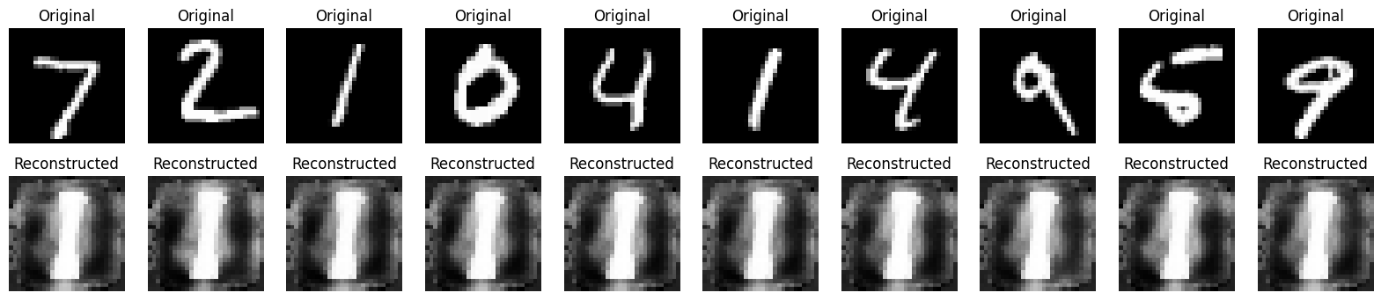
    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(decoded\_imgs[i].reshape(28, 28), cmap='gray')

    plt.title("Reconstructed")

    plt.axis('off')

plt.show()



autoencoder.save('autoencoder.keras')

**VIVA VOICE-**

What is an autoencoder?

* Answer: An autoencoder is a type of neural network that learns to compress data into a lower-dimensional representation (encoding) and then reconstructs the data from this compressed form (decoding), minimizing reconstruction error.

Explain the role of the encoder and decoder in an autoencoder.

* Answer: The encoder maps the input to a lower-dimensional representation, capturing essential information. The decoder then reconstructs the input from this compressed representation, aiming to recreate the original data as accurately as possible.

What is the purpose of the latent space in an autoencoder?

* Answer: The latent space, or "code," is the compressed representation of the input data. It contains the most important features, making it efficient for tasks like dimensionality reduction and feature extraction.

How do denoising autoencoders work, and what are they used for?

* Answer: Denoising autoencoders are trained to reconstruct clean data from noisy inputs. This helps the model learn robust features and can be used for noise reduction in data, especially in images.

What is a variational autoencoder (VAE)?

* Answer: A VAE is a probabilistic autoencoder that learns a distribution over the latent space. By sampling from this distribution, it can generate new data, making it useful for generative tasks.

**PROGRAM 12**

**AIM-** Implement NLP to analyse restaurant reviews in Python.

**THEORY-**

Overview of NLP:

* Natural Language Processing (NLP) is a field at the intersection of computer science, artificial intelligence, and linguistics focused on enabling computers to understand, interpret, and generate human language. NLP techniques allow machines to process and analyze large amounts of natural language data, making it useful in applications like machine translation, sentiment analysis, chatbots, and information retrieval.

Key Components of NLP:

* Tokenization: Splitting text into smaller units, such as words, sentences, or subwords. Tokenization is the first step in processing text data and is fundamental for further text processing tasks.
* Stopword Removal: Removing common words that do not carry significant meaning (like "and," "the," "is") to reduce noise in the data.
* Stemming and Lemmatization: Reducing words to their root forms. Stemming cuts off suffixes, while lemmatization uses linguistic rules to reduce words to their base forms (e.g., "running" to "run").
* Part of Speech (POS) Tagging: Labeling each word with its part of speech, such as noun, verb, adjective, etc., which helps in understanding sentence structure.
* Named Entity Recognition (NER): Identifying and categorizing named entities (e.g., people, organizations, locations) within the text.

Techniques in NLP:

* Bag of Words (BoW): Represents text as a collection of words without considering their order. It creates a vocabulary of unique words in the dataset, and each document is represented by a vector of word counts.
* TF-IDF (Term Frequency-Inverse Document Frequency): A statistical measure that evaluates the importance of a word in a document relative to the entire corpus. It helps highlight meaningful words by giving less importance to common words across documents.
* Word Embeddings: Dense vector representations of words that capture semantic relationships. Techniques like Word2Vec, GloVe, and FastText create embeddings that reflect the context and meaning of words.
* Language Models: Models that predict the probability of a sequence of words. Early models like n-grams were statistical, but deep learning-based models like RNNs, LSTMs, and transformers (e.g., BERT, GPT) have since greatly advanced NLP.

Deep Learning in NLP:

* Recurrent Neural Networks (RNNs): Used to process sequential data, but they struggle with long-term dependencies due to vanishing gradients.
* LSTM and GRU: Variants of RNNs that use gating mechanisms to retain long-term dependencies in sequences.
* Transformers: A neural network architecture that uses self-attention to process entire sequences simultaneously. Transformers, like BERT and GPT, are foundational in NLP for their ability to handle long dependencies and capture complex context.

CODE AND OUTPUT-

import pandas as pd

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

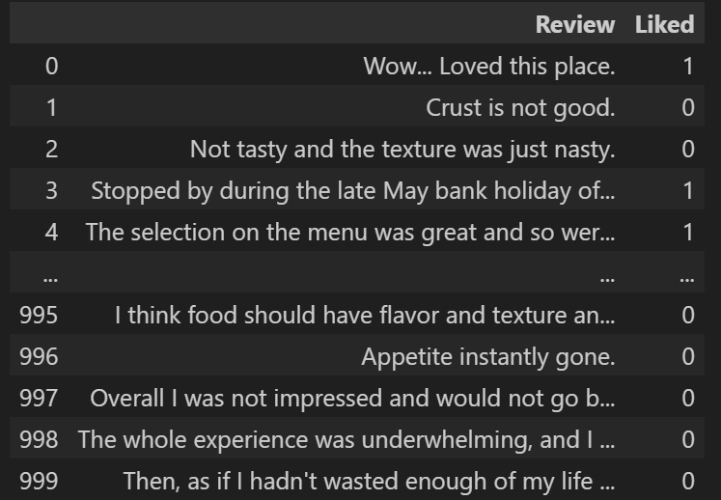
from tensorflow.keras import \*

from tensorflow.keras.layers import \*

tf.test.is\_gpu\_available()

df=pd.read\_csv("Restaurant\_Reviews.tsv",sep="\t")

df



reviews = df['Review'].values

labels = df['Liked'].values

import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

# Parameters

vocab\_size = 5000

max\_length = 100

embedding\_dim = 16

# Tokenize and pad sequences

tokenizer = Tokenizer(num\_words=vocab\_size, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(reviews)

sequences = tokenizer.texts\_to\_sequences(reviews)

padded\_sequences = pad\_sequences(sequences, maxlen=max\_length, padding='post', truncating='post')

model = tf.keras.Sequential([

    tf.keras.layers.Embedding(vocab\_size, embedding\_dim, input\_length=max\_length),

    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),

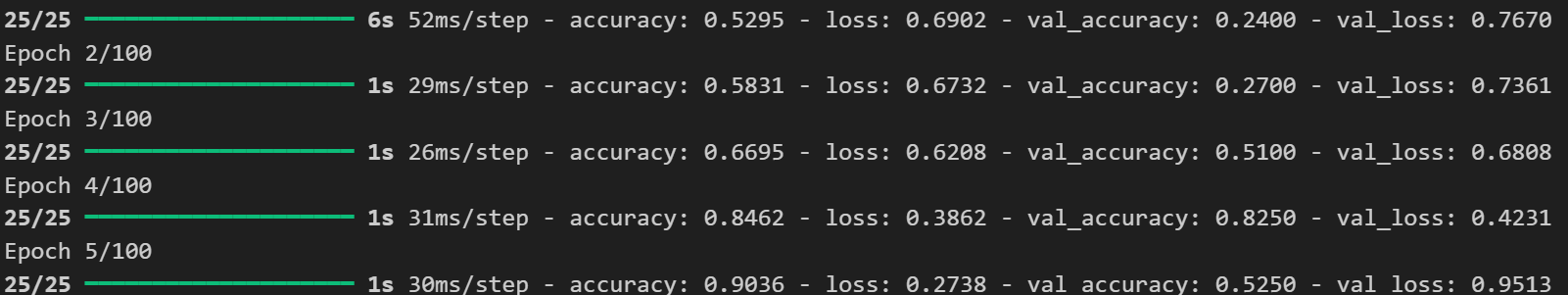
    tf.keras.layers.Dense(64, activation='relu'),

    tf.keras.layers.Dense(1, activation='sigmoid')

])

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

history = model.fit(padded\_sequences, labels, epochs=100, validation\_split=0.2)



def plot\_curves(history):

    plt.plot(history.history['accuracy'])

    plt.plot(history.history['val\_accuracy'])

    plt.title('model accuracy')

    plt.ylabel('accuracy')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

    # "Loss"

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

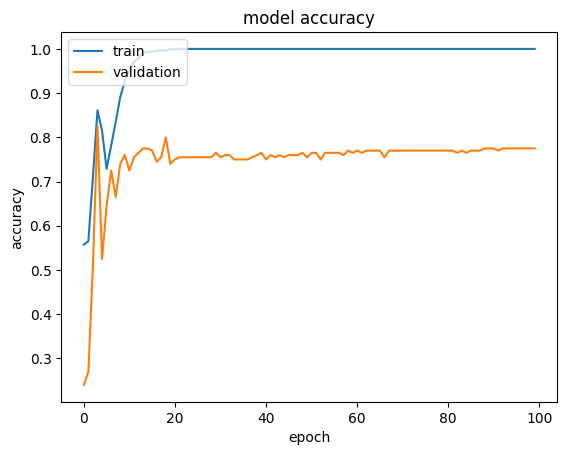
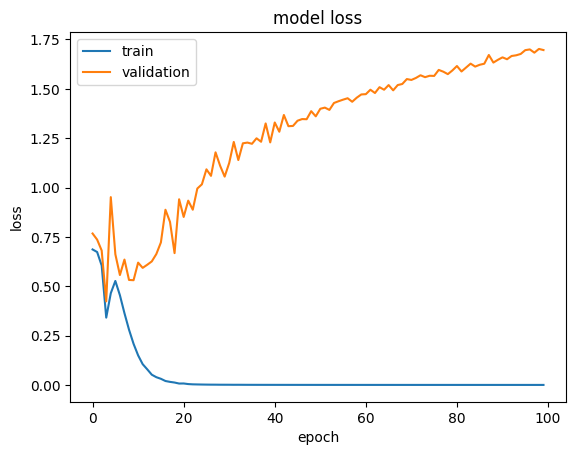
    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

plot\_curves(history)

model.save("nlp.keras")

# Evaluate the model on the validation data

loss, accuracy = model.evaluate(padded\_sequences, labels, verbose=2)

print(f"Loss: {loss}")

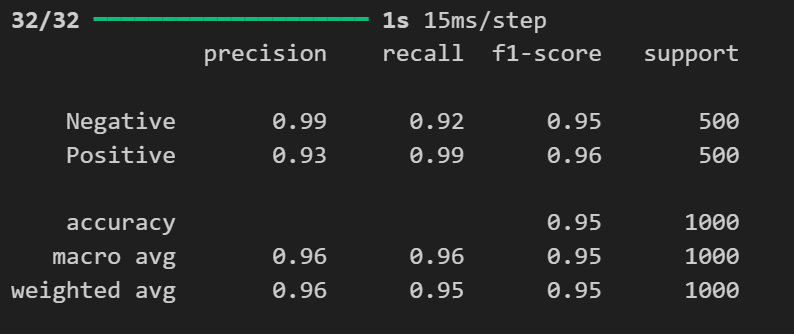
print(f"Accuracy: {accuracy}")



from sklearn.metrics import classification\_report

predictions = (model.predict(padded\_sequences) > 0.5).astype("int32")

print(classification\_report(labels, predictions, target\_names=["Negative", "Positive"]))



**VIVA VOICE-**

What is Natural Language Processing (NLP)?

* Answer: NLP is a field of AI focused on enabling machines to understand, interpret, and generate human language, allowing for applications like translation, sentiment analysis, and text summarization.

What is tokenization in NLP, and why is it important?

* Answer: Tokenization is the process of breaking down text into smaller units, such as words or sentences. It’s essential because it enables the machine to understand the structure of language and facilitates further processing.

Explain the difference between stemming and lemmatization.

* Answer: Stemming removes suffixes to bring words to their root form, often creating non-words (e.g., "running" to "run"). Lemmatization uses linguistic rules to reduce words to their base form (e.g., "running" to "run"), preserving correct word forms.

What are word embeddings, and how do they differ from Bag of Words?

* Answer: Word embeddings are dense vector representations of words that capture semantic meanings. Unlike Bag of Words, which counts word frequency, embeddings place similar words close in vector space, capturing contextual meaning.

What is TF-IDF, and how does it work?

* Answer: TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure that evaluates the importance of a word in a document relative to a corpus. It increases with the frequency in a specific document but decreases if the word is common across the entire corpus.

**PROGRAM 8**

**AIM-** Comparison of various pre-trained models(ResNet, DenseNet, VGGNet) for diagnosing Brain Tumor (Use any Brain Tumor dataset)

**THEORY-**

Overview of Image Classification with CNN Models:

* Image classification with deep learning models, especially Convolutional Neural Networks (CNNs), has advanced significantly, thanks to pre-trained models like ResNet, DenseNet, and VGGNet. These models are designed to extract meaningful features from images and have shown remarkable performance across diverse applications, from object detection to image classification.

Pre-Trained Models in Image Classification:

a. VGGNet:

* Architecture: VGGNet uses a deep CNN structure with multiple convolutional layers and pooling layers. It’s characterized by simplicity, as it employs only 3x3 convolution filters and small strides.
* Strengths: VGGNet is effective at extracting spatial features due to its deep architecture. Its uniform layer design allows it to perform well on a variety of image classification tasks, especially those with large datasets.
* Limitations: VGGNet is computationally intensive with a high number of parameters, making it slower and resource-demanding. Additionally, its fully connected layers can lead to overfitting on smaller datasets.

b. ResNet (Residual Network):

* Architecture: ResNet is notable for its residual connections or skip connections that help bypass layers, enabling the network to learn identity mappings. This solves the vanishing gradient problem, making ResNet efficient for training deep models.
* Strengths: ResNet supports extremely deep architectures, enabling the capture of complex patterns. Its skip connections help maintain accuracy even in deep networks, making it versatile for complex classification tasks.
* Limitations: While ResNet mitigates degradation in deep networks, it requires careful tuning, as very deep networks can lead to overfitting on small datasets.

c. DenseNet:

* Architecture: DenseNet introduces dense connections where each layer is connected to every other layer, promoting feature reuse across the network. This reduces the number of parameters and makes feature extraction more efficient.
* Strengths: DenseNet is highly parameter-efficient and reduces redundancy by reusing features, which improves model accuracy and helps prevent overfitting. DenseNet also has improved gradient flow, making it effective for both shallow and deep networks.
* Limitations: DenseNet’s dense connections require more memory during training, and this model architecture can be slower when the input image size is large.

Comparison Summary:

* VGGNet: Good for straightforward feature extraction but requires high computational power.
* ResNet: Handles deep architectures efficiently, best suited for tasks needing complex feature extraction.
* DenseNet: Parameter-efficient and highly accurate but requires more memory, ideal for cases needing minimal feature redundancy.

Evaluation Metrics for Comparison:

* To evaluate these models, common metrics include accuracy, precision, recall, F1-score, and AUC (Area Under the Curve). Training time, parameter count, and computational efficiency are also important when choosing a model for a given application.

CODE AND OUTPUT-

import pandas as pd

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

import warnings

import os

os.environ['TF\_ENABLE\_ONEDNN\_OPTS'] = '0'

warnings.filterwarnings('ignore')

tf.test.gpu\_device\_name()

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.preprocessing import image\_dataset\_from\_directory

train\_dir=r'archive (6)/Data/train'

test\_dir=r'archive (6)/Data/test'

train\_data\_genrator=ImageDataGenerator()

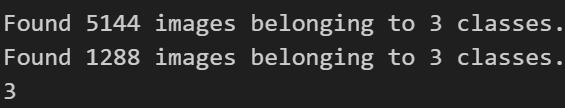
train\_data=train\_data\_genrator.flow\_from\_directory(train\_dir,target\_size=(224,224),batch\_size=32,class\_mode='categorical')

test\_data\_generator=ImageDataGenerator()

test\_data=test\_data\_generator.flow\_from\_directory(test\_dir,target\_size=(224,224),batch\_size=32,class\_mode='categorical')

num\_classes=train\_data.num\_classes

print(num\_classes)



from tensorflow.keras.callbacks import EarlyStopping

early\_stopping=EarlyStopping(patience=3,monitor="val\_loss")

def plot\_curves(history):

    plt.plot(history.history['accuracy'])

    plt.plot(history.history['val\_accuracy'])

    plt.title('model accuracy')

    plt.ylabel('accuracy')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

    # "Loss"

    plt.plot(history.history['loss'])

    plt.plot(history.history['val\_loss'])

    plt.title('model loss')

    plt.ylabel('loss')

    plt.xlabel('epoch')

    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()

## RESNET 50

from tensorflow.keras.applications import ResNet50V2

from tensorflow.keras.layers import \*

import tensorflow

resnet\_50=ResNet50V2(input\_shape=(224,224,3),classes=num\_classes,include\_top=False,weights='imagenet')

input=tensorflow.keras.layers.Input(name="Input\_Layer",shape=(224,224,3))

# x=tf.keras.layers.Rescaling(1./255)(input)

x=resnet\_50(input)

# x=tensorflow.keras.layers.GlobalAveragePooling2D()(x)

x=tensorflow.keras.layers.Flatten()(x)

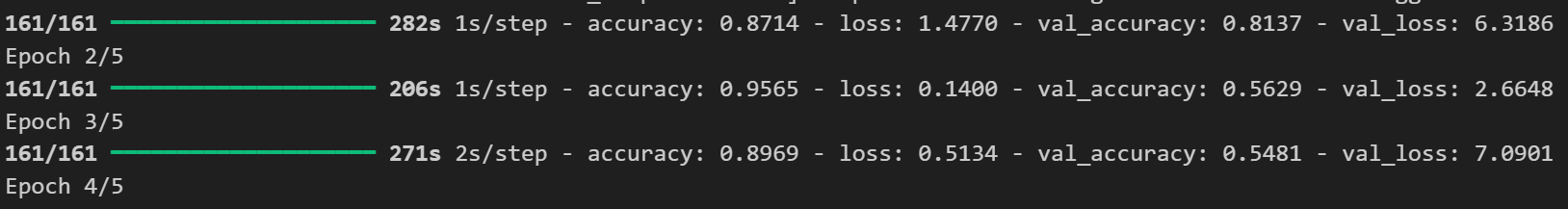
x=tensorflow.keras.layers.Dense(512,activation='relu')(x)

output=tensorflow.keras.layers.Dense(num\_classes,activation='softmax',name='Output\_Layer')(x)

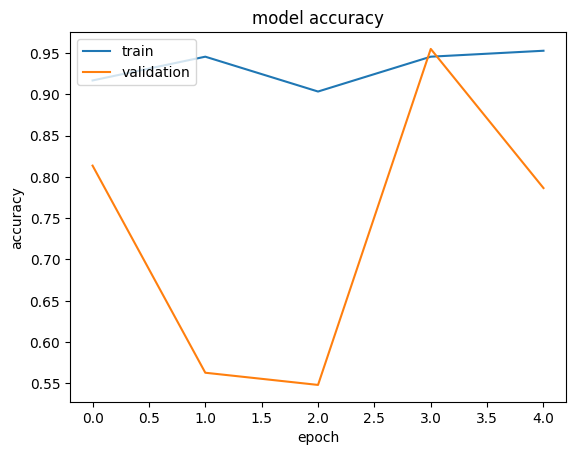
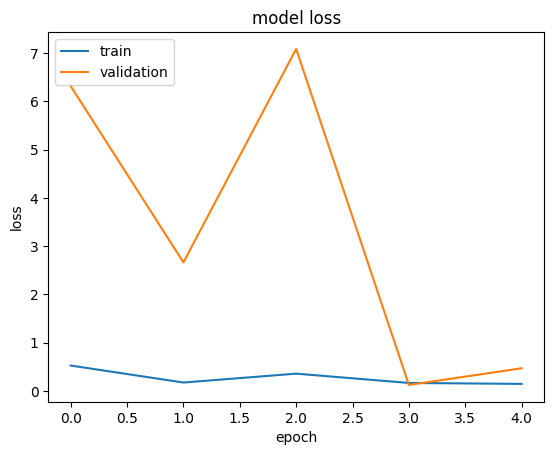
model\_resnet\_50=tf.keras.Model(inputs=input,outputs=output)

model\_resnet\_50.compile(loss="categorical\_crossentropy",metrics=["accuracy"],optimizer="adam")

history\_resnet\_50=model\_resnet\_50.fit(train\_data,validation\_data=test\_data,epochs=5,callbacks=[early\_stopping])



plot\_curves(history\_resnet\_50)



model\_resnet\_50.save("resnet50.keras")

## DenseNet

from tensorflow.keras.applications import DenseNet201

densenet\_201=DenseNet201(input\_shape=(224,224,3),classes=num\_classes,include\_top=False,weights='imagenet')

input=Input(name="Input\_Layer",shape=(224,224,3))

x=tf.keras.layers.Rescaling(1./255)(input)

x=densenet\_201(x)

x=GlobalAveragePooling2D()(x)

x=Dense(512, activation='relu')(x)

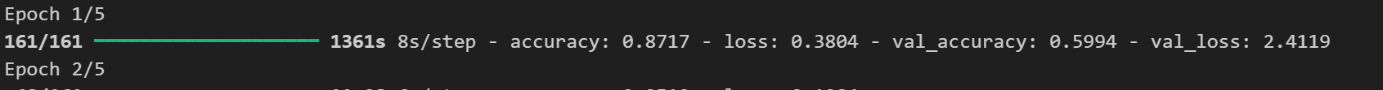
x=Dropout(0.5)(x)

output=Dense(num\_classes,activation='softmax',name='Output\_Layer')(x)

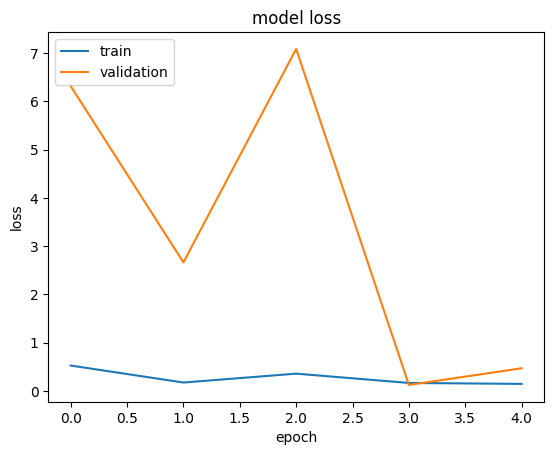
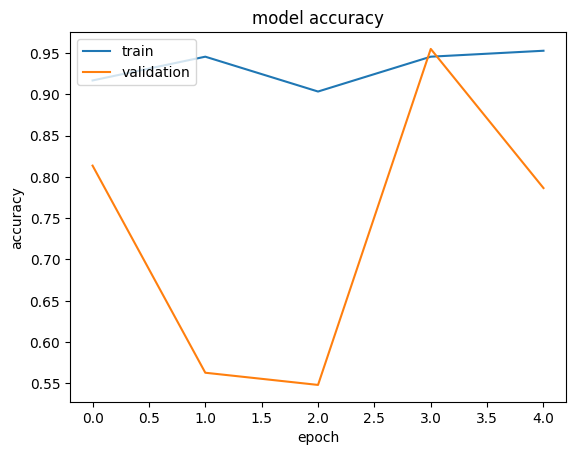
model\_densenet\_201=tf.keras.Model(input,output)

model\_densenet\_201.compile(loss="categorical\_crossentropy",metrics=["accuracy"],optimizer="adam")

history\_densenet\_201=model\_densenet\_201.fit(train\_data,validation\_data=test\_data,epochs=5,callbacks=[early\_stopping])



plot\_curves(history\_densenet\_201)



model\_densenet\_201.save("Densenet.keras")

## VGG NET

from tensorflow.keras import Model

vgg\_model\_load = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

vgg\_model=vgg\_model\_load.output

x = tensorflow.keras.layers.GlobalAveragePooling2D()(vgg\_model)

x = tensorflow.keras.layers.Dense(1024, activation='relu')(x)

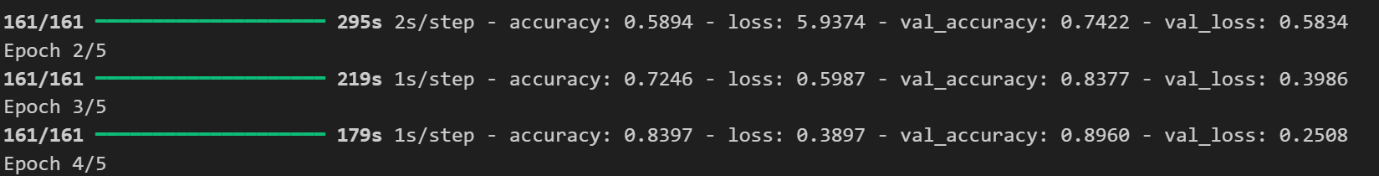
predictions = tensorflow.keras.layers.Dense(num\_classes, activation='softmax')(x)

vgg\_model=Model(inputs=vgg\_model\_load.input,outputs=predictions)

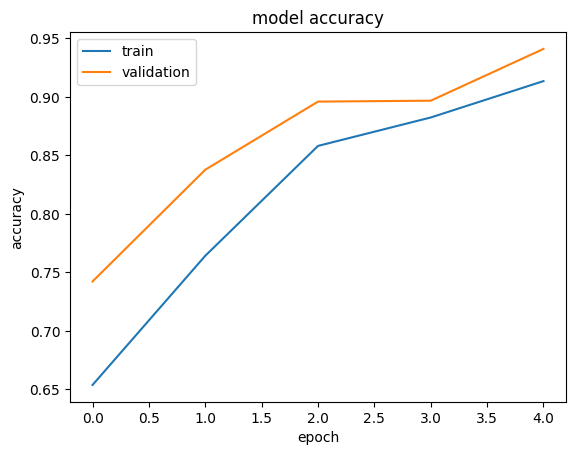
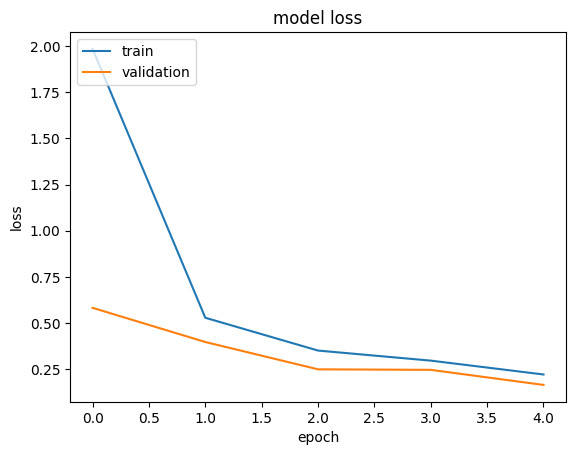
early\_stopping=EarlyStopping(patience=3,monitor="val\_loss")

vgg\_model.compile(optimizer="adam",loss="categorical\_crossentropy",metrics=['accuracy'])

history=vgg\_model.fit(x=train\_data,validation\_data=test\_data,epochs=5,callbacks=[early\_stopping],verbose=1)



plot\_curves(history)



vgg\_model.save("vgg16\_Mri.keras")

**VIVA VOICE-**

What is VGGNet, and what makes it unique?

* Answer: VGGNet is a CNN model with a deep architecture that uses 3x3 convolution filters and small strides. Its uniform design and simplicity make it effective in feature extraction for various tasks, but it is computationally heavy due to a high number of parameters.

Why is ResNet considered an improvement over traditional CNNs?

* Answer: ResNet introduces residual (skip) connections, which help in training deeper networks by preventing the vanishing gradient problem. This allows the model to learn complex features without degradation in performance as depth increases.

How does DenseNet differ from ResNet in its connections?

* Answer: DenseNet uses dense connections, meaning each layer receives input from all previous layers, encouraging feature reuse and reducing parameter count, making it more efficient and accurate with fewer resources.

What are the main advantages of using a DenseNet model?

* Answer: DenseNet is parameter-efficient, has better feature reuse, and reduces the chance of overfitting due to dense connections, which improves gradient flow and accuracy across shallow and deep architectures.

Why might VGGNet not be ideal for all tasks, despite its strong feature extraction capabilities?

* Answer: VGGNet is computationally demanding and memory-intensive, with many parameters, which makes it slower to train and less suitable for tasks with limited resources or small datasets.

**PROGRAM 5**

**AIM-**introduction to transfer learning and fine tuning in deep cnn. Also, discuss various pre-trained networks with their advantage and drawbacks.

**THEORY-**

1. Introduction to transfer learning: transfer learning is a machine learning technique where a model developed for one task is reused as the starting point for a model on a second task. In deep learning, particularly with convolutional neural networks (cnns), transfer learning has proven highly effective because cnns trained on large datasets like imagenet have already learned useful features such as edge detection, shape recognition, and texture identification. These features can be generalized across many different types of image recognition tasks.

Instead of training a deep learning model from scratch (which can be computationally expensive and time-consuming, requiring large datasets), transfer learning allows us to leverage the knowledge learned by a pre-trained model.

2. Concept of transfer learning in deep cnn: in deep cnns, lower layers learn general features like edges and textures, while higher layers learn more task-specific features. In transfer learning, we use the lower layers of a pre-trained network as a feature extractor and modify the higher layers to suit the new task.

This significantly reduces training time and provides better performance, especially when the new dataset is smaller.

3. Fine-tuning: fine-tuning is a specific type of transfer learning where you not only replace the output layer but also allow some (or all) of the pre-trained model's layers to be updated during training.

In typical transfer learning:

* The pre-trained model's layers are frozen (i.e., their weights are not updated).
* Only the newly added classifier (top layer) is trained on the new dataset.

In fine-tuning:

* Some of the pre-trained layers (usually higher layers closer to the output) are unfrozen, meaning they can learn and adjust weights based on the new dataset.
* The learning rate during fine-tuning is usually lower to ensure the changes in the weights are small and don’t drastically affect the model’s pre-learned features.

4. Types of transfer learning: there are two primary types of transfer learning in deep learning:

A. Feature extraction (frozen layers):

* Here, the pre-trained model is used as a feature extractor. The layers are frozen, and the output layer is modified to suit the new dataset. Only the new classifier is trained on the new data.
* This method works well when the dataset is small and the new task is similar to the one the model was pre-trained on.

B. Fine-tuning (unfreezing layers):

* Fine-tuning involves unfreezing some of the layers of the pre-trained model, allowing them to learn from the new dataset.
* Fine-tuning is typically used when the dataset is larger, or the new task is somewhat different from the original task.
* It allows the model to adapt the learned representations to the specifics of the new task.

5. Common pre-trained networks:

Several cnn architectures are widely used in transfer learning and fine-tuning. These networks are typically trained on large datasets like imagenet, which contains over 1 million images and 1000 classes.

6. Advantages of transfer learning:

* Reduced training time: since the base layers are already pre-trained, training time is significantly reduced.
* Better performance with less data: transfer learning allows better generalization, even with smaller datasets.
* Overcoming the need for large datasets: deep learning models require vast amounts of labeled data for training, which may not always be available. Transfer learning allows you to apply models that were trained on large datasets to your own smaller datasets.

7. Drawbacks of transfer learning:

* Domain specificity: if the task or domain of the new dataset is very different from the original dataset (e.g., medical images vs. Natural images), transfer learning may not work well.
* Large models: many pre-trained networks are quite large, requiring more computational resources, even if you're only using them as a feature extractor.
* Overfitting: in some cases, transferring too much knowledge from the pre-trained model can lead to overfitting, especially when the dataset is small.

8. Applications of transfer learning:

A. Image classification: transfer learning is frequently used in image classification tasks where models like vgg, resnet, and inception have shown excellent performance.

b. Object detection and segmentation: pre-trained models can be fine-tuned for tasks like detecting objects in an image or segmenting images into different classes.

C. Natural language processing (nlp): transfer learning is also widely used in nlp tasks where models like bert and gpt-3 are pre-trained on vast corpora of text and fine-tuned for specific language-related tasks.

D. Medical imaging: transfer learning has shown promise in medical imaging tasks like disease diagnosis, where data is scarce but critical.

Viva questions

1. What is transfer learning?
   * Transfer learning refers to using a model trained on one task as the starting point for a model on another related task.
2. Why do we use transfer learning in cnns?
   * It is used to leverage the learned features from large datasets to reduce training time and improve performance on tasks with smaller datasets.
3. What is fine-tuning in cnns?
   * Fine-tuning is the process of retraining some or all layers of a pre-trained network on a new dataset to adapt it to the new task.
4. What is the difference between transfer learning and fine-tuning?
   * In transfer learning, the pre-trained model is used without changing the weights. In fine-tuning, we adjust some or all of the weights based on the new dataset.
5. Give an example of a pre-trained cnn model.
   * Examples include vgg16, resnet, inceptionnet.

**PROGRAM-6**

**AIM -** implementation of transfer learning using pre-trained model (mobilenetv2) for image classification in python

**THEORY-**

Introduction to transfer learning:

transfer learning is a machine learning technique where a model trained on one task is repurposed to solve another related task. In deep learning, especially with convolutional neural networks (cnns), transfer learning is commonly used to leverage models pre-trained on large datasets such as imagenet for smaller and more specific tasks. This saves time and computational resources and can often result in higher accuracy than training a model from scratch, particularly when the new task has limited data.

Mobilenetv2:

mobilenetv2 is a lightweight deep neural network architecture designed by google for mobile and embedded vision applications. It is optimized to perform well even on devices with limited computational power, such as smartphones. The architecture is based on depthwise separable convolutions, which significantly reduce the number of parameters and computational cost while retaining model accuracy.

Mobilenetv2 is particularly suitable for transfer learning and fine-tuning because it balances performance and efficiency, making it an excellent choice for real-time applications or resource-constrained environments.

Mobilenetv2 architecture:

Mobilenetv2 builds on its predecessor, mobilenetv1, and introduces several key improvements:

* Depthwise separable convolutions: this technique separates convolution into two operations:
  + Depthwise convolution: applies a single filter to each input channel.
  + Pointwise convolution: applies a 1x1 convolution to combine the outputs of the depthwise convolution. This reduces the number of computations and parameters, making the model lighter.
* Inverted residuals and linear bottlenecks: in traditional residual networks (like resnet), the residual blocks allow the input to bypass layers and directly connect to the output. In mobilenetv2, these residual connections are "inverted" and involve a linear bottleneck layer. This helps in preserving the representational power while reducing complexity.
* Efficient layer design: the overall design focuses on making the network smaller and faster, allowing it to be deployed on edge devices while maintaining reasonable accuracy for a wide range of tasks.

Code and output-

Import numpy as np

Import pandas as pd

Import matplotlib.pyplot as plt

Import tensorflow as tf

# from keras.preprocessing.image import imagedatagenerator

From tensorflow.keras.preprocessing.image import imagedatagenerator

Train\_data\_generator=imagedatagenerator(rescale=1./255,shear\_range=0.2,zoom\_range=0.2,horizontal\_flip=true)

Training\_set=train\_data\_generator.flow\_from\_directory(r"output\train",target\_size=(224,224),batch\_size=32,class\_mode='binary')

Test\_data\_generator=imagedatagenerator(rescale=1./255)

Testing\_set=test\_data\_generator.flow\_from\_directory(r"output\val",target\_size=(224,224),batch\_size=32,class\_mode='binary')





From tensorflow.keras.callbacks import earlystopping

Base\_model = tf.keras.applications.mobilenetv2(input\_shape=(224,224, 3),

                                               include\_top=false,

                                               weights='imagenet')

Base\_model.trainable = false

Model = tf.keras.sequential([

    base\_model,

    tf.keras.layers.globalaveragepooling2d(),

    tf.keras.layers.dense(10, activation='softmax')  # adjust the output units to match the number of classes

])

Model.compile(optimizer=tf.keras.optimizers.adam(),

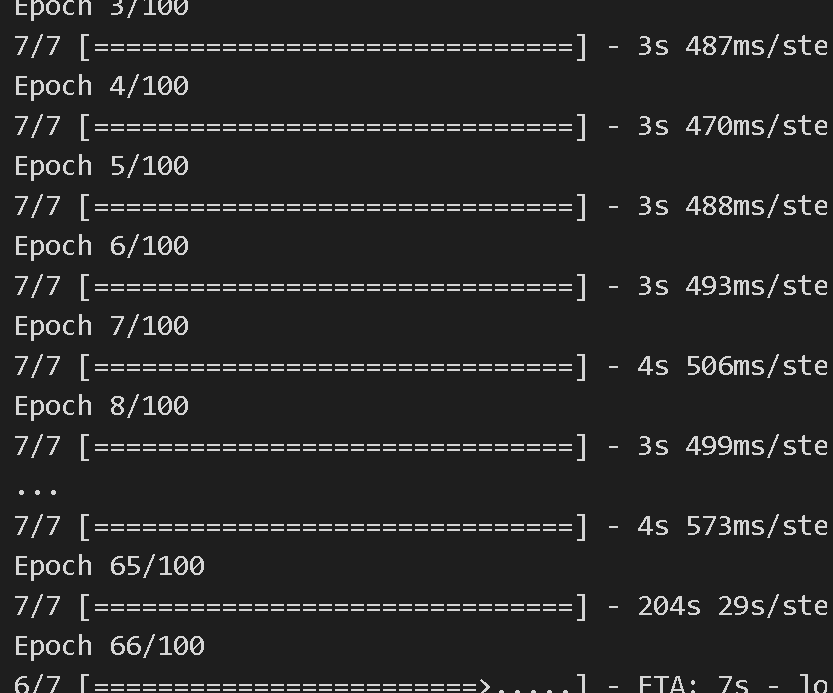
              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

# early\_stopping=earlystopping(patience=3)

Model.fit(training\_set, epochs=100, validation\_data=testing\_set)

        #   ,callbacks=[early\_stopping])



**VIVA QUESTIONS-**

1. What is mobilenetv2?

* Mobilenetv2 is a lightweight convolutional neural network designed for mobile and edge devices, utilizing depthwise separable convolutions to reduce complexity and size.

1. Why do we use mobilenetv2 for transfer learning?

* Due to its efficiency and low computational requirements, it is suitable for resource-constrained environments like mobile and embedded devices.

1. What are depthwise separable convolutions?

* Depthwise separable convolutions divide a standard convolution into two simpler operations: depthwise convolution and pointwise convolution, drastically reducing the number of parameters.

1. What is the importance of freezing layers in transfer learning?

* Freezing layers ensures that the pre-trained weights are not modified during initial training, preserving the learned features.

1. At which point should you unfreeze layers during transfer learning?

* After training the new classifier on top of the frozen base model, you may unfreeze some layers and fine-tune the model to improve performance on the new task.

**PROGRAM NO. 4**

**AIM**-Implement CNN to classify COVID-19 X-ray images in Python.

**THEORY-**

Convolutional Neural Networks (CNNs) are a powerful tool for machine learning, especially in tasks related to computer vision. Convolutional Neural Networks, or CNNs, are a specialized class of neural networks designed to effectively process grid-like data, such as images.

In this article, we are going to discuss convolutional neural networks (CNN) in machine learning in detail.

What is Convolutional Neural Network(CNN)?

A Convolutional Neural Network (CNN) is a type of deep learning algorithm that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The architecture of CNNs is inspired by the visual processing in the human brain, and they are well-suited for capturing hierarchical patterns and spatial dependencies within images.

CNNs are trained using a large dataset of labeled images, where the network learns to recognize patterns and features that are associated with specific objects or classes. Proven to be highly effective in image-related tasks, achieving state-of-the-art performance in various computer vision applications. Their ability to automatically learn hierarchical representations of features makes them well-suited for tasks where the spatial relationships and patterns in the data are crucial for accurate predictions. CNNs are widely used in areas such as image classification, object detection, facial recognition, and medical image analysis.

The convolutional layers are the key component of a CNN, where filters are applied to the input image to extract features such as edges, textures, and shapes.

The output of the convolutional layers is then passed through pooling layers, which are used to down-sample the feature maps, reducing the spatial dimensions while retaining the most important information. The output of the pooling layers is then passed through one or more fully connected layers, which are used to make a prediction or classify the image.

Convolutional Neural Network Training

CNNs are trained using a supervised learning approach. This means that the CNN is given a set of labeled training images. The CNN then learns to map the input images to their correct labels.

The training process for a CNN involves the following steps:

1. Data Preparation: The training images are preprocessed to ensure that they are all in the same format and size.
2. Loss Function: A loss function is used to measure how well the CNN is performing on the training data. The loss function is typically calculated by taking the difference between the predicted labels and the actual labels of the training images.
3. Optimizer: An optimizer is used to update the weights of the CNN in order to minimize the loss function.
4. Backpropagation: Backpropagation is a technique used to calculate the gradients of the loss function with respect to the weights of the CNN. The gradients are then used to update the weights of the CNN using the optimizer.

CNN Evaluation

After training, CNN can be evaluated on a held-out test set. A collection of pictures that the CNN has not seen during training makes up the test set. How well the CNN performs on the test set is a good predictor of how well it will function on actual data.

The efficiency of a CNN on picture categorization tasks can be evaluated using a variety of criteria. Among the most popular metrics are:

* Accuracy: Accuracy is the percentage of test images that the CNN correctly classifies.

1. Precision: Precision is the percentage of test images that the CNN predicts as a particular class and that are actually of that class.
2. Recall: Recall is the percentage of test images that are of a particular class and that the CNN predicts as that class.

CODE and OUTPUT-

import tensorflow as tf

import matplotlib.pyplot as plt

from PIL import Image

tf.test.is\_gpu\_available()train\_dir=r"archive (6)\Data\train"

test\_dir=r"archive (6)/Data/test"

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_data\_generator=ImageDataGenerator(rescale=1./255,shear\_range=0.2,zoom\_range=0.2,horizontal\_flip=True)

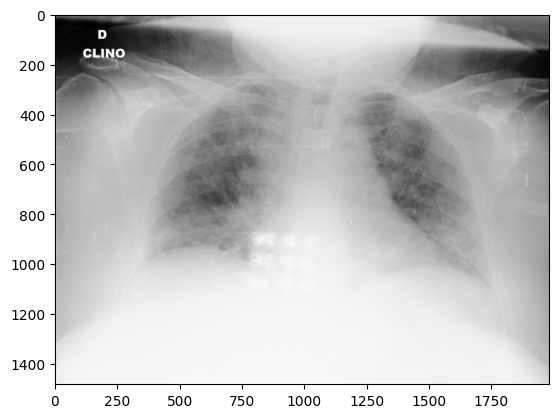
training\_set=train\_data\_generator.flow\_from\_directory(r"archive (6)/Data/train",target\_size=(64,64),batch\_size=1,class\_mode='categorical')



test\_data\_generator=ImageDataGenerator(rescale=1./255,)

testing\_set=test\_data\_generator.flow\_from\_directory(r"archive (6)/Data/test",target\_size=(64,64),batch\_size=1,class\_mode='categorical')



plt.imshow(Image.open(r"archive (6)/Data/train/COVID19/COVID19(1).jpg"))

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras import models, layers

from tensorflow.keras.layers import \*

model = models.Sequential()

model.add(layers.Conv2D(64, (3, 3), activation='relu',input\_shape=(64,64,3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2,2)))

Flatten the 3D output to 1D

model.add(layers.Flatten())

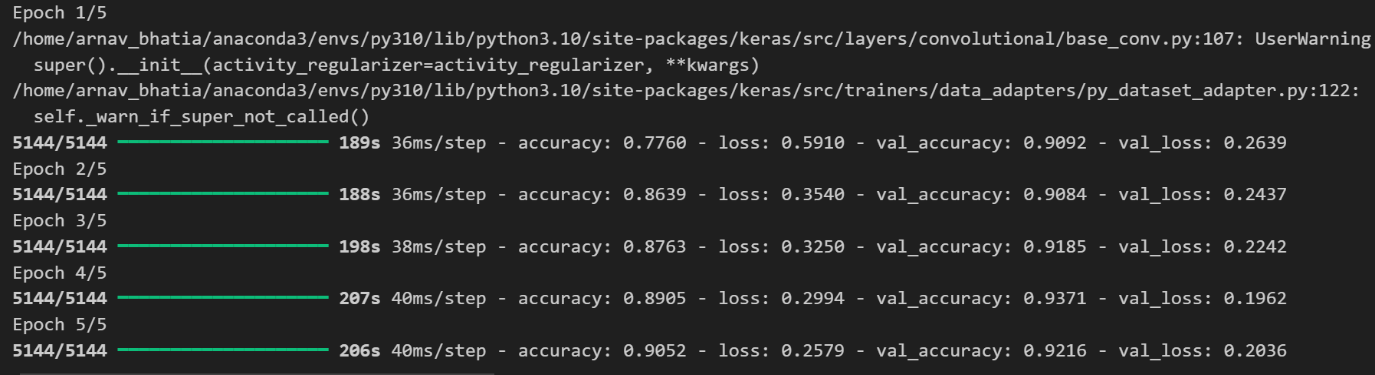
model.add(layers.Dense(64, activation='relu'))

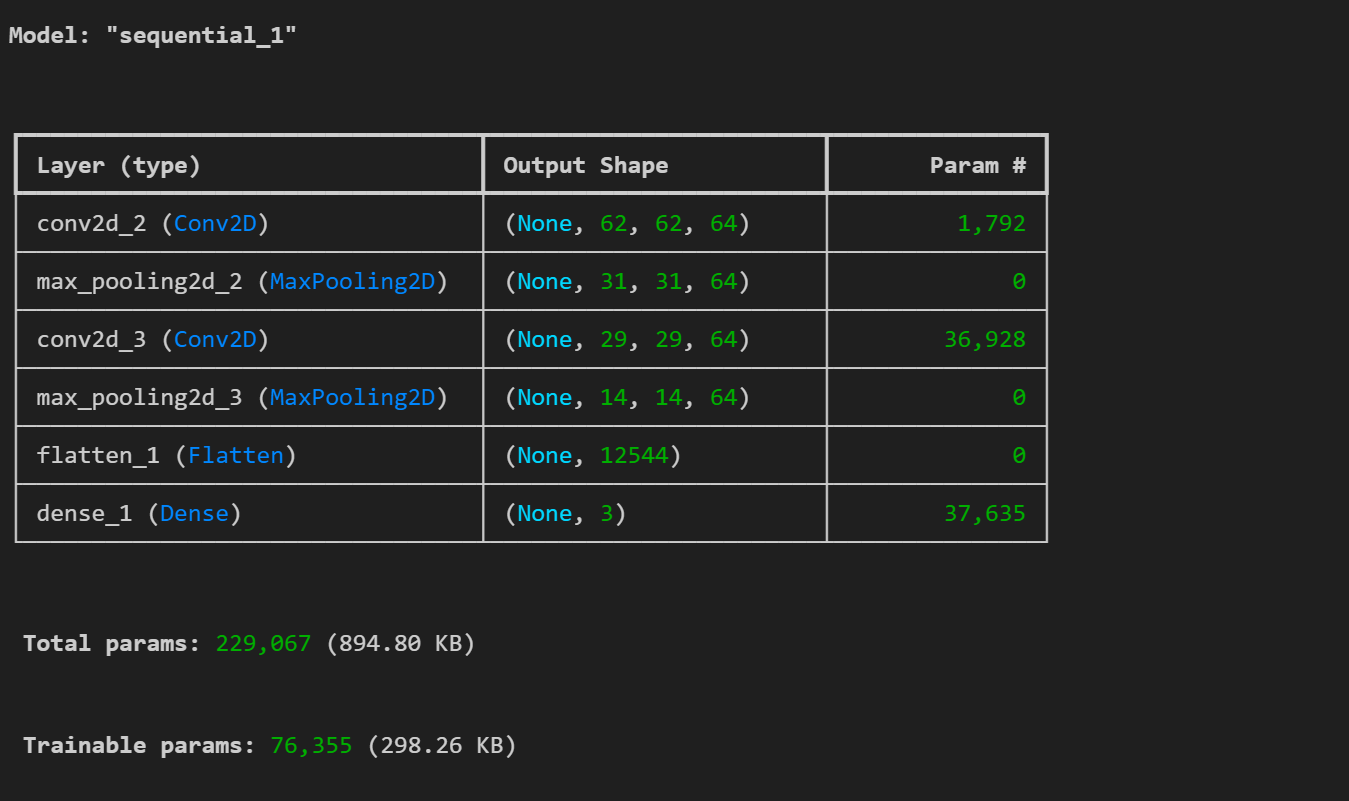
model.add(layers.Dense(3, activation='sigmoid'))

early\_stopping = EarlyStopping(patience=3)

model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=["accuracy"])

model.fit(x=training\_set, validation\_data=testing\_set, epochs=5, callbacks=[early\_stopping])

model.summary()



**VIVA QUESTIONS-**

1. What is the ReLU activation function, and why is it used in CNNs?

Answer:  
ReLU (Rectified Linear Unit) is an activation function that introduces non-linearity into the network. It outputs the input directly if it is positive, otherwise, it outputs zero. It helps the network learn complex patterns.

2.Why are fully connected layers used at the end of CNNs?

Answer:  
Fully connected layers (FC layers) at the end of CNNs take the high-level features extracted by convolutional layers and combine them to make final predictions. These layers allow for classification or regression tasks by outputting the final scores or probabilities.

3. What is the purpose of feature maps in CNN?

Answer:  
Feature maps represent the output of a convolutional layer after applying filters to the input image. They highlight the presence of certain features (such as edges or textures) in different spatial regions of the image.

4. What is the difference between local and global features in CNN?

Answer:  
Local features are extracted by convolutional layers and correspond to small patterns (e.g., edges, corners) in specific regions of the image. Global features, learned in deeper layers or fully connected layers, represent the overall structure or semantics of the image (e.g., object shapes).

5. How does CNN handle overfitting?

Answer:  
Overfitting in CNN can be reduced through:

* Data augmentation: artificially increasing the size of the dataset.
* Dropout: randomly dropping neurons during training.
* L2 regularization: penalizing large weights.
* Batch normalization: normalizing the inputs to each layer.

6. What is transfer learning, and how is it applied to CNNs?

Answer:  
Transfer learning involves taking a pre-trained CNN model (trained on a large dataset like ImageNet) and fine-tuning it for a different but related task by retraining only the final few layers or the entire network.

**PROGRAM NO. 3**

**AIM**-Implementation of Convolutional Neural Network for MRI dataset in Python.

**THEORY-**

Introduction:

Convolutional Neural Networks (CNNs) are a specialized class of artificial neural networks that have proven to be extremely effective for processing and analyzing visual data, such as images or videos. They were inspired by the structure of the visual cortex in animals, where different neurons respond to specific local regions of the visual field.

CNNs have revolutionized computer vision tasks such as image classification, object detection, and face recognition, and are a fundamental part of modern deep learning architectures.

Key Characteristics of CNNs:

1. Local Receptive Fields:

- CNNs exploit the local spatial structure of images. Filters are small and scan local regions (receptive fields) of the input image, capturing local patterns such as edges, textures, and corners.

- As we move deeper into the network, the receptive field increases, allowing the network to capture more global patterns and high-level features.

2. Weight Sharing:

- In fully connected networks, each neuron has its own set of weights. In CNNs, however, the same filter (set of weights) is applied across the entire input, which drastically reduces the number of parameters and computations.

- Weight sharing helps CNNs generalize better and reduces the risk of overfitting.

3. Translation Invariance:

- By using convolutional and pooling layers, CNNs become invariant to translations and small distortions in the input. For example, if an object in an image shifts slightly, CNNs can still recognize it due to the way features are extracted.

4. Hierarchical Feature Learning:

- CNNs learn hierarchical representations of images. The initial layers detect low-level features like edges, while deeper layers capture more complex patterns such as shapes and objects.

Training CNNs:

Training a CNN involves the following key steps:

1. Forward Propagation:

- The input image passes through several convolutional, activation, pooling, and fully connected layers, and the network outputs predictions based on the learned features.

2. Loss Function:

- The loss function (e.g., cross-entropy for classification tasks) calculates the difference between the predicted output and the actual labels.

3. Backpropagation:

- The gradients of the loss function with respect to the network parameters are computed using backpropagation. These gradients are then used to update the weights of the filters and fully connected layers via an optimization algorithm (like Stochastic Gradient Descent or Adam).

4. Optimization:

- The goal of training is to minimize the loss function. Optimization algorithms like Stochastic Gradient Descent (SGD) or Adam adjust the network's weights to minimize the prediction error.

Applications of CNNs:

1. Image Classification:

- CNNs are widely used for classifying images into predefined categories. Examples include facial recognition, object recognition, and medical imaging.

2. Object Detection:

- CNNs can be extended to detect objects within an image and place bounding boxes around them. Examples include YOLO (You Only Look Once) and Faster R-CNN.

3. Segmentation:

- In image segmentation, CNNs are used to label each pixel in an image as belonging to a particular class (e.g., background, road, building in an autonomous driving scenario).

4. Video Processing:

- CNNs can be used to analyze video frames for tasks such as action recognition, scene understanding, and video classification.

Conclusion:

CNNs have become the backbone of many computer vision applications due to their ability to automatically and efficiently capture important patterns in image data. By using layers of convolutions, pooling, and activations, CNNs build a hierarchy of features that allow them to perform tasks like image classification, object detection, and image segmentation with high accuracy.

CODE and OUTPUT-

import tensorflow as tf

import pandas as pd

import sklearn

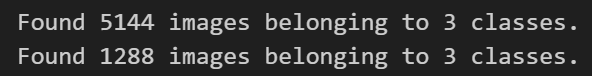
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_data\_generator=ImageDataGenerator(rescale=1./255,shear\_range=0.2,zoom\_range=0.2,horizontal\_flip=True)

training\_set=train\_data\_generator.flow\_from\_directory(r"archive (6)/Data/train",target\_size=(64,64),batch\_size=32,class\_mode='binary')

test\_data\_generator=ImageDataGenerator(rescale=1./255,)

testing\_set=test\_data\_generator.flow\_from\_directory(r"archive (6)/Data/test",target\_size=(64,64),batch\_size=32,class\_mode='binary')



from tensorflow.keras.callbacks import EarlyStopping

cnn=tf.keras.models.Sequential([

    tf.keras.layers.Conv2D(filters=32,kernel\_size=3,activation="relu",input\_shape=[64,64,3]),

    tf.keras.layers.MaxPool2D(strides=2,pool\_size=2),

    tf.keras.layers.Conv2D(filters=32,kernel\_size=3,activation="relu"),

    tf.keras.layers.MaxPool2D(strides=2,pool\_size=2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(activation="relu",units=128),

    tf.keras.layers.Dense(activation="sigmoid",units=1)]

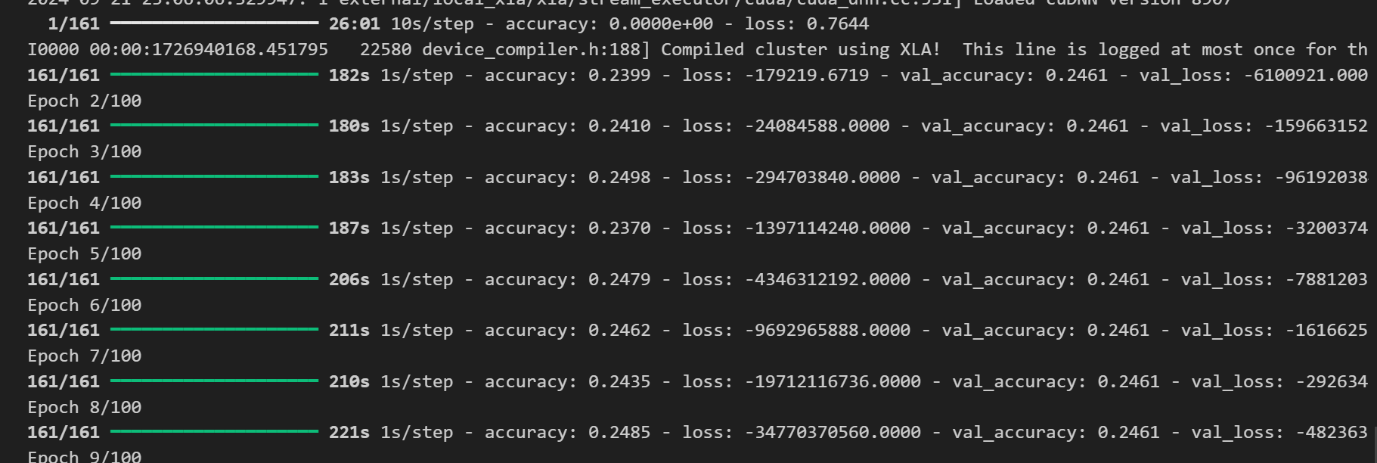
)

early\_stopping=EarlyStopping(patience=3)

cnn.compile(optimizer="adam",loss="binary\_crossentropy",metrics=["accuracy"])

cnn.fit(x=training\_set,validation\_data=testing\_set,epochs=100,callbacks=[early\_stopping])

cnn.save("cnn1.h5")



**VIVA QUESTIONS-**

1. What is a Convolutional Neural Network (CNN)?

Answer:  
A CNN is a class of deep learning models that is specifically designed to process and recognize patterns in visual data, such as images or videos. It automatically captures spatial hierarchies in data through convolutional layers, pooling layers, and fully connected layers.

2. What is the role of the convolutional layer in CNN?

Answer:  
The convolutional layer applies filters (or kernels) to the input image or previous layer to extract features such as edges, textures, or other important characteristics by performing element-wise multiplications and summing the result.

3. What are filters in CNN, and how do they work?

Answer:  
Filters (or kernels) are small matrices that slide across the input image, applying convolution operations. They help detect specific patterns (such as edges, textures) by capturing local spatial features of the input data.

4. What is stride in CNN?

Answer:  
Stride refers to the number of pixels by which the filter shifts over the input image. A stride of 1 moves the filter by one pixel, while a stride of 2 skips one pixel for each movement, reducing the output size.

5. What is padding in CNN, and why is it used?

Answer:  
Padding involves adding extra pixels (usually zeros) around the border of the input image to maintain the spatial dimensions after the convolution operation. It ensures that the features at the edges of the image are also considered.

6. What is pooling, and what types of pooling are used in CNNs?

Answer:  
Pooling is a down-sampling operation that reduces the spatial dimensions of the feature maps. The most common types are Max Pooling (which takes the maximum value) and Average Pooling (which takes the average value) within a region of the feature map.