PROGRAM 7

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

THEORY-

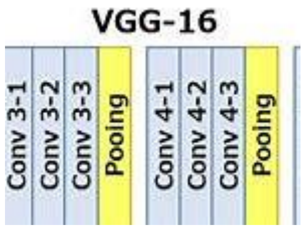
Overview of VGG-16:

- 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:

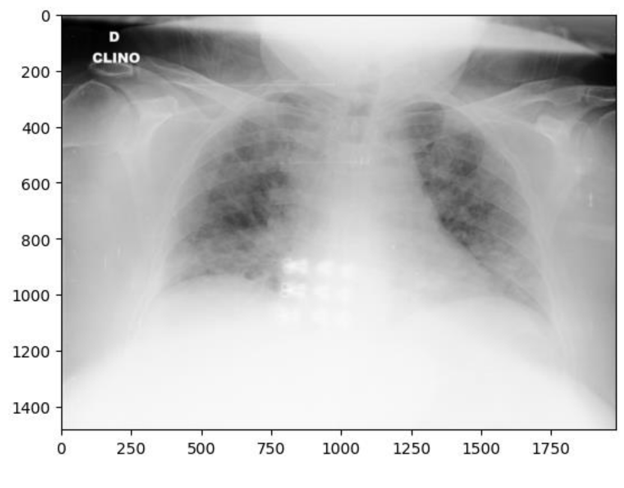
- Large Memory Requirement: With around 138 million parameters, VGG-16 is memory-intensive, making it difficult to deploy on devices with limited resources.
- High Computation Cost: The depth and number of parameters result in a high computational cost, which can be slow without specialized hardware (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)

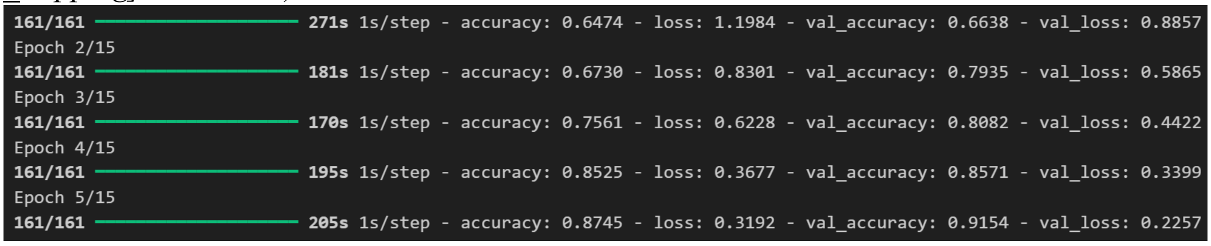
predictions = tensorflow.keras.layers.Dense(num_classes, activation='softmax')(x)

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'])

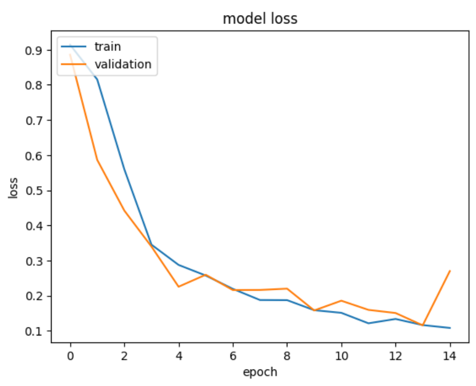
history=vgg_model.fit(x=train_data,validation_data=test_data,epochs=15,callbacks=[early_stopping],verbose=1)

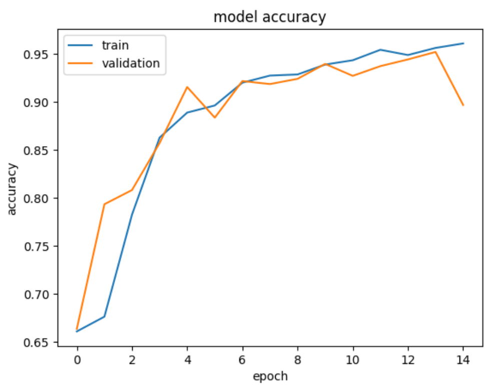


```
161/161 ━━━━━━━━━━━━━━━━━━━━ 271s 1s/step - accuracy: 0.6474 - loss: 1.1984 - val_accuracy: 0.6638 - val_loss: 0.8857
Epoch 2/15
161/161 ━━━━━━━━━━━━━━━━━━━━ 181s 1s/step - accuracy: 0.6730 - loss: 0.8301 - val_accuracy: 0.7935 - val_loss: 0.5865
Epoch 3/15
161/161 ━━━━━━━━━━━━━━━━━━━━ 170s 1s/step - accuracy: 0.7561 - loss: 0.6228 - val_accuracy: 0.8082 - val_loss: 0.4422
Epoch 4/15
161/161 ━━━━━━━━━━━━━━━━━━━━ 195s 1s/step - accuracy: 0.8525 - loss: 0.3677 - val_accuracy: 0.8571 - val_loss: 0.3399
Epoch 5/15
161/161 ━━━━━━━━━━━━━━━━━━━━ 205s 1s/step - accuracy: 0.8745 - loss: 0.3192 - val_accuracy: 0.9154 - val_loss: 0.2257
```

```
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.

2. **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.

3. **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.

4. **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.

5. **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.

THEORY-

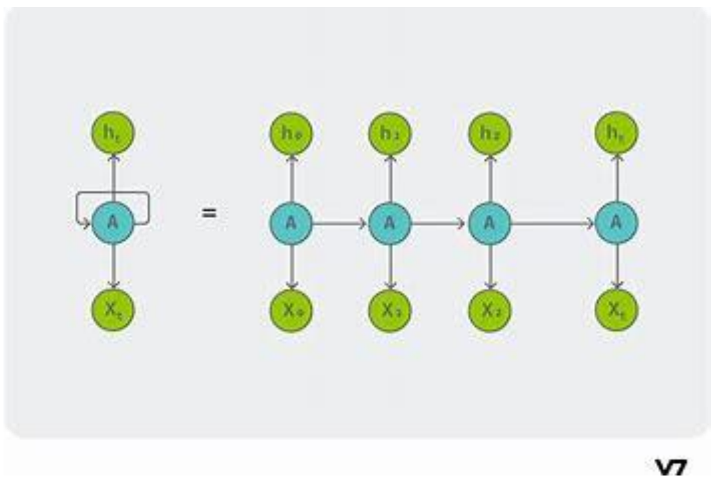
Overview of Recurrent Neural Networks (RNNs):

- 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 Layer**: The RNN processes sequences in a time-dependent manner, taking input at each time step. Each input at time ttt, denoted as xtx_txt, is a part of a sequence {x1,x2,…,xT}\{x_1, x_2, \ldots, x_T\}{x1,x2,…,xT}.
- **Hidden Layer**: The core component of an RNN is the hidden layer, which is recurrent, meaning it retains a "memory" by passing information from one time step to the next. The hidden state at time ttt, hth_tht, depends on both the current input

xtx_txt and the previous hidden state ht−1h_{t-1}ht−1, allowing it to carry information through time.

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b_h)$$

Here, WhW_hWh and WxW_xWx are the weight matrices for the previous hidden state and the current input, respectively, and bhb_hbh is the bias term. The non-linearity (often a tanh or ReLU function) allows the RNN to learn complex patterns.

- **Output Layer**: At each time step, the hidden state is transformed to produce an output yty_tyt, which can be used immediately or passed into the next time step. This output is typically calculated by:

$$y_t = \text{activation}(W_y \cdot h_t + b_y)$$

where WyW_yWy is the weight matrix for the hidden state and byb_yby is a bias term.

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.

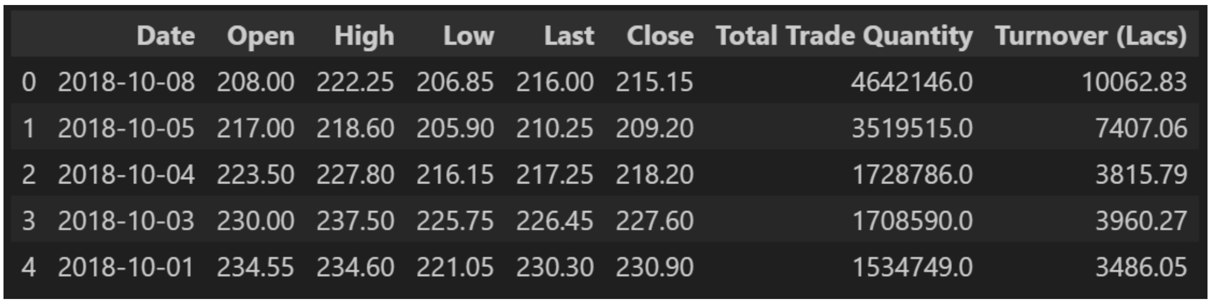
Challenges with Standard RNNs:

- 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-

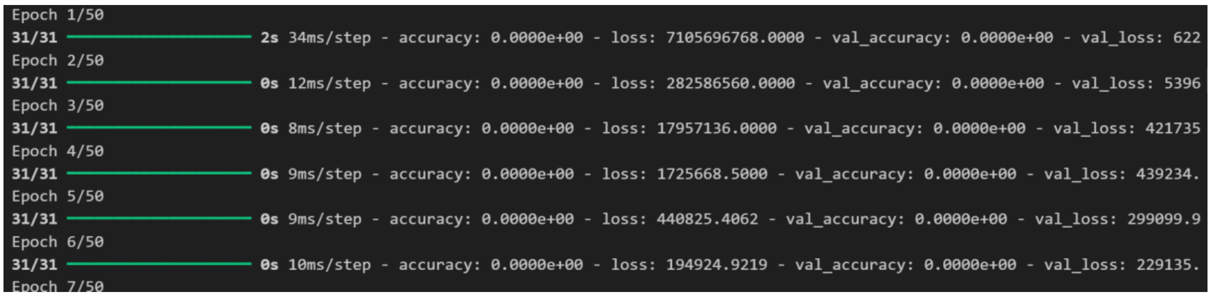
```
import pandas as pd

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()
```



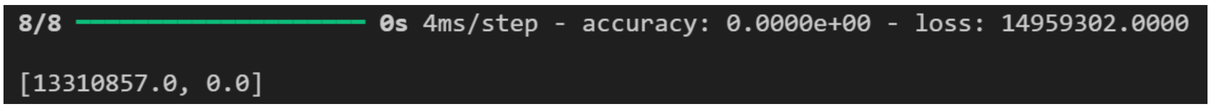
	Date	Open	High	Low	Last	Close	Total Trade Quantity	Turnover (Lacs)
0	2018-10-08	208.00	222.25	206.85	216.00	215.15	4642146.0	10062.83
1	2018-10-05	217.00	218.60	205.90	210.25	209.20	3519515.0	7407.06
2	2018-10-04	223.50	227.80	216.15	217.25	218.20	1728786.0	3815.79
3	2018-10-03	230.00	237.50	225.75	226.45	227.60	1708590.0	3960.27
4	2018-10-01	234.55	234.60	221.05	230.30	230.90	1534749.0	3486.05

```
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_t
est,np.float32)],batch_size=32,epochs=50)
```



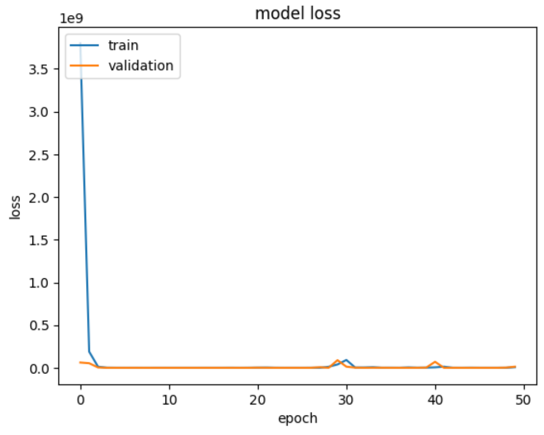
```
Epoch 1/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 2s 34ms/step - accuracy: 0.0000e+00 - loss: 7105696768.0000 - val_accuracy: 0.0000e+00 - val_loss: 622
Epoch 2/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.0000e+00 - loss: 282586560.0000 - val_accuracy: 0.0000e+00 - val_loss: 5396
Epoch 3/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.0000e+00 - loss: 17957136.0000 - val_accuracy: 0.0000e+00 - val_loss: 421735
Epoch 4/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.0000e+00 - loss: 1725668.5000 - val_accuracy: 0.0000e+00 - val_loss: 439234.
Epoch 5/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.0000e+00 - loss: 440825.4062 - val_accuracy: 0.0000e+00 - val_loss: 299099.9
Epoch 6/50
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.0000e+00 - loss: 194924.9219 - val_accuracy: 0.0000e+00 - val_loss: 229135.
Epoch 7/50
```

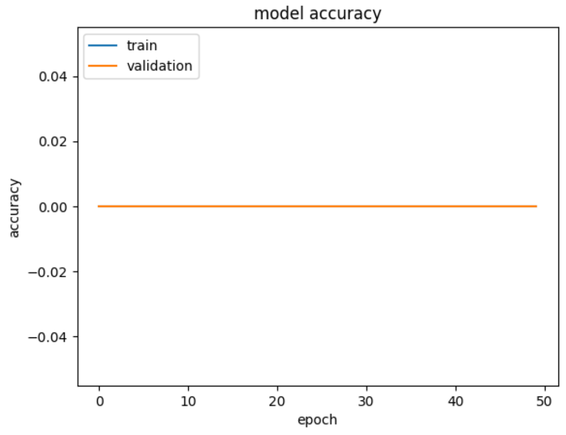
model.evaluate(tf.convert_to_tensor(x_test,np.float32),tf.convert_to_tensor(y_test,np.float32)
)



```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0000e+00 - loss: 14959302.0000

[13310857.0, 0.0]
```

```
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 11

AIM- Implementation of Autoencoders on Image 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:

- The encoder compresses the input data into a lower-dimensional **Encoder:** 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:

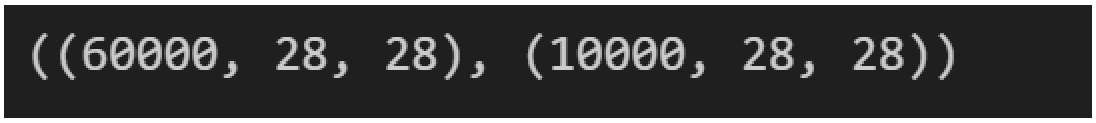
- Autoencoders are trained to minimize reconstruction loss, which is the difference between the original input and its reconstructed output. Common loss functions include Mean Squared Error (MSE) for continuous data or Binary Cross-Entropy for binary data.

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.

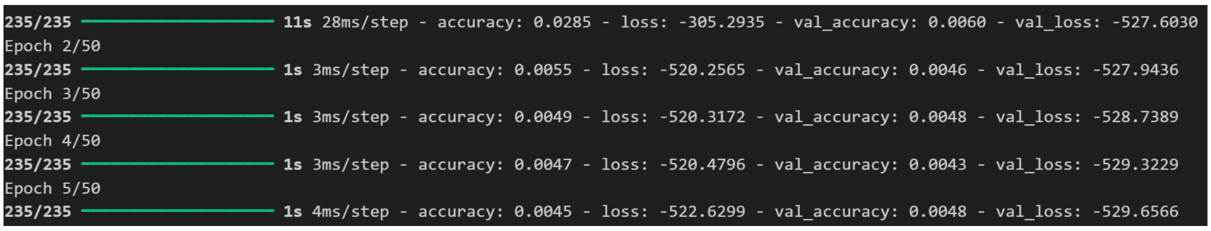
CODE AND OUTPUT-

```
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
```



```
((60000, 28, 28), (10000, 28, 28))
```

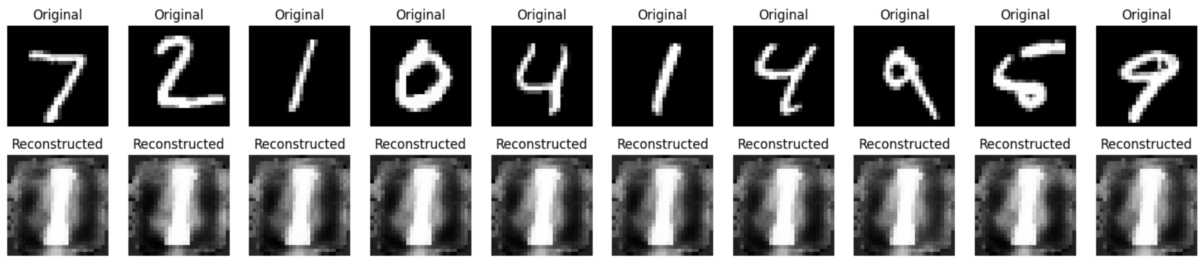
```
# Build the Autoencoder model
input_img = Input(shape=(28,28))
x=tf.keras.layers.Rescaling(1./255)(input_img)
x=tf.keras.layers.Flatten()(x)# 28x28 = 784
#! Encoder
encoded = Dense(64, activation='relu')(x)
#! Decoder
decoded = Dense(784, activation='sigmoid')(encoded)
decoded = Reshape((28,28))(decoded)
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))
```



```
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 28ms/step - accuracy: 0.0285 - loss: -305.2935 - val_accuracy: 0.0060 - val_loss: -527.6030
Epoch 2/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.0055 - loss: -520.2565 - val_accuracy: 0.0046 - val_loss: -527.9436
Epoch 3/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.0049 - loss: -520.3172 - val_accuracy: 0.0048 - val_loss: -528.7389
Epoch 4/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.0047 - loss: -520.4796 - val_accuracy: 0.0043 - val_loss: -529.3229
Epoch 5/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0045 - loss: -522.6299 - val_accuracy: 0.0048 - val_loss: -529.6566
```

```
# ! 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-

1. **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.

2. **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.

3. **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.

4. **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.

5. **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:

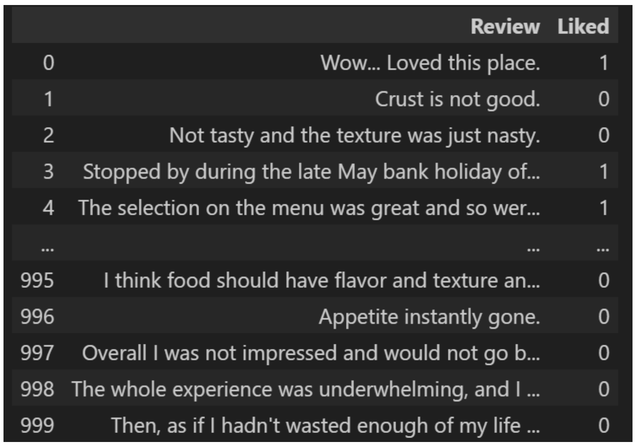
- **Bag of Words (BoW):** Represents text as a collection of words without considering their order. It creates a vocabulary of unique words in the dataset, and each document is represented by a vector of word counts.
- **TF-IDF (Term Frequency-Inverse Document Frequency**): A statistical measure that 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-

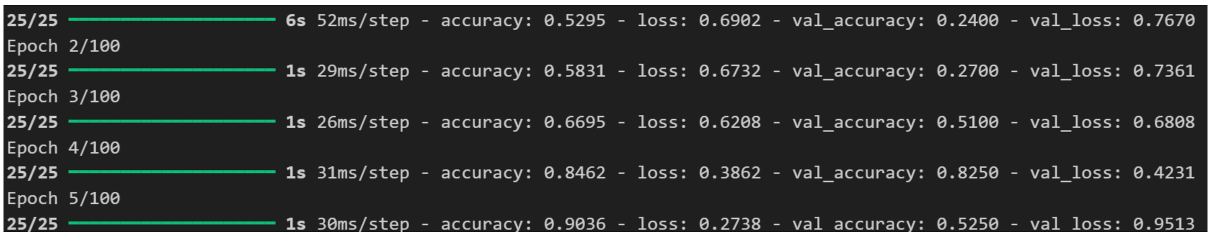
```
import pandas as pd
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras import *
from tensorflow.keras.layers import *
tf.test.is_gpu_available()
df=pd.read_csv("Restaurant_Reviews.tsv",sep="\t")
df
```



```
                                               Review  Liked
0                            Wow... Loved this place.      1
1                                  Crust is not good.      0
2            Not tasty and the texture was just nasty.      0
3      Stopped by during the late May bank holiday of...      1
4      The selection on the menu was great and so wer...      1
...                                               ...    ...
995    I think food should have flavor and texture an...      0
996                           Appetite instantly gone.      0
997    Overall I was not impressed and would not go b...      0
998  The whole experience was underwhelming, and I ...      0
999    Then, as if I hadn't wasted enough of my life ...      0
```

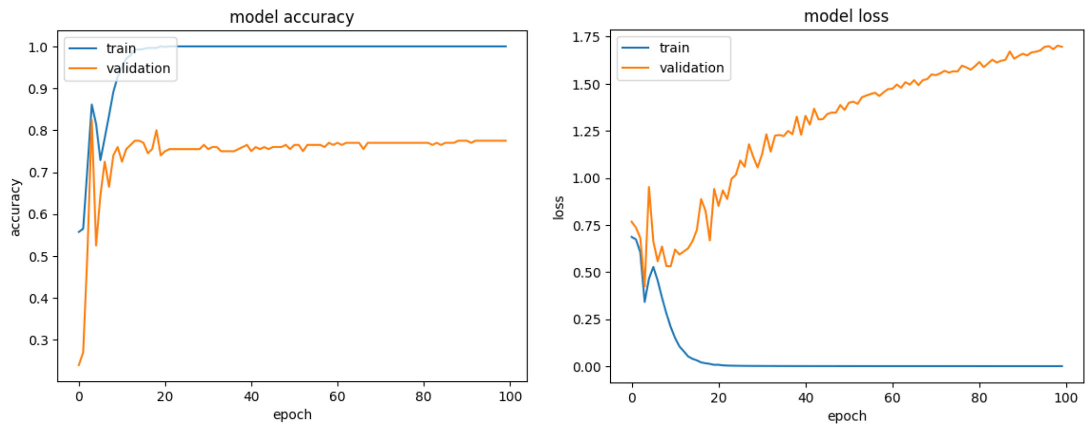
```
reviews = df['Review'].values
labels = df['Liked'].values
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
# 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)
```

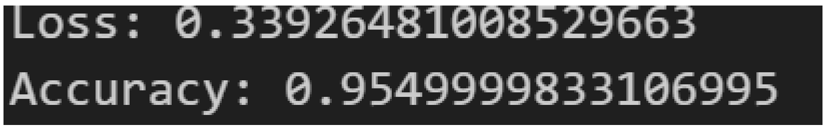


```
25/25 ━━━━━━━━━━━━━━━━━━━━ 6s 52ms/step - accuracy: 0.5295 - loss: 0.6902 - val_accuracy: 0.2400 - val_loss: 0.7670
Epoch 2/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.5831 - loss: 0.6732 - val_accuracy: 0.2700 - val_loss: 0.7361
Epoch 3/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6695 - loss: 0.6208 - val_accuracy: 0.5100 - val_loss: 0.6808
Epoch 4/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.8462 - loss: 0.3862 - val_accuracy: 0.8250 - val_loss: 0.4231
Epoch 5/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.9036 - loss: 0.2738 - val_accuracy: 0.5250 - val_loss: 0.9513
```

```
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)
```

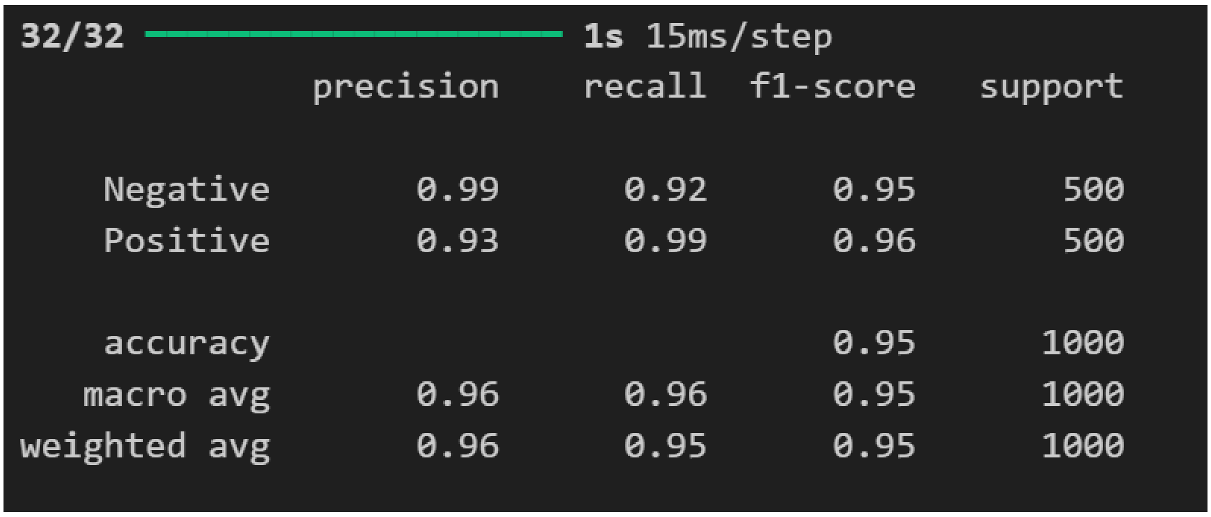


```
model.save("nlp.keras")
# Evaluate the model on the validation data
loss, accuracy = model.evaluate(padded_sequences, labels, verbose=2)
print(f"Loss: {loss}")
print(f"Accuracy: {accuracy}")
```



```
Loss: 0.33926481008529663
Accuracy: 0.9549999833106995
```

```
from sklearn.metrics import classification_report
predictions = (model.predict(padded_sequences) > 0.5).astype("int32")
print(classification_report(labels, predictions, target_names=["Negative", "Positive"]))
```



```
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step
              precision    recall  f1-score   support

    Negative       0.99      0.92      0.95       500
    Positive       0.93      0.99      0.96       500

    accuracy                           0.95      1000
   macro avg       0.96      0.96      0.95      1000
weighted avg       0.96      0.95      0.95      1000
```

VIVA VOICE-

1. **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.

2. **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.

3. **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.

4. **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.

5. **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)

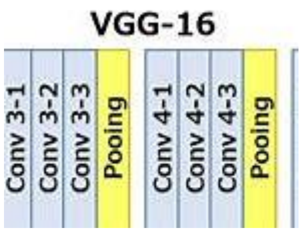
THEORY-

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.

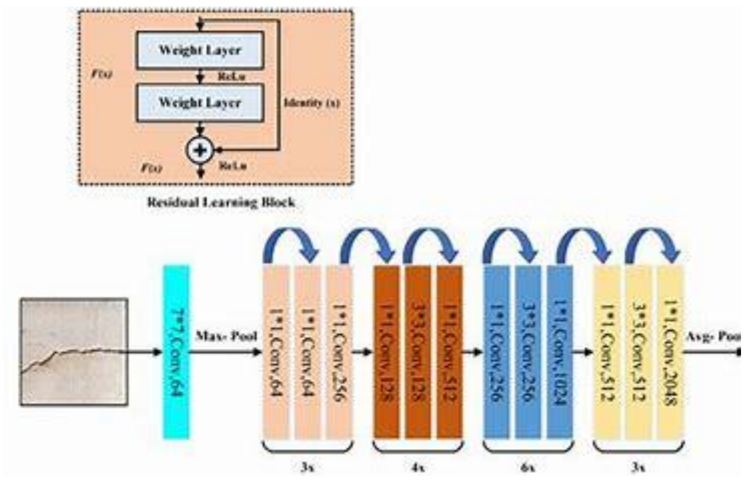
Pre-Trained Models in Image Classification:

a. VGGNet:



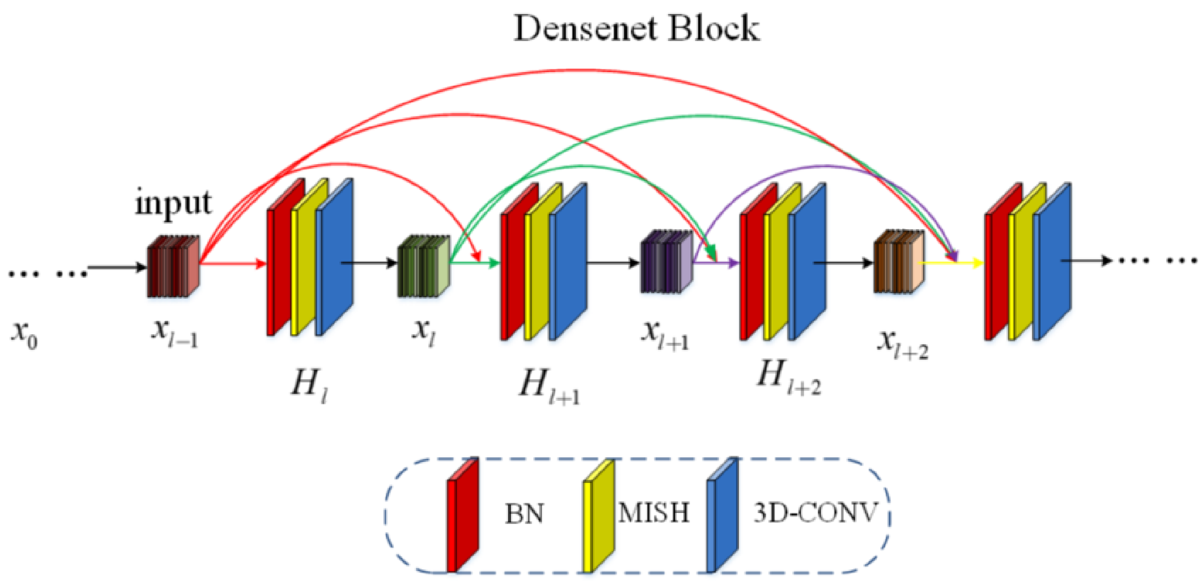
- Architecture: VGGNet uses a deep CNN structure with multiple convolutional layers and pooling layers. It’s characterized by simplicity, as it employs only 3x3 convolution filters and small strides.
- Strengths: VGGNet is effective at extracting spatial features due to its deep architecture. Its uniform 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:

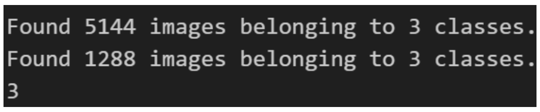
- VGGNet: Good for straightforward feature extraction but requires high computational power.
- ResNet: Handles deep architectures efficiently, best suited for tasks needing complex feature extraction.
- DenseNet: Parameter-efficient and highly accurate but requires more memory, ideal for cases needing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

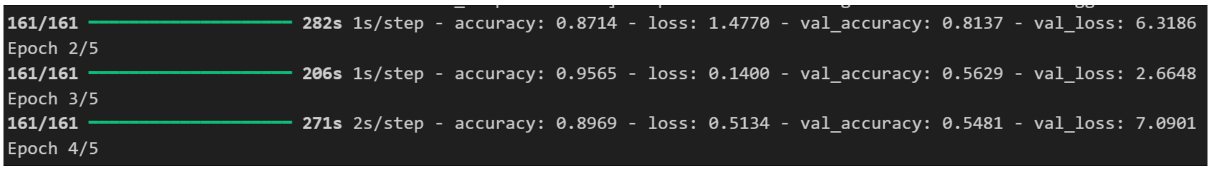
```
import pandas as pd

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_siz
e=32,class_mode='categorical')

num_classes=train_data.num_classes

print(num_classes)
```



```
Found 5144 images belonging to 3 classes.
Found 1288 images belonging to 3 classes.
3
```

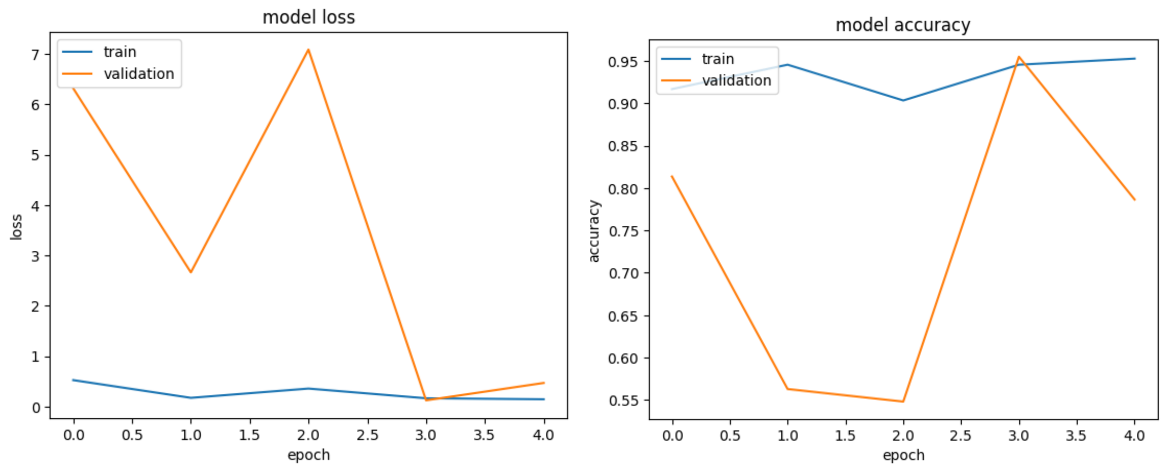
```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping=EarlyStopping(patience=3,monitor="val_loss")
def plot_curves(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    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])
```



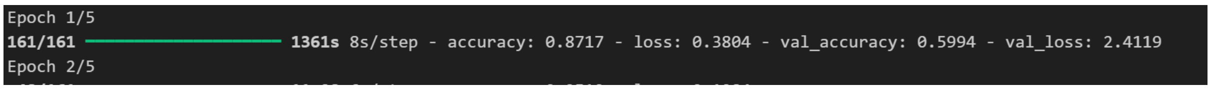
```
161/161 ━━━━━━━━━━━━━━━━━━━━ 282s 1s/step - accuracy: 0.8714 - loss: 1.4770 - val_accuracy: 0.8137 - val_loss: 6.3186
Epoch 2/5
161/161 ━━━━━━━━━━━━━━━━━━━━ 206s 1s/step - accuracy: 0.9565 - loss: 0.1400 - val_accuracy: 0.5629 - val_loss: 2.6648
Epoch 3/5
161/161 ━━━━━━━━━━━━━━━━━━━━ 271s 2s/step - accuracy: 0.8969 - loss: 0.5134 - val_accuracy: 0.5481 - val_loss: 7.0901
Epoch 4/5
```

```
plot_curves(history_resnet_50)
```



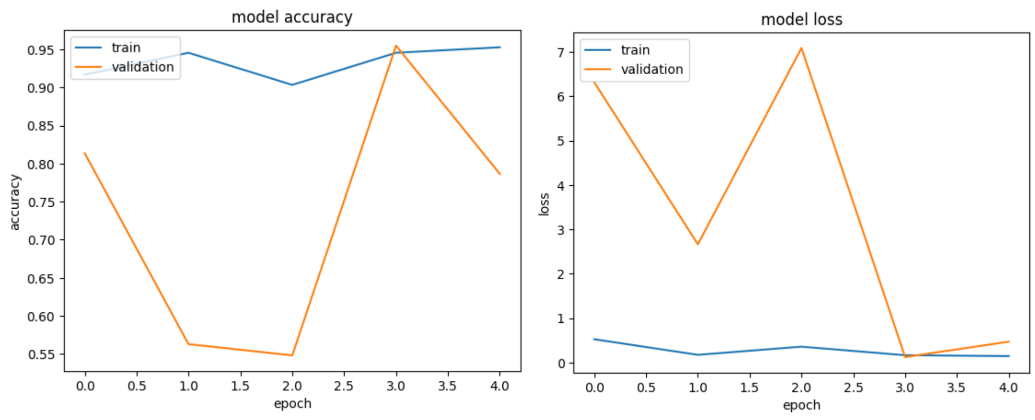
```
model_resnet_50.save("resnet50.keras")

## 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])
```

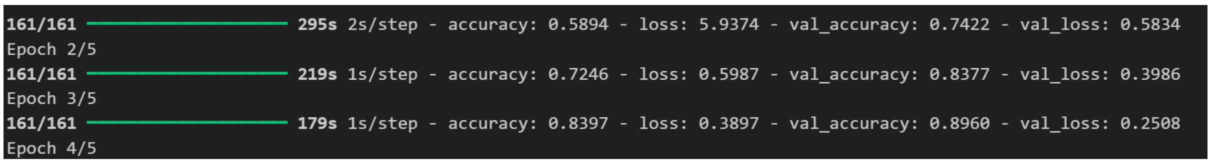


```
Epoch 1/5
161/161 ━━━━━━━━━━━━━━━━━━━━ 1361s 8s/step - accuracy: 0.8717 - loss: 0.3804 - val_accuracy: 0.5994 - val_loss: 2.4119
Epoch 2/5
```

```
plot_curves(history_densenet_201)
```

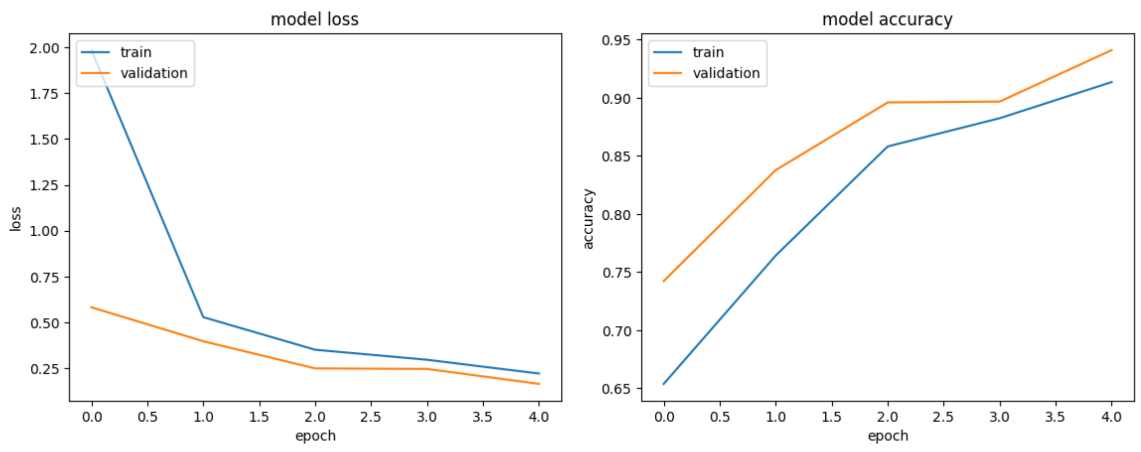


```
model_densenet_201.save("Densenet.keras")

## 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)
```



```
161/161 ━━━━━━━━━━━━━━━━━━━━ 295s 2s/step - accuracy: 0.5894 - loss: 5.9374 - val_accuracy: 0.7422 - val_loss: 0.5834
Epoch 2/5
161/161 ━━━━━━━━━━━━━━━━━━━━ 219s 1s/step - accuracy: 0.7246 - loss: 0.5987 - val_accuracy: 0.8377 - val_loss: 0.3986
Epoch 3/5
161/161 ━━━━━━━━━━━━━━━━━━━━ 179s 1s/step - accuracy: 0.8397 - loss: 0.3897 - val_accuracy: 0.8960 - val_loss: 0.2508
Epoch 4/5
```

plot_curves(history)



vgg_model.save("vgg16_Mri.keras")

VIVA VOICE-

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.