Program 1

Write a program to demonstrate the working of a deep neural network for classification task.

Approach A

In [ ]:
```
!pip install tensorflow
```

In [ ]:
```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D

(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

In [ ]:
```
y_train
```

Out[ ]:
```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

In [ ]:
```
!pip install np_utils

from tensorflow.keras.utils import to_categorical
import tensorflow as tf
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
y_train=tf.keras.utils.to_categorical(y_train,n_classes)
y_test =tf.keras.utils.to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", y_train.shape)
```

```
Collecting np_utils
  Downloading np_utils-0.6.0.tar.gz (61 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/62.0 kB ? eta -:--:--
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.0/62.0 kB 2.6 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy>=1.0 in /usr/local/lib/python3.11/dist-packages (from np_
utils) (2.0.2)
Building wheels for collected packages: np_utils
  Building wheel for np_utils (setup.py) ... done
  Created wheel for np_utils: filename=np_utils-0.6.0-py3-none-any.whl size=56437 sha256=01170
fe26c7f69fd27b5abd026a1e1e867699f666d34865b69dc07e7fa6e1fdf
  Stored in directory: /root/.cache/pip/wheels/19/0d/33/eaa4dcda5799bcbb51733c0744970d10edb4b9
add4f41beb43
Successfully built np_utils
Installing collected packages: np_utils
Successfully installed np_utils-0.6.0
Shape before one-hot encoding:  (60000,)
Shape after one-hot encoding:  (60000, 10)
```

In [ ]:
```
y_train
```

Out[ ]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]])
```

In [ ]:
```
y_train.shape
```

Out[ ]:
```
(60000, 10)
```

In [ ]:
```
X_train.shape
```

Out[ ]:
```
(60000, 784)
```

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(100, input_dim=784, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(),  # Use CategoricalCrossentropy 
              metrics=['accuracy'])

print(model.summary())
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer us
ing an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	78,500
dense_1 (Dense)	(None, 10)	1,010

Total params: 79,510 (310.59 KB)

Trainable params: 79,510 (310.59 KB)

Non-trainable params: 0 (0.00 B)

None

In [ ]:

```
import tensorflow as tf

model.fit(X_train, y_train, epochs=50, batch_size=32)
```

```
Epoch 1/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8711 - loss: 0.4532
Epoch 2/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9623 - loss: 0.1263
Epoch 3/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9740 - loss: 0.0868
Epoch 4/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9824 - loss: 0.0585
Epoch 5/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9869 - loss: 0.0460
Epoch 6/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9881 - loss: 0.0378
Epoch 7/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9913 - loss: 0.0301
Epoch 8/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9932 - loss: 0.0243
Epoch 9/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9946 - loss: 0.0189
Epoch 10/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9957 - loss: 0.0164
Epoch 11/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9957 - loss: 0.0144
Epoch 12/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9956 - loss: 0.0136
Epoch 13/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9976 - loss: 0.0087
Epoch 14/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9973 - loss: 0.0091
Epoch 15/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9983 - loss: 0.0067
Epoch 16/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9971 - loss: 0.0085
Epoch 17/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9984 - loss: 0.0057
Epoch 18/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9981 - loss: 0.0066
Epoch 19/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9979 - loss: 0.0077
Epoch 20/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9984 - loss: 0.0049
Epoch 21/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9986 - loss: 0.0045
Epoch 22/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9989 - loss: 0.0040
Epoch 23/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9992 - loss: 0.0029
Epoch 24/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9984 - loss: 0.0051
Epoch 25/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9989 - loss: 0.0033
Epoch 26/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9996 - loss: 0.0018
Epoch 27/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9972 - loss: 0.0084
Epoch 28/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9993 - loss: 0.0026
Epoch 29/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9987 - loss: 0.0035
Epoch 30/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9995 - loss: 0.0021
Epoch 31/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9988 - loss: 0.0045
Epoch 32/50
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9988 - loss: 0.0031
Epoch 33/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9980 - loss: 0.0069
Epoch 34/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9989 - loss: 0.0037
Epoch 35/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9976 - loss: 0.0063
Epoch 36/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9995 - loss: 0.0016
Epoch 37/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9988 - loss: 0.0034
Epoch 38/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9993 - loss: 0.0022
Epoch 39/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9994 - loss: 0.0018
Epoch 40/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9987 - loss: 0.0040
Epoch 41/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9993 - loss: 0.0021
Epoch 42/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9990 - loss: 0.0032
Epoch 43/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9994 - loss: 0.0018
Epoch 44/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9991 - loss: 0.0030
Epoch 45/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9983 - loss: 0.0044
Epoch 46/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9997 - loss: 8.6140e-04
Epoch 47/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9987 - loss: 0.0038
Epoch 48/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9987 - loss: 0.0034
Epoch 49/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9996 - loss: 0.0012
Epoch 50/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9990 - loss: 0.0027
```

Out[ ]: <keras.src.callbacks.history.History at 0x78727aaa3890>

In [ ]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9731 - loss: 0.2194
Test accuracy: 0.9779000282287598
```

Approach B

In [ ]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=['accuracy'])
```

```
print(X_train.shape)
print(y_train.shape)
model.fit(X_train, y_train, epochs=50, batch_size=32)

test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')
```

```
(60000, 784)
(60000, 10)
Epoch 1/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8813 - loss: 0.4166
Epoch 2/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9678 - loss: 0.1049
Epoch 3/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9784 - loss: 0.0679
Epoch 4/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9834 - loss: 0.0516
Epoch 5/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9864 - loss: 0.0417
Epoch 6/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9891 - loss: 0.0324
Epoch 7/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9909 - loss: 0.0260
Epoch 8/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9931 - loss: 0.0204
Epoch 9/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9934 - loss: 0.0195
Epoch 10/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9944 - loss: 0.0167
Epoch 11/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9946 - loss: 0.0160
Epoch 12/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9960 - loss: 0.0116
Epoch 13/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9955 - loss: 0.0127
Epoch 14/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9964 - loss: 0.0118
Epoch 15/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9969 - loss: 0.0086
Epoch 16/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9959 - loss: 0.0120
Epoch 17/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9964 - loss: 0.0106
Epoch 18/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9961 - loss: 0.0107
Epoch 19/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9960 - loss: 0.0114
Epoch 20/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9969 - loss: 0.0093
Epoch 21/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9968 - loss: 0.0103
Epoch 22/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9970 - loss: 0.0093
Epoch 23/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9976 - loss: 0.0074
Epoch 24/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9964 - loss: 0.0109
Epoch 25/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9971 - loss: 0.0084
Epoch 26/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9978 - loss: 0.0067
Epoch 27/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9977 - loss: 0.0067
Epoch 28/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9979 - loss: 0.0062
Epoch 29/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9972 - loss: 0.0085
Epoch 30/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9983 - loss: 0.0048
Epoch 31/50
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9973 - loss: 0.0096
Epoch 32/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9981 - loss: 0.0064
Epoch 33/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9972 - loss: 0.0085
Epoch 34/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9986 - loss: 0.0038
Epoch 35/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9976 - loss: 0.0082
Epoch 36/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9986 - loss: 0.0046
Epoch 37/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9976 - loss: 0.0086
Epoch 38/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9983 - loss: 0.0056
Epoch 39/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9983 - loss: 0.0058
Epoch 40/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9985 - loss: 0.0052
Epoch 41/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9989 - loss: 0.0035
Epoch 42/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9977 - loss: 0.0084
Epoch 43/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9986 - loss: 0.0049
Epoch 44/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9981 - loss: 0.0063
Epoch 45/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9983 - loss: 0.0052
Epoch 46/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9993 - loss: 0.0029
Epoch 47/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9978 - loss: 0.0092
Epoch 48/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9981 - loss: 0.0053
Epoch 49/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9990 - loss: 0.0031
Epoch 50/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9981 - loss: 0.0073
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9784 - loss: 0.2102
Test accuracy: 0.9814000129699707
```

Program 2

Design and implement a Convolutional Neural Network (CNN) for classification of image dataset.

Approach A

In [ ]:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

print("Total number of samples=",len(x_train))
print("Total number of class labels=",len(y_train))
print("Total number of samples=",len(x_test))
print("Total number of class labels=",len(y_test))

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history1 = model.fit(x_train, y_train,epochs=5, batch_size=32, validation_data=(x_test, y_tes

test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=32)
print(f"Test Accuracy: {test_acc:.2%}")
print("----------------------------------------END------------------------------------------
```

```
Total number of samples= 50000
Total number of class labels= 50000
Total number of samples= 10000
Total number of class labels= 10000
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserW
arning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential mo
dels, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/5
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.3783 - loss: 1.7001 - val_accuracy:
0.5547 - val_loss: 1.2480
Epoch 2/5
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.5954 - loss: 1.1599 - val_accuracy:
0.6107 - val_loss: 1.1034
Epoch 3/5
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6495 - loss: 1.0070 - val_accuracy:
0.6531 - val_loss: 1.0011
Epoch 4/5
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6810 - loss: 0.9172 - val_accuracy:
0.6790 - val_loss: 0.9459
Epoch 5/5
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7121 - loss: 0.8303 - val_accuracy:
0.6615 - val_loss: 0.9887
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6684 - loss: 0.9751
Test Accuracy: 66.15%
----------------------------------------END---------------------------------------------------
----
```

Approach B

In [ ]:

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

print("---------------------------------------------------------------------------------------
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print("Training data shape=", x_train.shape, ",","Number of training samples=", len(x_train))
print("Training labels shape=", y_train.shape,",","Number of training labels=", len(y_train))
print("Testing data shape=", x_test.shape,",","Number of testing samples=", len(x_test))

print("Testing labels shape=", y_test.shape,",","Number of testing labels=", len(y_test))
print("---------------------------------------------------------------------------------------
print("---------------------------------------------------------------------------------------
print("x_train values before normalization\n", x_train)
print("----------------------------------------------------------------------------------\n\
print("x_test values before normalization\n", x_test)
print("----------------------------------------------------------------------------------\n\
```

```
-------------------------------------------------------------------------------------
Training data shape= (50000, 32, 32, 3) , Number of training samples= 50000
Training labels shape= (50000, 1) , Number of training labels= 50000
Testing data shape= (10000, 32, 32, 3) , Number of testing samples= 10000
Testing labels shape= (10000, 1) , Number of testing labels= 10000
--------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------
x_train values before normalization
 [[[[ 59  62  63]
   [ 43  46  45]
   [ 50  48  43]
   ...
   [158 132 108]
   [152 125 102]
   [148 124 103]]

  [[ 16  20  20]
   [  0   0   0]
   [ 18   8   0]
   ...
   [123  88  55]
   [119  83  50]
   [122  87  57]]

  [[ 25  24  21]
   [ 16   7   0]
   [ 49  27   8]
   ...
   [118  84  50]
   [120  84  50]
   [109  73  42]]

  ...

  [[208 170  96]
   [201 153  34]
   [198 161  26]
   ...
   [160 133  70]
   [ 56  31   7]
   [ 53  34  20]]

  [[180 139  96]
   [173 123  42]
   [186 144  30]
   ...
   [184 148  94]
   [ 97  62  34]
   [ 83  53  34]]

  [[177 144 116]
   [168 129  94]
   [179 142  87]
   ...
   [216 184 140]
   [151 118  84]
   [123  92  72]]]


 [[[154 177 187]
   [126 137 136]
   [105 104  95]
   ...
```

```
 [[ 95 126  78]
  [ 95 123  76]
  [101 128  81]
  ...
  [ 93 124  80]
  [ 95 123  81]
  [ 92 120  80]]]


[[[ 73  78  75]
  [ 98 103 113]
  [ 99 106 114]
  ...
  [135 150 152]
  [135 149 154]
  [203 215 223]]

 [[ 69  73  70]
  [ 84  89  97]
  [ 68  75  81]
  ...
  [ 85  95  89]
  [ 71  82  80]
  [120 133 135]]

 [[ 69  73  70]
  [ 90  95 100]
  [ 62  71  74]
  ...
  [ 74  81  70]
  [ 53  62  54]
  [ 62  74  69]]

 ...

 [[123 128  96]
  [132 132 102]
  [129 128 100]
  ...
  [108 107  88]
  [ 62  60  55]
  [ 27  27  28]]

 [[115 121  91]
  [123 124  95]
  [129 126  99]
  ...
  [115 116  94]
  [ 66  65  59]
  [ 27  27  27]]

 [[116 120  90]
  [121 122  94]
  [129 128 101]
  ...
  [116 115  94]
  [ 68  65  58]
  [ 27  26  26]]]]
----------------------------------------------------------------------------------
```

In [ ]:

```
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
print("---------------------------------------------------------------------------------
print("x_train values after normalization\n", x_train)
print("------------------------------------------------------------------------------\n\
print("x_test values after normalization\n", x_test)
print("------------------------------------------------------------------------------\n\

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship
for i in range(9):
  plt.subplot(3, 3, i + 1)
  plt.imshow(x_train[i])
  plt.title(class_names[y_train[i][0]])
  plt.axis('off')

plt.show()
```

```
---------------------------------------------------------------------------------------
x_train values after normalization
 [[[[0.23137255 0.24313726 0.24705882]
   [0.16862746 0.18039216 0.1764706 ]
   [0.19607843 0.1882353  0.16862746]
   ...
   [0.61960787 0.5176471  0.42352942]
   [0.59607846 0.49019608 0.4       ]
   [0.5803922  0.4862745  0.40392157]]

  [[0.0627451  0.07843138 0.07843138]
   [0.         0.         0.        ]
   [0.07058824 0.03137255 0.        ]
   ...
   [0.48235294 0.34509805 0.21568628]
   [0.46666667 0.3254902  0.19607843]
   [0.47843137 0.34117648 0.22352941]]

  [[0.09803922 0.09411765 0.08235294]
   [0.0627451  0.02745098 0.        ]
   [0.19215687 0.10588235 0.03137255]
   ...
   [0.4627451  0.32941177 0.19607843]
   [0.47058824 0.32941177 0.19607843]
   [0.42745098 0.28627452 0.16470589]]

  ...

  [[0.8156863  0.6666667  0.3764706 ]
   [0.7882353  0.6        0.13333334]
   [0.7764706  0.6313726  0.10196079]
   ...
   [0.627451   0.52156866 0.27450982]
   [0.21960784 0.12156863 0.02745098]
   [0.20784314 0.13333334 0.07843138]]

  [[0.7058824  0.54509807 0.3764706 ]
   [0.6784314  0.48235294 0.16470589]
   [0.7294118  0.5647059  0.11764706]
   ...
   [0.72156864 0.5803922  0.36862746]
   [0.38039216 0.24313726 0.13333334]
   [0.3254902  0.20784314 0.13333334]]

  [[0.69411767 0.5647059  0.45490196]
   [0.65882355 0.5058824  0.36862746]
   [0.7019608  0.5568628  0.34117648]
   ...
   [0.84705883 0.72156864 0.54901963]
   [0.5921569  0.4627451  0.32941177]
   [0.48235294 0.36078432 0.28235295]]]


 [[[0.6039216  0.69411767 0.73333335]
   [0.49411765 0.5372549  0.53333336]
   [0.4117647  0.40784314 0.37254903]
   ...
   [0.35686275 0.37254903 0.2784314 ]
   [0.34117648 0.3529412  0.2784314 ]
   [0.30980393 0.31764707 0.27450982]]

  [[0.54901963 0.627451   0.6627451 ]
   [0.5686275  0.6        0.6039216 ]
```

```
  [0.37254903 0.48235294 0.31764707]
  [0.36078432 0.47058824 0.3137255 ]]]


[[[0.28627452 0.30588236 0.29411766]
  [0.38431373 0.40392157 0.44313726]
  [0.3882353  0.41568628 0.44705883]
  ...
  [0.5294118  0.5882353  0.59607846]
  [0.5294118  0.58431375 0.6039216 ]
  [0.79607844 0.84313726 0.8745098 ]]

 [[0.27058825 0.28627452 0.27450982]
  [0.32941177 0.34901962 0.38039216]
  [0.26666668 0.29411766 0.31764707]
  ...
  [0.33333334 0.37254903 0.34901962]
  [0.2784314  0.32156864 0.3137255 ]
  [0.47058824 0.52156866 0.5294118 ]]

 [[0.27058825 0.28627452 0.27450982]
  [0.3529412  0.37254903 0.39215687]
  [0.24313726 0.2784314  0.2901961 ]
  ...
  [0.2901961  0.31764707 0.27450982]
  [0.20784314 0.24313726 0.21176471]
  [0.24313726 0.2901961  0.27058825]]

 ...

 [[0.48235294 0.5019608  0.3764706 ]
  [0.5176471  0.5176471  0.4       ]
  [0.5058824  0.5019608  0.39215687]
  ...
  [0.42352942 0.41960785 0.34509805]
  [0.24313726 0.23529412 0.21568628]
  [0.10588235 0.10588235 0.10980392]]

 [[0.4509804  0.4745098  0.35686275]
  [0.48235294 0.4862745  0.37254903]
  [0.5058824  0.49411765 0.3882353 ]
  ...
  [0.4509804  0.45490196 0.36862746]
  [0.25882354 0.25490198 0.23137255]
  [0.10588235 0.10588235 0.10588235]]

 [[0.45490196 0.47058824 0.3529412 ]
  [0.4745098  0.47843137 0.36862746]
  [0.5058824  0.5019608  0.39607844]
  ...
  [0.45490196 0.4509804  0.36862746]
  [0.26666668 0.25490198 0.22745098]
  [0.10588235 0.10196079 0.10196079]]]]
```

In [ ]:

```
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)
model = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history1 = model.fit(x_train, y_train,epochs=5, batch_size=256, validation_data=(x_test, y_te

model.summary()
```

```
Epoch 1/5
196/196 ━━━━━━━━━━━━━━━━━━━━ 7s 24ms/step - accuracy: 0.3019 - loss: 1.9158 - val_accuracy: 0.
4767 - val_loss: 1.4781
Epoch 2/5
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.5015 - loss: 1.3923 - val_accuracy: 0.5
121 - val_loss: 1.3584
Epoch 3/5
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.5579 - loss: 1.2520 - val_accuracy: 0.5
746 - val_loss: 1.2005
Epoch 4/5
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.5931 - loss: 1.1528 - val_accuracy: 0.6
096 - val_loss: 1.1202
Epoch 5/5
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.6219 - loss: 1.0841 - val_accuracy: 0.6
246 - val_loss: 1.0712
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_10 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_11 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_11 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_5 (Flatten)	(None, 2304)	0
dense_10 (Dense)	(None, 64)	147,520
dense_11 (Dense)	(None, 10)	650

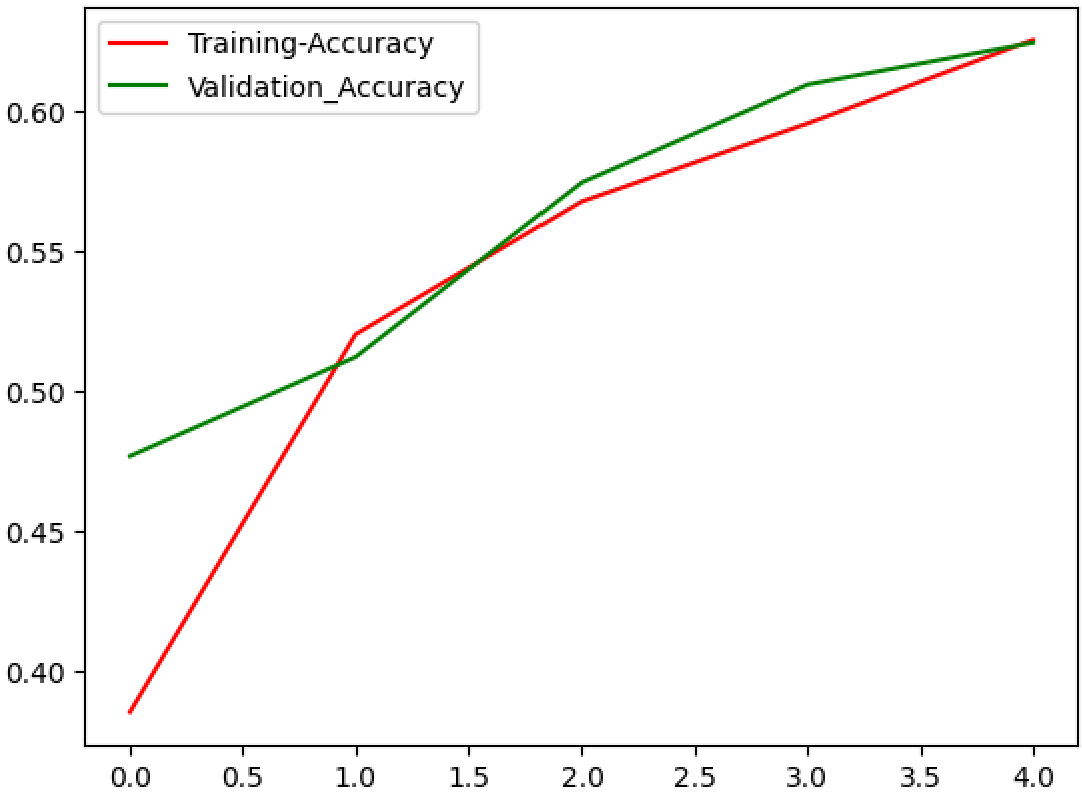
Total params: 502,688 (1.92 MB)

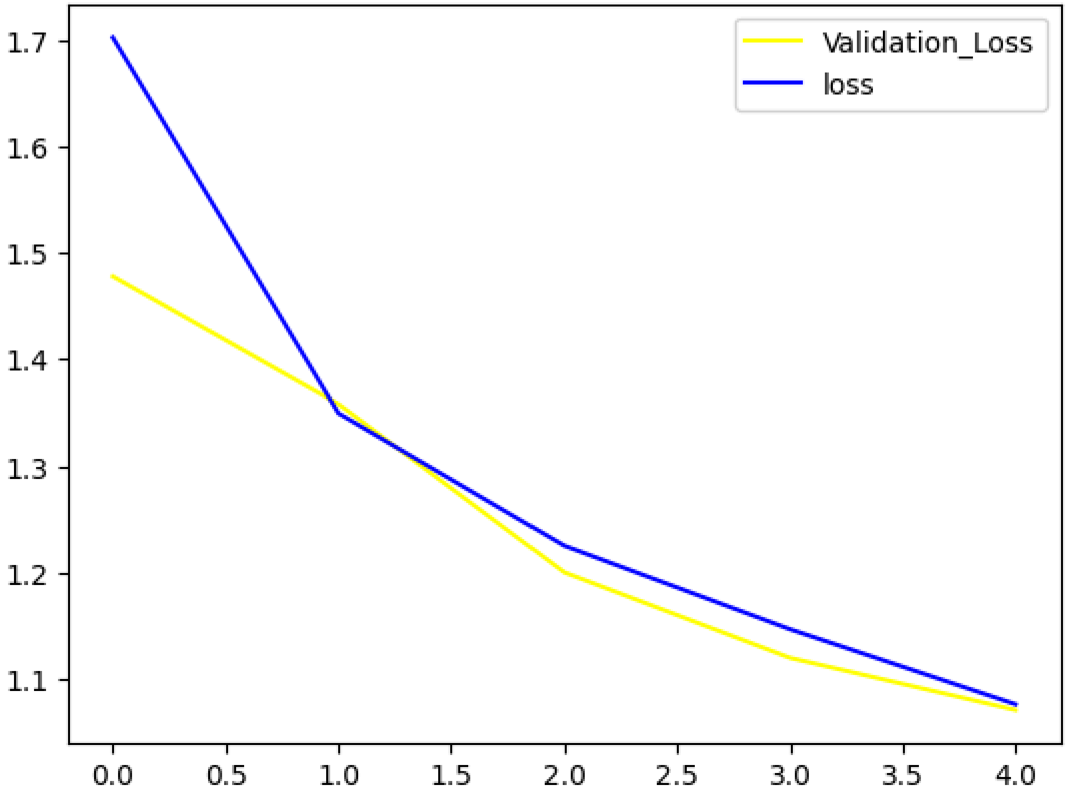
Trainable params: 167,562 (654.54 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 335,126 (1.28 MB)

In [ ]:
```
plt.plot(history1.history['accuracy'], label='Training-Accuracy', color='red')
plt.plot(history1.history['val_accuracy'], label='Validation_Accuracy', color='green')
plt.legend()
plt.show()

plt.plot(history1.history['val_loss'], label='Validation_Loss', color='yellow')
plt.plot(history1.history['loss'], label='loss', color='blue')
plt.legend()
plt.show()
print("------------------------------------------------------------------------------------------
print("x_test=", len(x_test), "y_test=", len(y_test))
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=4)
print(f"Test Accuracy: {test_acc:.2%}")
print("----------------------------------------END-------------------------------------------
```






```
-------------------------------------------------------------------------------------------------
----
x_test= 10000 y_test= 10000
2500/2500 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.6294 - loss: 1.0605
Test Accuracy: 62.46%
----------------------------------------END------------------------------------------------------
----
```

Approach C

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship

for i in range(25):
  plt.subplot(5, 5, i + 1)
  plt.imshow(x_train[i])
  plt.title(class_names[y_train[i][0]])
  plt.axis('off')
plt.show()
```

In [ ]:

```
from tensorflow.keras.callbacks import EarlyStopping
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 0)
model = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
  layers.BatchNormalization(),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.BatchNormalization(),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history1 = model.fit(x_train, y_train,epochs=10, batch_size=256, validation_data=(x_test, y_t

model.summary()
```

```
Epoch 1/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 9s 30ms/step - accuracy: 0.3897 - loss: 1.7775 - val_accuracy: 0.
5408 - val_loss: 1.2738
Epoch 2/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.6383 - loss: 1.0362 - val_accuracy: 0.6
031 - val_loss: 1.1446
Epoch 3/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.7012 - loss: 0.8495 - val_accuracy: 0.6
504 - val_loss: 1.0335
Epoch 4/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.7477 - loss: 0.7215 - val_accuracy: 0.6
472 - val_loss: 1.0640
Epoch 5/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.7805 - loss: 0.6214 - val_accuracy: 0.
6532 - val_loss: 1.0730
Epoch 6/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.8142 - loss: 0.5402 - val_accuracy: 0.
6683 - val_loss: 1.0296
Epoch 7/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8418 - loss: 0.4548 - val_accuracy: 0.6
517 - val_loss: 1.1579
Epoch 8/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8621 - loss: 0.3937 - val_accuracy: 0.6
595 - val_loss: 1.1971
Epoch 9/10
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8912 - loss: 0.3223 - val_accuracy: 0.6
550 - val_loss: 1.2337
```

Model: "sequential_6"

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 30, 30, 32)	896
batch_normalization_2 (BatchNormalization)	(None, 30, 30, 32)	128
max_pooling2d_12 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_13 (Conv2D)	(None, 13, 13, 64)	18,496
batch_normalization_3 (BatchNormalization)	(None, 13, 13, 64)	256
max_pooling2d_13 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_6 (Flatten)	(None, 2304)	0
dense_12 (Dense)	(None, 64)	147,520
dense_13 (Dense)	(None, 10)	650

Total params: 503,456 (1.92 MB)

Trainable params: 167,754 (655.29 KB)

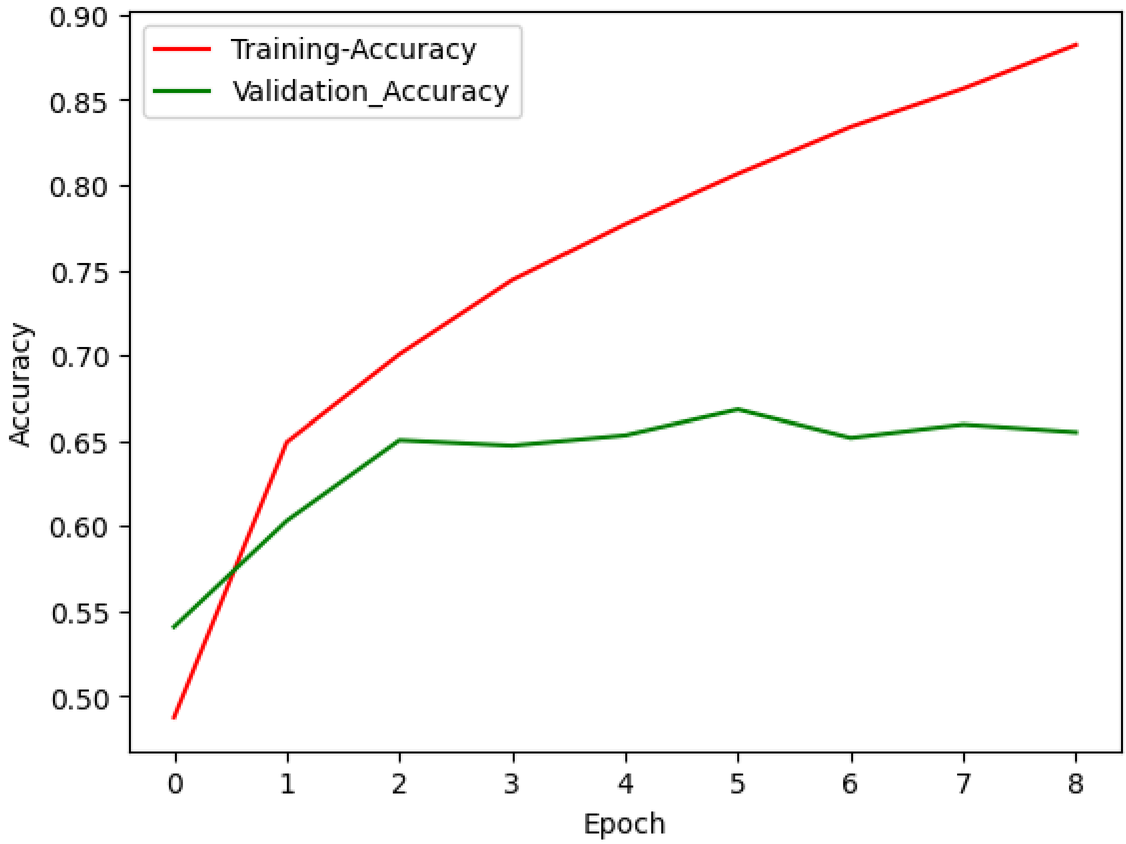
Non-trainable params: 192 (768.00 B)

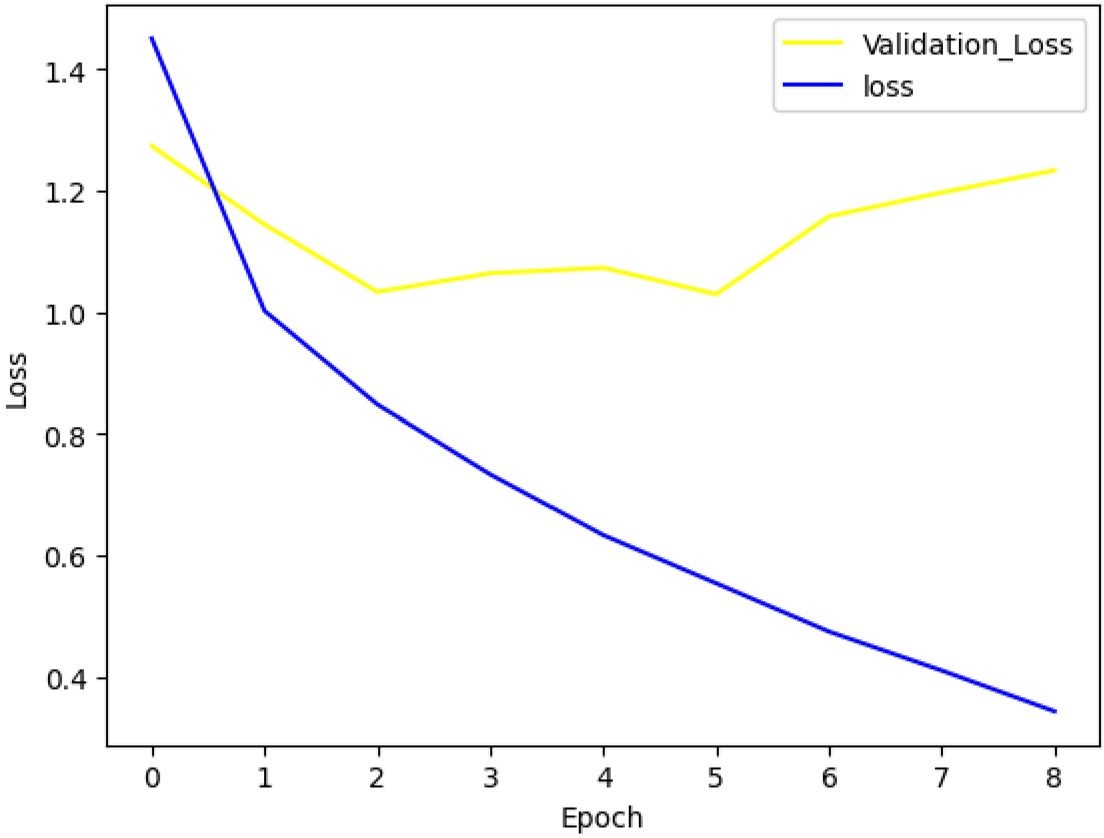
Optimizer params: 335,510 (1.28 MB)

In [ ]:

```
plt.plot(history1.history['accuracy'], label='Training-Accuracy', color='red')
plt.plot(history1.history['val_accuracy'], label='Validation_Accuracy', color='green')
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
```

```
plt.show()

plt.plot(history1.history['val_loss'], label='Validation_Loss', color='yellow')
plt.plot(history1.history['loss'], label='loss', color='blue')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()

print("-----------------------------------------------------------------------------------------
print("x_test=", len(x_test), "y_test=", len(y_test))

test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=4)
print(f"Test Accuracy: {test_acc:.2%}")
print("----------------------------------------END-------------------------------------------
```






```
----------------------------------------------------------------------------------------------
----
x_test= 10000 y_test= 10000
2500/2500 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.6490 - loss: 1.2403
Test Accuracy: 65.50%
---------------------------------------END--------------------------------------------------
----
```

In [ ]:
```python
import numpy as np
sample_index = 3
sample_image = np.expand_dims(x_test[sample_index], axis=0)
prediction = model.predict(sample_image)
predicted_class = np.argmax(prediction)
cifar10_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's
print(f"Predicted class: {cifar10_labels[predicted_class]}")
```

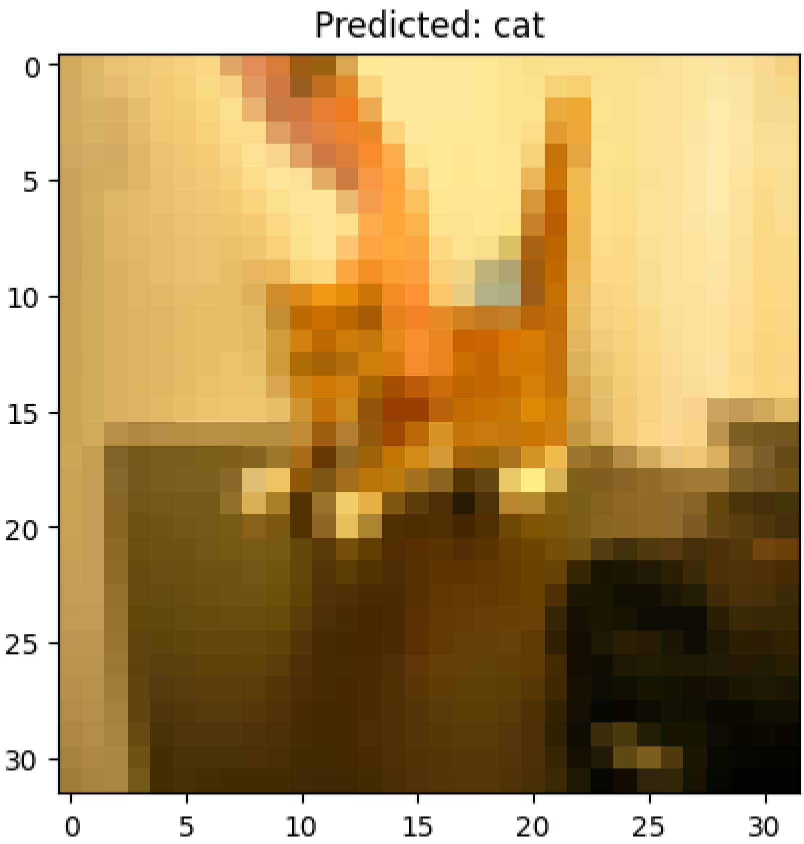
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 413ms/step
Predicted class: airplane
```

In [ ]:
```python
from tensorflow.keras.preprocessing import image
import numpy as np
from PIL import Image
img_path = '/content/cat_0001.jpg'
img = Image.open(img_path).resize((32, 32)).convert('RGB')
img_array = np.array(img).astype('float32') / 255.0
img_array = np.expand_dims(img_array, axis=0)

prediction = model.predict(img_array)
predicted_class = np.argmax(prediction)
print(f"Predicted class: {cifar10_labels[predicted_class]}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 411ms/step
Predicted class: cat
```

In [ ]:
```python
import matplotlib.pyplot as plt
plt.imshow(img)
plt.title(f"Predicted: {cifar10_labels[predicted_class]}")
plt.show()
```

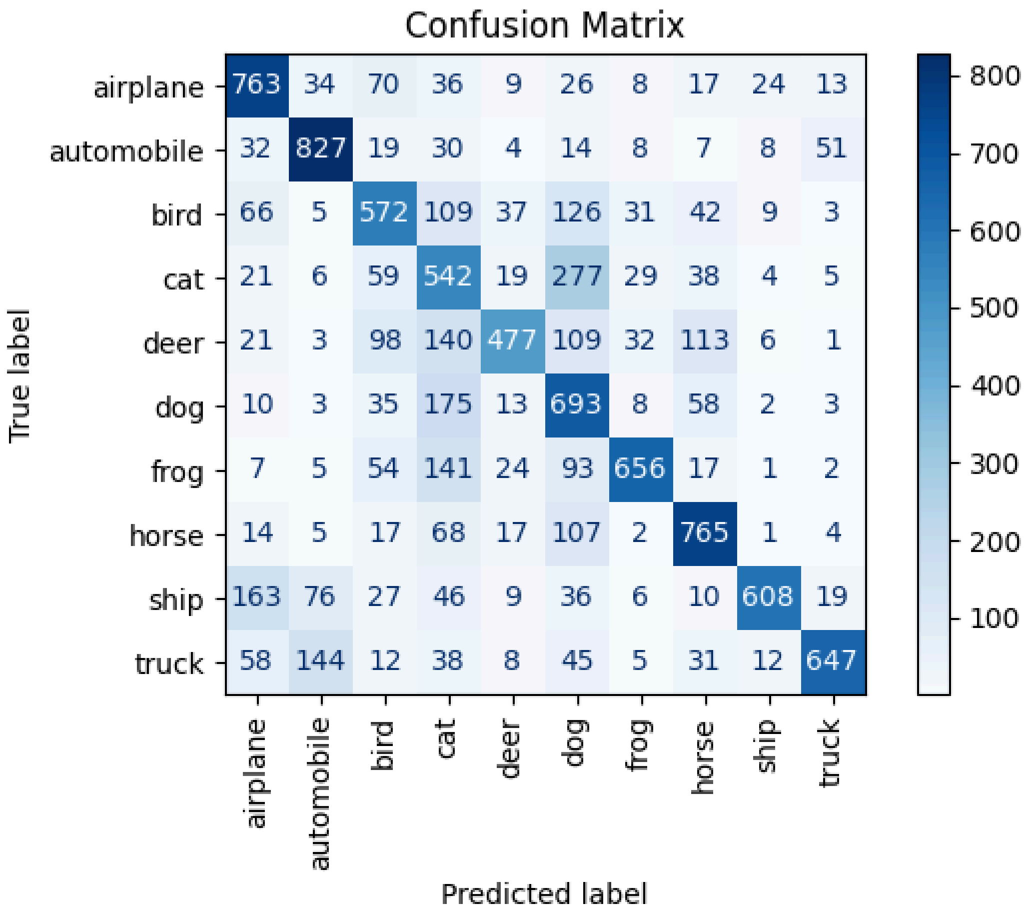
```
import pandas as pd
y_pred_probs = model.predict(x_test)
y_pred_classes = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)
df = pd.DataFrame({
  "Actual Class": [class_names[i] for i in y_true],
  "Predicted Class": [class_names[i] for i in y_pred_classes]
})

misclassified = df[df["Actual Class"] != df["Predicted Class"]]
correctly_classified = df[df["Actual Class"] == df["Predicted Class"]]

print("Number of Misclassified Samples:",len(misclassified))
print(misclassified)
print("Number of Correctly Classified Samples:",len(misclassified))
print(correctly_classified)

print("Total samples:", len(y_test))
print("Misclassified samples:", len(misclassified))
```



```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Number of Misclassified Samples: 3450
     Actual Class Predicted Class
1            ship      automobile
2            ship           horse
4            frog            deer
5            frog             dog
10       airplane             dog
...           ...             ...
9986         ship        airplane
9989         bird            deer
9993          dog             cat
9995         ship             cat
9996          cat             dog

[3450 rows x 2 columns]
Number of Correctly Classified Samples: 3450
     Actual Class Predicted Class
0             cat             cat
3        airplane        airplane
6      automobile      automobile
7            frog            frog
8             cat             cat
...           ...             ...
9992          cat             cat
9994          cat             cat
9997          dog             dog
9998   automobile      automobile
9999        horse           horse

[6550 rows x 2 columns]
Total samples: 10000
Misclassified samples: 3450
```

In [ ]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true, y_pred_classes)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=class_names)
disp.plot(cmap=plt.cm.Blues, xticks_rotation='vertical')
plt.title("Confusion Matrix")
plt.tight_layout()
plt.show()
```



In [ ]:

```python
from sklearn.metrics import classification_report
report=classification_report(y_true,y_pred_classes,target_names=class_names)
print(report)
```

```
              precision    recall  f1-score   support

    airplane       0.66      0.76      0.71      1000
  automobile       0.75      0.83      0.78      1000
        bird       0.59      0.57      0.58      1000
         cat       0.41      0.54      0.47      1000
        deer       0.77      0.48      0.59      1000
         dog       0.45      0.69      0.55      1000
        frog       0.84      0.66      0.74      1000
       horse       0.70      0.77      0.73      1000
        ship       0.90      0.61      0.73      1000
       truck       0.86      0.65      0.74      1000

    accuracy                           0.66     10000
   macro avg       0.69      0.66      0.66     10000
weighted avg       0.69      0.66      0.66     10000
```

Program 3

Write a program to enable pre-train models to classify a given image dataset.

In [ ]:
```
!pip install tensorflow
```

In [ ]:
```
!unzip "/content/drive/MyDrive/cats-dogs-dataset.zip" -d /content/
```

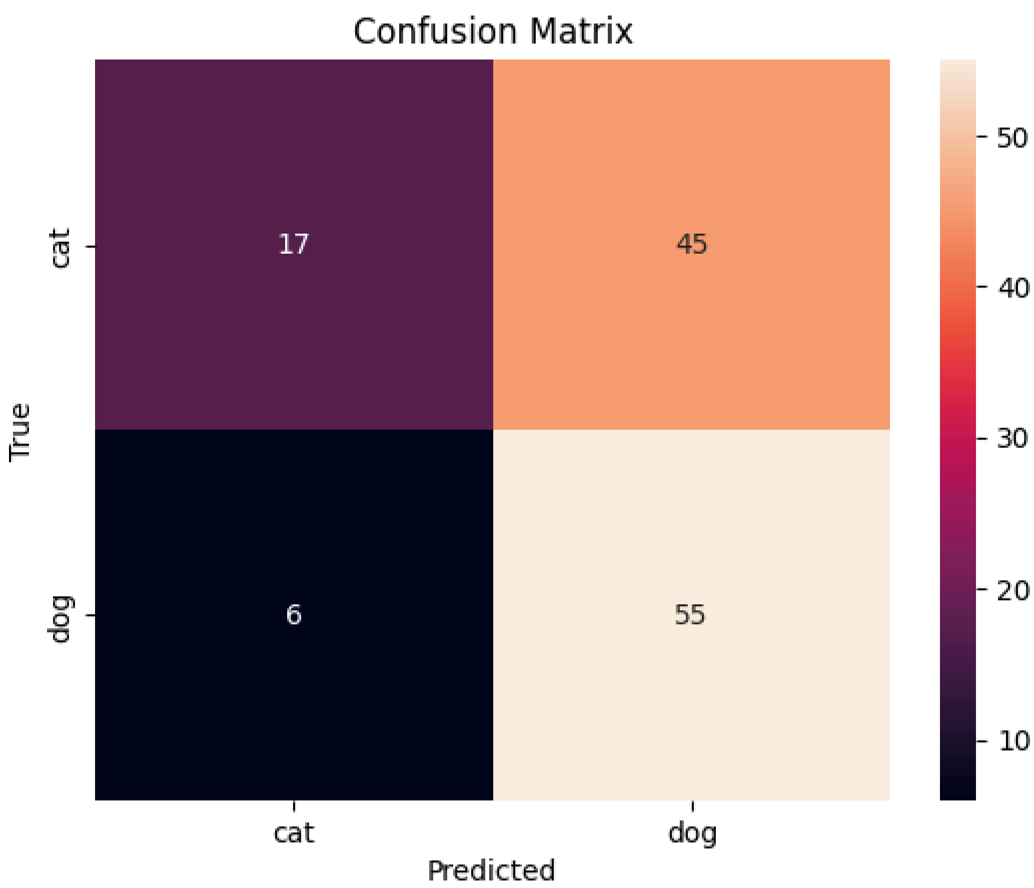
Approach A (VGG16)

In [ ]:
```
import os
import json
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import cv2
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

train_dataset_path = "/content/cats-dogs-dataset/Training"
test_dataset_path = "/content/cats-dogs-dataset/Testing"
IMG_SIZE = 224
Model_save_path = "vgg16.h5"

def get_label_from_filename(filename):
    return filename.lower().split("_")[0]

def load_data(dataset_path):
    images, labels = [], []
    class_names = set()

    for file in tqdm(os.listdir(dataset_path)):
        if file.endswith((".jpg", ".png", ".jpeg")):
            path = os.path.join(dataset_path, file)
            img = cv2.imread(path)

            if img is None:
                print(f"Warning: Unable to read image: {file}")
                continue

            img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
            img = preprocess_input(img)

            label = get_label_from_filename(file)
            class_names.add(label)

            images.append(img)
            labels.append(label)

    class_names = sorted(list(class_names))
```

```python
    label_to_index = {name: idx for idx, name in enumerate(class_names)}

    labels = np.array([label_to_index[label] for label in labels])
    images = np.array(images)

    return images, labels, class_names, label_to_index

X_train_full, y_train_full, class_names, label_to_index = load_data(train_dataset_path)
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full, test_size=0.15,

y_train = tf.keras.utils.to_categorical(y_train, num_classes=len(class_names))
y_val = tf.keras.utils.to_categorical(y_val, num_classes=len(class_names))

X_test, y_test, _, _ = load_data(test_dataset_path)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=len(class_names))

base_model = VGG16(input_shape=(IMG_SIZE, IMG_SIZE, 3), include_top=False, weights='imagenet'
base_model.trainable = True

inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = base_model(inputs)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation='relu', kernel_regularizer=l2(0.001))(x)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(len(class_names), activation='softmax')(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer=Adam(learning_rate=0.0007), loss='categorical_crossentropy', metrics=

history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=5, batch_size=32

test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {test_acc:.4f}")

model.save(Model_save_path)
print(f"Model saved to {Model_save_path}")

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

print(classification_report(y_true, y_pred_classes, target_names=class_names))

cm = confusion_matrix(y_true, y_pred_classes)
sns.heatmap(cm, annot=True, fmt='d', xticklabels=class_names, yticklabels=class_names)
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

```
100%|██████████| 479/479 [00:01<00:00, 253.04it/s]
100%|██████████| 123/123 [00:00<00:00, 335.64it/s]
```

```
Epoch 1/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 65s 5s/step - accuracy: 0.5092 - loss: 7.1317 - val_accuracy: 0.444
4 - val_loss: 1.0177
Epoch 2/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 7s 503ms/step - accuracy: 0.5367 - loss: 1.0110 - val_accuracy: 0.4
306 - val_loss: 1.0092
Epoch 3/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 7s 530ms/step - accuracy: 0.4867 - loss: 1.0278 - val_accuracy: 0.4
306 - val_loss: 0.9833
Epoch 4/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 7s 534ms/step - accuracy: 0.5355 - loss: 0.9793 - val_accuracy: 0.4
306 - val_loss: 0.9665
Epoch 5/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 7s 517ms/step - accuracy: 0.5220 - loss: 0.9592 - val_accuracy: 0.6
806 - val_loss: 0.9387
4/4 ━━━━━━━━━━━━━━━━━━━━ 13s 4s/step - accuracy: 0.6102 - loss: 0.9403
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.sav
e_model(model)`. This file format is considered legacy. We recommend using instead the native
Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_mode
l.keras')`.
Test Accuracy: 0.5854
Model saved to vgg16.h5
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 456ms/step
              precision    recall  f1-score   support

         cat       0.74      0.27      0.40        62
         dog       0.55      0.90      0.68        61

    accuracy                           0.59       123
   macro avg       0.64      0.59      0.54       123
weighted avg       0.65      0.59      0.54       123
```



Approach B (ResNet50)

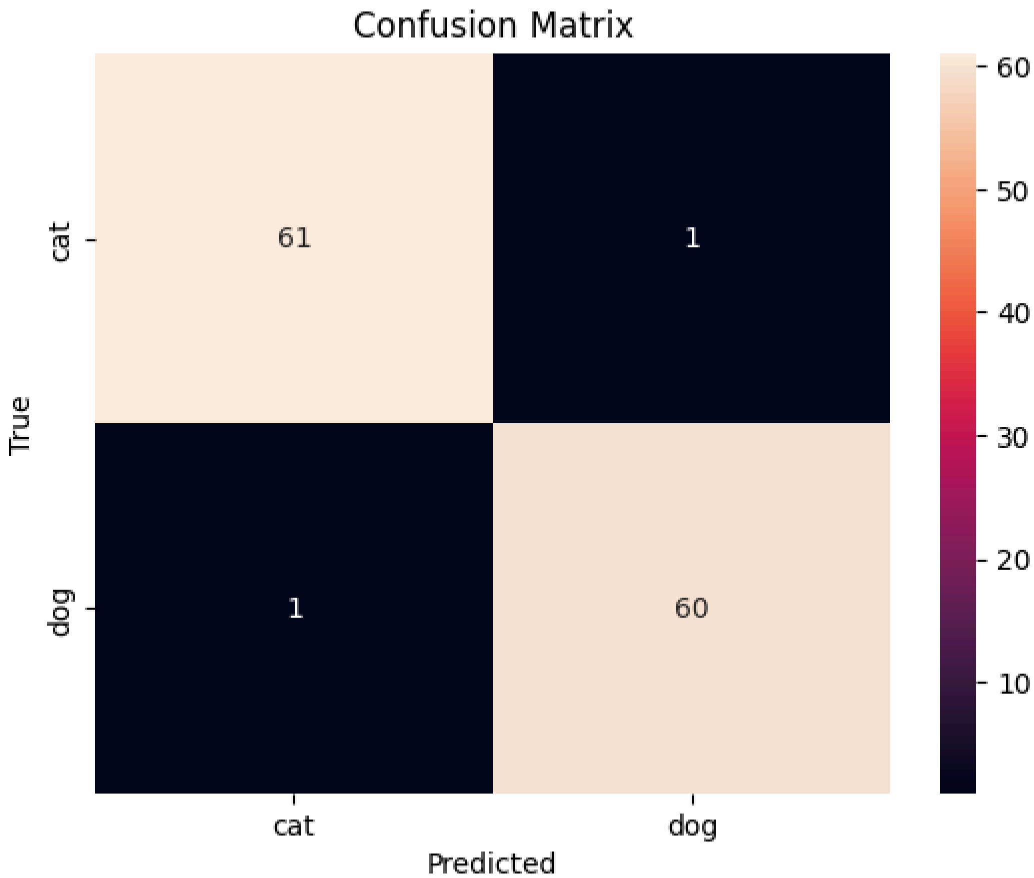
In [ ]:

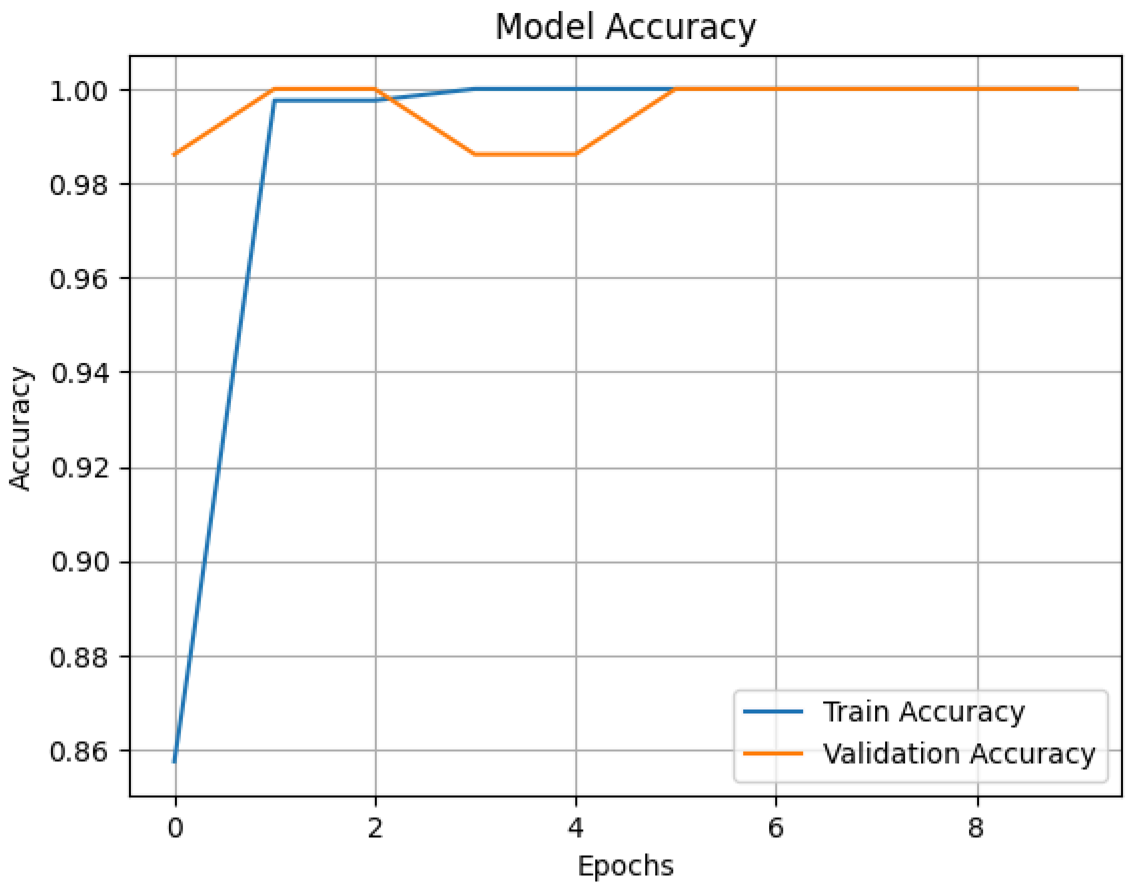
```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import cv2
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

train_dataset_path = "/content/cats-dogs-dataset/Training"
test_dataset_path = "/content/cats-dogs-dataset/Testing"
IMG_SIZE = 224
Model_save_path = "resnet50_all_layers.h5"

def get_label_from_filename(filename):
    return filename.lower().split("_")[0]

def load_data(dataset_path):
    images, labels = [], []
    class_names = set()
    for file in tqdm(os.listdir(dataset_path)):
        if file.endswith((".jpg", ".png", ".jpeg")):
            path = os.path.join(dataset_path, file)
            img = cv2.imread(path)
            if img is None:
                print(f"Warning: Unable to read image: {file}")
                continue
            img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
            img = preprocess_input(img)
            label = get_label_from_filename(file)
            class_names.add(label)
            images.append(img)
            labels.append(label)
    class_names = sorted(list(class_names))
    label_to_index = {name: idx for idx, name in enumerate(class_names)}
    labels = np.array([label_to_index[label] for label in labels])
    images = np.array(images)
    return images, labels, class_names, label_to_index

X_train_full, y_train_full, class_names, label_to_index = load_data(train_dataset_path)

X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full, test_size=0.15,
y_train = tf.keras.utils.to_categorical(y_train, num_classes=len(class_names))
y_val = tf.keras.utils.to_categorical(y_val, num_classes=len(class_names))
X_test, y_test, _, _ = load_data(test_dataset_path)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=len(class_names))

base_model = ResNet50(input_shape=(IMG_SIZE, IMG_SIZE, 3), include_top=False, weights='imagen
base_model.trainable = True

inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = base_model(inputs, training=True)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation='relu', kernel_regularizer=l2(0.001))(x)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(len(class_names), activation='softmax')(x)
```

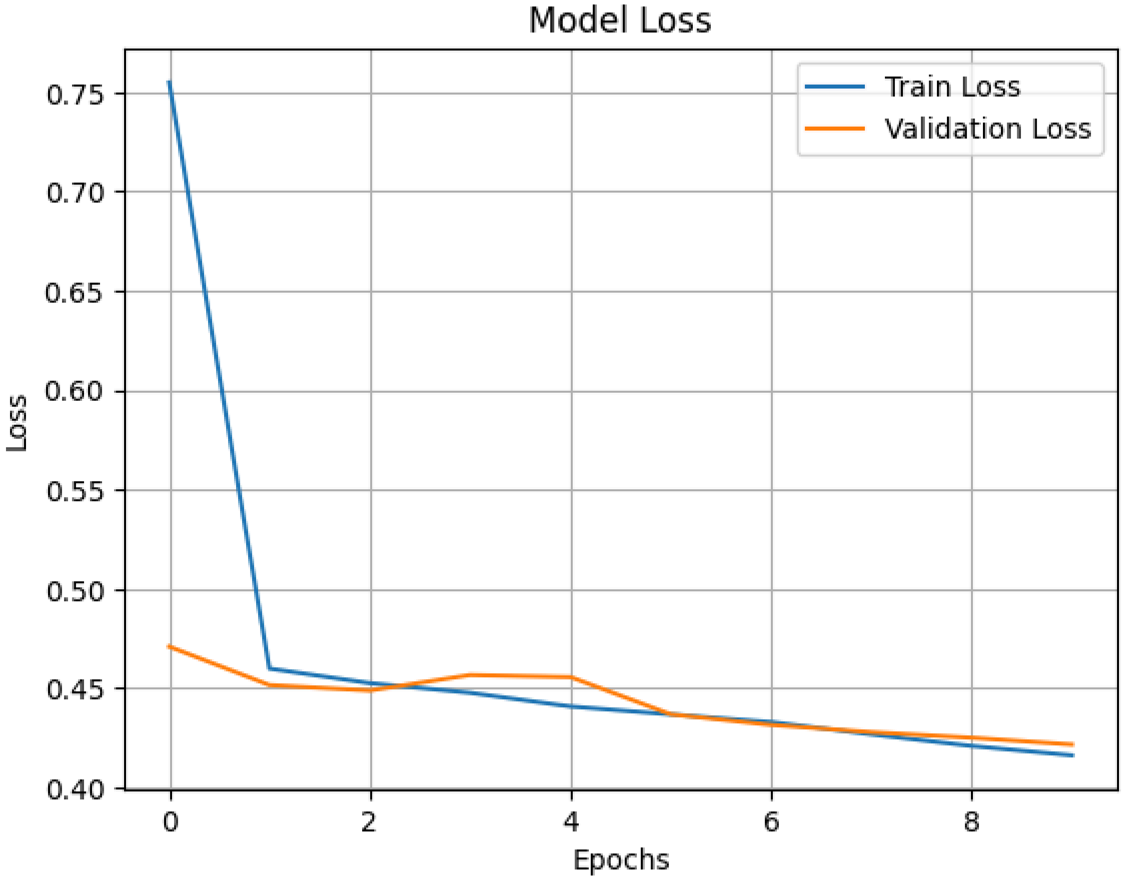
```python
model = keras.Model(inputs, outputs)

model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy', metrics=
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, batch_size=3

test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {test_acc:.4f}")

model.save(Model_save_path)
print(f"Model saved to {Model_save_path}")

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

print(classification_report(y_true, y_pred_classes, target_names=class_names))

cm = confusion_matrix(y_true, y_pred_classes)
sns.heatmap(cm, annot=True, fmt='d', xticklabels=class_names, yticklabels=class_names)
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title("Model Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()

plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title("Model Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```

```
100%|██████████| 479/479 [00:01<00:00, 432.23it/s]
100%|██████████| 123/123 [00:00<00:00, 244.34it/s]
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resn
et50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Epoch 1/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 110s 4s/step - accuracy: 0.7633 - loss: 0.9342 - val_accuracy: 0.98
61 - val_loss: 0.4710
Epoch 2/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 9s 327ms/step - accuracy: 0.9968 - loss: 0.4623 - val_accuracy: 1.0
000 - val_loss: 0.4516
Epoch 3/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 331ms/step - accuracy: 0.9988 - loss: 0.4517 - val_accuracy: 1.0
000 - val_loss: 0.4489
Epoch 4/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 317ms/step - accuracy: 1.0000 - loss: 0.4476 - val_accuracy: 0.9
861 - val_loss: 0.4567
Epoch 5/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 334ms/step - accuracy: 1.0000 - loss: 0.4411 - val_accuracy: 0.9
861 - val_loss: 0.4556
Epoch 6/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 326ms/step - accuracy: 1.0000 - loss: 0.4384 - val_accuracy: 1.0
000 - val_loss: 0.4369
Epoch 7/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 320ms/step - accuracy: 1.0000 - loss: 0.4341 - val_accuracy: 1.0
000 - val_loss: 0.4316
Epoch 8/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 316ms/step - accuracy: 1.0000 - loss: 0.4276 - val_accuracy: 1.0
000 - val_loss: 0.4279
Epoch 9/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 308ms/step - accuracy: 1.0000 - loss: 0.4220 - val_accuracy: 1.0
000 - val_loss: 0.4251
Epoch 10/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 309ms/step - accuracy: 1.0000 - loss: 0.4173 - val_accuracy: 1.0
000 - val_loss: 0.4217
4/4 ━━━━━━━━━━━━━━━━━━━━ 5s 2s/step - accuracy: 0.9800 - loss: 0.4678
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.sav
e_model(model)`. This file format is considered legacy. We recommend using instead the native
Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_mode
l.keras')`.
Test Accuracy: 0.9837
Model saved to resnet50_all_layers.h5
4/4 ━━━━━━━━━━━━━━━━━━━━ 9s 2s/step
              precision    recall  f1-score   support

         cat       0.98      0.98      0.98        62
         dog       0.98      0.98      0.98        61

    accuracy                           0.98       123
   macro avg       0.98      0.98      0.98       123
weighted avg       0.98      0.98      0.98       123
```


Confusion Matrix
True
cat
dog
Predicted
61
1
1
60
60
50
40
30
20
10


Model Accuracy
Accuracy
Epochs
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
2
4
6
8
Train Accuracy
Validation Accuracy


Model Loss
Train Loss
Validation Loss
Loss
Epochs
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
2
4
6
8

Program 4

Design and implement a neural based network for generating word embedding for words in a document corpus.

In [ ]:
```
import gensim
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess
```

In [ ]:
```
corpus=[
    "Deep learning is a core subject of artificial intelligence",
    "Machine learning is a subbranch of deep learning",
    "Convolutional Neural Network (CNN) is a basic deep neural network in deep learning",
    "Alex and Visual Geometry Group (VGG) neural networks are pre trained deep neural network
    "Deep residual network is used in image recognition"
]
```

In [ ]:
```
tokenized_corpus=[simple_preprocess(line) for line in corpus]
```

In [ ]:
```
print(tokenized_corpus)
```

```
[['deep', 'learning', 'is', 'core', 'subject', 'of', 'artificial', 'intelligence'], ['machin
e', 'learning', 'is', 'subbranch', 'of', 'deep', 'learning'], ['convolutional', 'neural', 'net
work', 'cnn', 'is', 'basic', 'deep', 'neural', 'network', 'in', 'deep', 'learning'], ['alex',
'and', 'visual', 'geometry', 'group', 'vgg', 'neural', 'networks', 'are', 'pre', 'trained', 'd
eep', 'neural', 'networks'], ['deep', 'residual', 'network', 'is', 'used', 'in', 'image', 'rec
ognition']]
```

In [ ]:
```
model = Word2Vec(
    sentences=tokenized_corpus,
    vector_size=300,
    window=3,
    min_count=1,
    sg=1,
    epochs=100
)
```

In [ ]:
```
words = [word for sentence in tokenized_corpus for word in sentence]
vocab = set(words)
word2idx = {word: idx for idx, word in enumerate(vocab)}
idx2word = {idx: word for word, idx in word2idx.items()}
vocab_size = len(vocab)
```

In [ ]:
```
print(words)
```

```
['deep', 'learning', 'is', 'core', 'subject', 'of', 'artificial', 'intelligence', 'machine',
'learning', 'is', 'subbranch', 'of', 'deep', 'learning', 'convolutional', 'neural', 'network',
'cnn', 'is', 'basic', 'deep', 'neural', 'network', 'in', 'deep', 'learning', 'alex', 'and', 'v
isual', 'geometry', 'group', 'vgg', 'neural', 'networks', 'are', 'pre', 'trained', 'deep', 'ne
ural', 'networks', 'deep', 'residual', 'network', 'is', 'used', 'in', 'image', 'recognition']
```

In [ ]:
```
print(vocab)
```

```
{'cnn', 'group', 'used', 'neural', 'learning', 'core', 'subject', 'machine', 'recognition', 'n
etworks', 'pre', 'trained', 'network', 'image', 'intelligence', 'geometry', 'residual', 'subbr
anch', 'are', 'basic', 'vgg', 'in', 'and', 'of', 'alex', 'is', 'deep', 'artificial', 'convolut
ional', 'visual'}
```

In [ ]:
```
print(vocab_size)
```

30

In [ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class WordEmbeddingModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(WordEmbeddingModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, vocab_size)

    def forward(self, inputs):
      embeds = self.embeddings(inputs)
      output = self.linear(embeds)
      return output

vector_size = 300
model = WordEmbeddingModel(vocab_size, vector_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

training_data = []
for sentence in tokenized_corpus:
    for i, target_word in enumerate(sentence):
        target_idx = word2idx[target_word]
        context_idx = word2idx[target_word]
        training_data.append((context_idx, target_idx))

epochs = 100
for epoch in range(epochs):
    total_loss = 0
    for context_idx, target_idx in training_data:
        context_tensor = torch.LongTensor([context_idx])
        target_tensor = torch.LongTensor([target_idx])

        outputs = model(context_tensor)
        loss = criterion(outputs, target_tensor)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        if (epoch + 1) % 10 == 0:
          print(f'Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(training_data):.4f}')

word_embeddings = model.embeddings.weight.data

print("\nWord embeddings:")
for word in ["deep", "learning", "intelligence", "network"]:
    if word in word2idx:
        idx = word2idx[word]
        print(f"{word}: {word_embeddings[idx].numpy()}")
```

```
Epoch [10/100], Loss: 0.0001
Epoch [10/100], Loss: 0.0004
Epoch [10/100], Loss: 0.0006
Epoch [10/100], Loss: 0.0017
Epoch [10/100], Loss: 0.0027
Epoch [10/100], Loss: 0.0031
Epoch [10/100], Loss: 0.0043
Epoch [10/100], Loss: 0.0050
Epoch [10/100], Loss: 0.0060
Epoch [10/100], Loss: 0.0063
Epoch [10/100], Loss: 0.0065
Epoch [10/100], Loss: 0.0074
Epoch [10/100], Loss: 0.0078
Epoch [10/100], Loss: 0.0079
Epoch [10/100], Loss: 0.0081
Epoch [10/100], Loss: 0.0092
Epoch [10/100], Loss: 0.0095
Epoch [10/100], Loss: 0.0098
Epoch [10/100], Loss: 0.0107
Epoch [10/100], Loss: 0.0110
Epoch [10/100], Loss: 0.0118
Epoch [10/100], Loss: 0.0119
Epoch [10/100], Loss: 0.0122
Epoch [10/100], Loss: 0.0125
Epoch [10/100], Loss: 0.0131
Epoch [10/100], Loss: 0.0132
Epoch [10/100], Loss: 0.0134
Epoch [10/100], Loss: 0.0141
Epoch [10/100], Loss: 0.0151
Epoch [10/100], Loss: 0.0158
Epoch [10/100], Loss: 0.0167
Epoch [10/100], Loss: 0.0175
Epoch [10/100], Loss: 0.0182
Epoch [10/100], Loss: 0.0185
Epoch [10/100], Loss: 0.0190
Epoch [10/100], Loss: 0.0199
Epoch [10/100], Loss: 0.0207
Epoch [10/100], Loss: 0.0214
Epoch [10/100], Loss: 0.0215
Epoch [10/100], Loss: 0.0217
Epoch [10/100], Loss: 0.0222
Epoch [10/100], Loss: 0.0224
Epoch [10/100], Loss: 0.0232
Epoch [10/100], Loss: 0.0235
Epoch [10/100], Loss: 0.0237
Epoch [10/100], Loss: 0.0244
Epoch [10/100], Loss: 0.0249
Epoch [10/100], Loss: 0.0258
Epoch [10/100], Loss: 0.0266
Epoch [20/100], Loss: 0.0000
Epoch [20/100], Loss: 0.0001
Epoch [20/100], Loss: 0.0002
Epoch [20/100], Loss: 0.0005
Epoch [20/100], Loss: 0.0008
Epoch [20/100], Loss: 0.0009
Epoch [20/100], Loss: 0.0012
Epoch [20/100], Loss: 0.0014
Epoch [20/100], Loss: 0.0017
Epoch [20/100], Loss: 0.0018
Epoch [20/100], Loss: 0.0018
Epoch [20/100], Loss: 0.0021
Epoch [20/100], Loss: 0.0022
Epoch [20/100], Loss: 0.0022
```

```
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0001
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0003
Epoch [100/100], Loss: 0.0003

Word embeddings:
deep: [ 1.3423235e+00 -1.6635489e+00  5.3646600e-01  1.3441798e-01
  5.8005172e-01 -8.8931817e-01 -1.2043906e+00 -8.7312007e-01
  4.3609205e-01  2.8637969e+00 -1.8959600e+00 -6.0942400e-01
 -1.8332468e+00  6.3445795e-01  2.0037314e-01 -1.2093500e+00
 -1.8195824e-01 -1.5730157e+00 -3.3391303e-01 -8.6618954e-01
 -9.8306745e-01  2.9520690e-01 -9.7716373e-01  5.2517664e-01
  7.8061759e-01 -3.9278874e-01 -4.4984993e-01 -8.8663232e-01
 -1.2577633e+00  1.7154276e+00 -3.4585401e-01  4.8501971e-01
  6.1871046e-01  1.9889870e+00 -1.2716299e+00 -9.0922213e-01
  1.3801308e+00 -9.1471724e-02 -2.0419068e+00 -7.7250510e-02
  4.0693030e-01 -9.4640964e-01  5.8541145e-02 -1.3435839e-01
  1.0893608e+00  2.4196583e-01 -5.6501174e-01 -1.3717782e+00
```

```
-1.3320101e+00  1.6115509e-02 -3.0457169e-01 -1.0566860e+00
 3.2094005e-01  1.6335626e+00  1.0661012e-01  2.2538546e-02
 6.4644814e-01  3.9380679e-01 -2.3495425e-01  4.7806641e-01
 2.6221395e+00  4.4673780e-01  2.1371830e+00 -3.8986942e-01
-3.2924646e-01 -4.8854294e-01  1.6110978e+00 -1.2838587e+00
-6.3956714e-01  6.7272669e-01  9.1743857e-01  5.5603951e-01
-9.9963528e-01  5.6133145e-01 -1.3249118e+00 -4.9047628e-01
-6.0997522e-01 -6.9017076e-01 -6.2951750e-01 -2.4087927e-01
-8.9429194e-01 -2.0854900e+00 -4.1197538e-01  9.3403739e-01
 5.5684328e-01  6.5473604e-01 -8.3409238e-01 -1.6668615e+00
 1.5381185e+00  1.1241339e+00 -1.4346555e+00  8.7984312e-01
 1.6619115e+00  1.7267632e+00  1.0092824e+00  1.1214318e+00
 1.9062141e-01 -3.8336083e-02  1.7225593e-01  7.0982349e-01
 2.1853539e-01 -3.3429071e-01 -8.6617106e-01 -2.5098005e-01
 1.7366718e+00 -1.9316987e+00 -9.1037834e-01  4.5361432e-01
-1.0421102e+00 -3.2354558e+00 -1.7593424e+00  1.5082923e-01
 9.5707983e-01  3.0627129e+00 -2.1190351e-01  8.8259298e-01
-9.1440278e-01  7.4756992e-01  8.2616735e-01  1.1236484e+00
 7.5301111e-01  2.6006106e-01  3.2380110e-01  3.6135498e-01
-2.7835846e-01  4.4001499e-01  7.4737132e-01  7.2155201e-01
 4.4764882e-01  1.2279739e+00  4.1343746e-01 -1.3416208e+00
 1.8850256e+00 -1.8431094e-01  2.1500177e+00 -7.3268592e-01
-1.5052283e+00 -7.8039306e-01  8.5989863e-01 -2.1354134e+00
 1.5192552e+00 -1.2617983e-01 -6.3077039e-01  8.9242351e-01
 2.1067643e+00  2.9995582e-01 -2.8124356e+00  5.3590190e-01
-4.4888136e-01 -2.8597149e-01  8.4044027e-01 -7.8398323e-01
 3.3575273e-01  1.6974774e-01  1.2699273e-01  6.1213571e-01
 9.0776145e-01  1.2591513e+00  1.8934529e+00  1.6806829e-01
-2.0420752e+00  8.8439995e-01 -7.6177269e-02  7.3890626e-01
-5.7005209e-01 -1.3197244e+00 -1.5885746e+00  1.8818369e-03
-1.8079129e+00 -1.1625034e+00 -3.4523484e-01  6.2580615e-01
-2.4134365e-01  1.2021214e+00 -7.3855418e-01  7.8885424e-01
 6.5004492e-01  2.3181970e+00 -6.7011511e-01  2.2237062e-02
 7.1645206e-01  1.5583168e+00  2.5746610e+00 -3.9158660e-01
 4.7005782e-01  5.8255935e-01  9.3002576e-01  7.1662444e-01
-1.0751835e+00 -1.1744523e+00 -6.9425607e-01  1.3107783e+00
 1.9637581e+00  1.1098621e+00  2.1717303e-01  9.5646769e-01
 1.1384602e+00 -1.9343712e+00 -9.7663963e-01 -3.9637727e-01
 7.4350202e-01 -1.7789813e-02  8.7859869e-01  1.4099795e+00
-1.0322231e+00 -1.8515648e-01 -1.1882948e+00  4.9275836e-01
 4.7488925e-01 -9.3912852e-01 -7.8542769e-01 -1.0889887e+00
-1.7218567e-01 -6.9630343e-01  3.1505796e-01 -4.4438934e-01
-1.0897197e+00  4.5582452e-01  5.1147050e-01 -6.6672122e-01
 5.6387144e-01 -6.7738372e-01 -1.4456521e+00 -2.3623291e-01
 1.1100490e+00  5.0761515e-01 -9.4933516e-01  1.2381195e+00
 3.8143858e-01  4.6194407e-01 -2.0989172e+00 -1.3249296e+00
 1.2646431e+00  5.7137448e-01 -3.3446807e-01  1.2710676e+00
 1.2063743e+00  8.8360870e-01  8.1841731e-01  8.4204549e-01
-1.3560604e+00  1.1208359e+00  1.3169919e+00 -3.9656532e-01
 7.7376819e-01 -1.0926485e+00  6.3057966e-03  1.7159562e-01
-4.6623805e-01 -9.0612358e-01 -3.9200288e-01 -4.6882305e-01
 8.2163000e-01  2.6864583e+00 -1.1575481e+00 -6.9867718e-01
-2.9237700e-01 -1.3905851e+00  2.7211329e-02 -1.2834181e-01
-4.3848714e-01 -1.6092318e+00  8.2917535e-01  6.1843753e-01
 1.0258918e+00 -1.3526669e+00  1.5865659e+00 -2.3166761e-03
-3.1572962e-01  9.3539381e-01 -3.6903780e-02 -1.2527717e+00
 6.2545073e-01 -3.5939065e-01 -1.4900972e+00 -1.9662325e-01
 1.2965504e+00  4.4162169e-01 -6.0677487e-01  3.5281595e-02
-1.0948530e+00 -5.0222993e-01  6.9750226e-01  6.8898183e-01
-2.0692857e-01  5.2309901e-01 -2.5655949e-01 -5.6491476e-01
 2.9111046e-01 -1.0217363e+00  8.7742567e-02  5.8306962e-01
 9.3137074e-01  4.9069038e-01 -4.8059890e-01 -1.3509746e-01
 5.2000093e-01  7.1134603e-01  1.3783640e-01  5.2220756e-01]
```

```
 -2.1545863   0.44442797  1.5008385   0.37013933 -0.95732254  0.09472698
  1.282538   -0.49393523 -0.02676304  3.3601835   0.2620785   0.65872586
 -0.4761328   1.6324779  -1.3387561   1.4255477   2.4557397   1.3181896
  1.870883   -0.29281518  0.84948695  0.33969143 -0.7241366   0.47844407
  0.28270337  0.03167895  1.4791602  -0.10358367 -0.55697036 -1.7013754
 -0.29714927  1.3895985  -0.33867553 -0.8181919   1.502634   -1.1011679
  0.26961017  0.778705    2.2484624   0.74084854 -0.45319998 -0.39371097
  1.1493397  -0.6540119   1.6204468  -0.8738289   2.801269   -2.3894613
 -0.05820201  0.69549924 -0.07420762  1.1327161   1.5162895  -0.04188767
  0.9652415  -0.18125235 -0.7349877   0.90399444 -0.6299266   0.81560004
 -0.01464408  0.00556458 -0.8347827   0.22132875  1.3130034  -0.9515643
  1.4123026   0.55425435 -1.212903   -1.1494116  -1.0354859  -0.09322888
  1.530044   -1.41171    -1.1932929  -0.5560176   0.01642353 -0.01638625
  0.06716421 -0.6534987  -1.0089262  -0.53134364  0.1469945  -1.5401335
  0.459967   -1.6862453  -1.0182955   1.9829645   0.6042661  -0.849061
 -2.3393505   1.0447005  -1.7758425   1.3703077  -0.5134719  -0.12931241
 -1.1724932   0.2643291   0.5811653  -0.07067284  1.4699466   0.12644416
 -0.13025188  1.5268131  -0.5809271   0.3394266  -0.09844043  0.4170801
  0.28160053  1.4179286   1.2536975   0.2169149   1.886993    1.612087
 -0.01091411  0.13052274 -2.7505574   1.8229882   0.9678832   0.5093532
 -1.7388744   0.86047083 -0.699194    0.18871821  0.99944526 -2.1409123
  1.1931305   0.6401121   0.05446924 -0.378995    1.4420906   0.6149744
 -0.42139184 -2.3203094   0.60517824  0.2730079   0.41521722  0.36480287
 -0.7812488  -1.4232873   0.3512373  -0.752118   -0.01677406 -0.6638958
  0.19970295 -1.1059885  -1.1245214   0.5343639  -2.1266203   0.87750375
 -1.3588809   1.9507389   0.56259984  1.8441358   0.4733372  -0.04254376
 -0.14967598 -1.347448    0.3551721  -0.08415017 -0.5016499  -1.3473525
  1.1926268  -1.6520413  -0.5035629   1.0943375  -0.17207107 -0.0423224
  0.41918963  0.14745241 -0.7218392   1.6059989   0.29314318 -0.39101067
  1.7613485  -0.11837661 -1.7924457  -0.64101595  0.30908385  0.7389745
 -0.508938   -0.56834716 -0.42208624  1.5897692   1.1693047  -0.26928884
 -2.9497643  -0.08284117 -0.7465631   2.24033     0.44054943  1.2132704
  0.34566605  0.16616808  1.460146   -1.0056117   0.6567358  -0.45341998
 -1.4530392  -1.9595912   0.70561653 -0.60928786  0.43130884  0.3064113
  0.14420594 -0.19301859  0.4009118  -0.3851875  -0.13393563  0.98935086
 -0.8014665   0.83046275  0.36168185  0.93739724 -0.7289244   0.9199218
  0.43069646 -1.270597   -1.6912705  -0.9802689  -0.10007995 -0.23384632]
network: [-1.1206226   0.46929976 -1.2475082  -0.4968033  -0.81215364  0.986324
 -0.89545155 -0.49714336 -0.8108639  -0.28703278 -1.0430897   0.38558236
 -0.38009346  0.6577991  -0.57209116 -0.5124606  -1.1808584   0.8209754
  0.5388342   0.36665422  0.35352615 -0.8063917  -0.87446547  1.6248353
 -0.0806017   0.03627003  1.4182904   0.0943373  -0.35752738 -1.5286359
  0.14290072  0.59779644  0.9021042   1.9239448   1.2024854   0.25987428
  0.40910777  0.41919965 -0.00503314  1.7657218  -0.49631053 -0.9097308
 -0.2798299  -1.1026864  -2.0326424  -1.2997792  -0.0596413   0.6169917
 -0.5653299  -0.13858587 -0.57027864 -0.00796393 -0.03660374 -1.0081657
 -1.0371897   2.6007214   0.50280863 -1.0976678   2.294191    0.30446175
  0.39377758  0.2915537   0.82436234 -1.0620675  -1.9359453   0.5003206
  1.0862509  -1.1990756   0.5722062   0.41971946 -1.8580048   0.8741552
 -0.74731714  0.9396884   0.9168732  -0.374982   -0.31577465  0.69951075
  0.69048643  1.2785275  -0.29485     0.6861151  -0.25641778  1.4946313
 -0.37433788  1.5751268  -0.12581801 -0.24936435 -0.30283865  0.7476671
 -0.59002084 -1.0066515   1.0152053  -0.35993308  1.7766483   0.14970684
  2.192142    1.1255614   0.5685001   1.8481747  -0.7610472  -1.2762871
 -0.9061149  -0.9471791   0.52847755 -0.71912974 -0.74762464  0.7965363
  1.5744615   0.4992149   0.5208625   1.6654015   0.2764946  -1.2409607
 -1.2829058   0.92293334  0.2792265   0.9671421   1.1450374  -0.2661592
  0.4835037  -0.73895204 -0.4066459  -0.70401853  1.6084485   1.0525317
  0.20320821 -0.2947971   1.0127094  -1.2391763  -0.36698148  0.9518979
 -0.04410775 -0.38815716  1.6931221   1.0805794   1.2174757   2.1171784
 -0.21023537  1.7899673   1.9819084  -0.27944666  0.76290464 -0.7145382
  1.9418652  -1.0738676   0.7295225  -1.3115475   0.7993577  -0.6990147
 -0.7173117  -1.5185556  -0.04750445  1.480288   -1.7294652   1.3610003
```

```
-0.15146978  0.8719234  -0.5496832  -0.47097534 -1.6953915   0.86437553
 0.32487965 -0.635689   -0.7858946  -0.83757377 -0.11262869 -0.67260003
-0.13339676  0.20789109 -0.41153008  0.8266552   0.9325274   0.19653969
-0.73195     0.52807665  1.1733284  -0.38597798  0.07874259  0.38175565
-1.5881846  -0.17645283 -0.6878827   0.7080699  -0.55411875 -0.5835981
-1.0408401  -0.35562286 -0.50272304  1.5883068  -0.31893906 -1.3427792
 0.0466677   0.35959074 -0.25196782  0.20154984  0.09569424  0.8786357
 2.07518     0.5370294  -1.0200137   0.4856418  -0.6969587   0.5102851
-0.99113774  1.5583943  -0.41771835  0.35292044 -0.38791507  1.0980865
 0.94722885 -0.43464506  1.3745263   0.75868016  0.9782632  -1.7942606
-1.151543    0.17737459  1.3407533  -1.1963626   0.26145145 -1.570719
 1.5798938  -1.10155    -0.17034604  0.12037086 -0.5411737  -1.286857
-0.32446003 -0.60009336  0.2784581  -0.8280759   0.3966491   0.5881504
 1.2820139  -0.19519305  0.5085836  -0.98276126  1.8272828   0.409466
 0.5866264  -0.6855875   1.3367965  -1.7300042   1.4287658   0.14250085
 2.4390507   0.2973348   0.25724283  0.60353935  1.8300854   0.95508
 2.7282798   0.17890452  0.64743066  1.0866114  -0.55811816  0.55307
-0.28713062 -0.6052359   0.84454644 -0.7813904   1.3022902  -0.6287979
 1.2482715  -1.4320291   0.490542   -0.47282106 -0.07720867  0.05318451
-0.43062368 -0.2893979  -0.29758662  0.44758633  0.8951796   0.6497135
-1.9934219  -0.05157532 -0.5258242  -1.946106   -0.6026017   1.5946316
 0.89122355  1.4285022  -1.0674429  -1.167263   -1.1580374   0.11457033
 0.89783657  0.06252659 -0.21707885  0.3619433   0.7844435  -2.0602355
 2.2965612  -0.6577597  -0.55528885  0.27078968 -0.5444098  -0.0783911 ]
```

Program 5

Build and demonstrate an auto encoder network using neural layers for data compression on image dataset.

In [ ]:
```
!pip install tensorflow
```

In [ ]:
```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.datasets import mnist
import numpy as np

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32')/255.
x_test = x_test.astype('float32')/255.

x_train_flat = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test_flat = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

print(f"Shape of x_train_flat: {x_train_flat.shape} \nShape of x_test_flat: {x_test_flat.shap
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Shape of x_train_flat: (60000, 784)
Shape of x_test_flat: (10000, 784)
```

In [ ]:
```
input_img = Input(shape=(784,))

encoder = Dense(128, activation='relu')(input_img)
encoder = Dense(64, activation='relu')(encoder)
encoder = Dense(32, activation='relu')(encoder)

decoder = Dense(64, activation='relu')(encoder)
decoder = Dense(128, activation='relu')(decoder)
decoder = Dense(784, activation='sigmoid')(decoder)

autoencoder = Model(input_img, decoder)

autoencoder.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 32)	2,080
dense_3 (Dense)	(None, 64)	2,112
dense_4 (Dense)	(None, 128)	8,320
dense_5 (Dense)	(None, 784)	101,136

Total params: 222,384 (868.69 KB)

Trainable params: 222,384 (868.69 KB)

Non-trainable params: 0 (0.00 B)

In [ ]:
```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
history = autoencoder.fit(x_train_flat, x_train_flat,
                          epochs = 50,
                          batch_size = 256,
                          shuffle = True,
                          validation_data = (x_test_flat,x_test_flat))
```

```
Epoch 1/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 7s 15ms/step - accuracy: 0.0079 - loss: 0.3518 - val_accuracy: 0.
0074 - val_loss: 0.1732
Epoch 2/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.0095 - loss: 0.1616 - val_accuracy: 0.0
102 - val_loss: 0.1377
Epoch 3/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0097 - loss: 0.1361 - val_accuracy: 0.0
082 - val_loss: 0.1256
Epoch 4/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0090 - loss: 0.1255 - val_accuracy: 0.0
129 - val_loss: 0.1194
Epoch 5/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0106 - loss: 0.1196 - val_accuracy: 0.0
124 - val_loss: 0.1144
Epoch 6/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0106 - loss: 0.1148 - val_accuracy: 0.0
105 - val_loss: 0.1105
Epoch 7/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0096 - loss: 0.1112 - val_accuracy: 0.0
114 - val_loss: 0.1080
Epoch 8/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0106 - loss: 0.1087 - val_accuracy: 0.0
120 - val_loss: 0.1056
Epoch 9/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.0109 - loss: 0.1067 - val_accuracy: 0.0
114 - val_loss: 0.1040
Epoch 10/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.0114 - loss: 0.1044 - val_accuracy: 0.0
120 - val_loss: 0.1018
Epoch 11/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0119 - loss: 0.1025 - val_accuracy: 0.0
122 - val_loss: 0.1009
Epoch 12/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0114 - loss: 0.1011 - val_accuracy: 0.0
143 - val_loss: 0.0989
Epoch 13/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0115 - loss: 0.0998 - val_accuracy: 0.0
127 - val_loss: 0.0978
Epoch 14/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0124 - loss: 0.0986 - val_accuracy: 0.0
113 - val_loss: 0.0970
Epoch 15/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0133 - loss: 0.0976 - val_accuracy: 0.0
135 - val_loss: 0.0957
Epoch 16/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0131 - loss: 0.0964 - val_accuracy: 0.0
133 - val_loss: 0.0951
Epoch 17/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0132 - loss: 0.0960 - val_accuracy: 0.0
149 - val_loss: 0.0942
Epoch 18/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0136 - loss: 0.0951 - val_accuracy: 0.0
132 - val_loss: 0.0937
Epoch 19/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0133 - loss: 0.0942 - val_accuracy: 0.0
109 - val_loss: 0.0937
Epoch 20/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0134 - loss: 0.0939 - val_accuracy: 0.0
160 - val_loss: 0.0928
Epoch 21/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.0149 - loss: 0.0932 - val_accuracy: 0.0
146 - val_loss: 0.0925
```

```
Epoch 22/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.0142 - loss: 0.0928 - val_accuracy: 0.0
108 - val_loss: 0.0917
Epoch 23/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0143 - loss: 0.0925 - val_accuracy: 0.0
147 - val_loss: 0.0911
Epoch 24/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0153 - loss: 0.0919 - val_accuracy: 0.0
156 - val_loss: 0.0910
Epoch 25/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0150 - loss: 0.0915 - val_accuracy: 0.0
145 - val_loss: 0.0904
Epoch 26/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0159 - loss: 0.0910 - val_accuracy: 0.0
145 - val_loss: 0.0898
Epoch 27/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0155 - loss: 0.0904 - val_accuracy: 0.0
149 - val_loss: 0.0895
Epoch 28/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0142 - loss: 0.0902 - val_accuracy: 0.0
134 - val_loss: 0.0895
Epoch 29/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0149 - loss: 0.0897 - val_accuracy: 0.0
153 - val_loss: 0.0889
Epoch 30/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0153 - loss: 0.0893 - val_accuracy: 0.0
150 - val_loss: 0.0885
Epoch 31/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0148 - loss: 0.0892 - val_accuracy: 0.0
158 - val_loss: 0.0881
Epoch 32/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0152 - loss: 0.0888 - val_accuracy: 0.0
145 - val_loss: 0.0880
Epoch 33/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0143 - loss: 0.0886 - val_accuracy: 0.0
136 - val_loss: 0.0878
Epoch 34/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.0130 - loss: 0.0883 - val_accuracy: 0.0
158 - val_loss: 0.0877
Epoch 35/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.0133 - loss: 0.0882 - val_accuracy: 0.0
129 - val_loss: 0.0875
Epoch 36/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0141 - loss: 0.0880 - val_accuracy: 0.0
120 - val_loss: 0.0874
Epoch 37/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0136 - loss: 0.0876 - val_accuracy: 0.0
143 - val_loss: 0.0869
Epoch 38/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0145 - loss: 0.0874 - val_accuracy: 0.0
136 - val_loss: 0.0870
Epoch 39/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0146 - loss: 0.0876 - val_accuracy: 0.0
154 - val_loss: 0.0870
Epoch 40/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0142 - loss: 0.0873 - val_accuracy: 0.0
149 - val_loss: 0.0867
Epoch 41/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0133 - loss: 0.0871 - val_accuracy: 0.0
118 - val_loss: 0.0863
Epoch 42/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.0132 - loss: 0.0869 - val_accuracy: 0.0
156 - val_loss: 0.0863
```

```
Epoch 43/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0137 - loss: 0.0868 - val_accuracy: 0.0
138 - val_loss: 0.0861
Epoch 44/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0144 - loss: 0.0867 - val_accuracy: 0.0
123 - val_loss: 0.0865
Epoch 45/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0141 - loss: 0.0866 - val_accuracy: 0.0
136 - val_loss: 0.0859
Epoch 46/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0134 - loss: 0.0865 - val_accuracy: 0.0
132 - val_loss: 0.0860
Epoch 47/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.0137 - loss: 0.0864 - val_accuracy: 0.0
145 - val_loss: 0.0858
Epoch 48/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0136 - loss: 0.0863 - val_accuracy: 0.0
134 - val_loss: 0.0858
Epoch 49/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0136 - loss: 0.0861 - val_accuracy: 0.0
127 - val_loss: 0.0856
Epoch 50/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.0141 - loss: 0.0860 - val_accuracy: 0.0
128 - val_loss: 0.0856
```

In [ ]:

```
test_loss, test_acc = autoencoder.evaluate(x_test_flat, x_test_flat)
print(f'Accuracy: {test_acc}, Loss: {test_loss}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.0152 - loss: 0.0857
Accuracy: 0.012799999676644802, Loss: 0.08559110760688782
```

Program 6

Design and implement a deep learning network for classification of textual documents.

In [ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

df = pd.read_csv('/content/text_data.csv')

texts = df['text'].tolist()
labels = df['label'].tolist()

max_words = 1000
tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

max_sequence_length = 20
padded_sequences = pad_sequences(sequences, maxlen=max_sequence_length, padding='post', trunc

X_train, X_test, y_train, y_test = train_test_split(padded_sequences, labels, test_size=0.2,

print("Preprocessing complete.")
print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Length of y_train: {len(y_train)}")
print(f"Length of y_test: {len(y_test)}")
```

```
Preprocessing complete.
Shape of X_train: (40, 20)
Shape of X_test: (10, 20)
Length of y_train: 40
Length of y_test: 10
```

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, GlobalAveragePooling1D

vocab_size = len(tokenizer.word_index) + 1

embedding_dim = 16

model = Sequential([
    Embedding(input_dim = vocab_size,
              output_dim = embedding_dim,
              input_length = max_sequence_length),
    GlobalAveragePooling1D(),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: UserWarning: Ar
gument `input_length` is deprecated. Just remove it.
  warnings.warn(
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	?	0 (unbuilt)
global_average_pooling1d_2 (GlobalAveragePooling1D)	?	0
dense_4 (Dense)	?	0 (unbuilt)
dense_5 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

In [ ]:

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

y_train = np.array(y_train)
y_test = np.array(y_test)

history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}")
print(f"Test Accuracy: {accuracy}")
```

```
Epoch 1/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 261ms/step - accuracy: 0.4667 - loss: 0.6935 - val_accuracy: 0.400
0 - val_loss: 0.6933
Epoch 2/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - accuracy: 0.5875 - loss: 0.6923 - val_accuracy: 0.4000
- val_loss: 0.6944
Epoch 3/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - accuracy: 0.5167 - loss: 0.6911 - val_accuracy: 0.4000
- val_loss: 0.6957
Epoch 4/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.5271 - loss: 0.6901 - val_accuracy: 0.4000
- val_loss: 0.6966
Epoch 5/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 0.5375 - loss: 0.6888 - val_accuracy: 0.4000
- val_loss: 0.6970
Epoch 6/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.5167 - loss: 0.6885 - val_accuracy: 0.4000
- val_loss: 0.6971
Epoch 7/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.5271 - loss: 0.6871 - val_accuracy: 0.4000
- val_loss: 0.6974
Epoch 8/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 0.5271 - loss: 0.6861 - val_accuracy: 0.4000
- val_loss: 0.6977
Epoch 9/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step - accuracy: 0.5271 - loss: 0.6850 - val_accuracy: 0.4000
- val_loss: 0.6980
Epoch 10/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.5479 - loss: 0.6828 - val_accuracy: 0.4000
- val_loss: 0.6980
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - accuracy: 0.4000 - loss: 0.6980
Test Loss: 0.6980057954788208
Test Accuracy: 0.4000000059604645
```

Program 7

Design and implement a deep learning network for forecasting time series data.

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
dates = pd.date_range(start='2023-01-01', periods=1000, freq='H')
data = np.random.rand(1000, 2) * 100
df_dummy = pd.DataFrame(data, index=dates, columns=['Feature1', 'Feature2'])
df_dummy.index.name = 'Timestamp'

df_dummy.iloc[50:150, 0] = np.nan
df_dummy.iloc[200:300, 1] = np.nan

df_dummy.to_csv('time_series_data.csv')

df = pd.read_csv('time_series_data.csv')

print(df.head())
print(df.info())
print(df.isnull().sum())

if 'Timestamp' in df.columns:
    df['Timestamp'] = pd.to_datetime(df['Timestamp'])
    df.set_index('Timestamp', inplace=True)
    df.sort_index(inplace=True)
else:
    print("Time column 'Timestamp' not found. Please check the column name.")

df.fillna(method='ffill', inplace=True)

print(df.head())
```

```
             Timestamp   Feature1   Feature2
0  2023-01-01 00:00:00  80.665638  61.746969
1  2023-01-01 01:00:00  75.401201  67.817525
2  2023-01-01 02:00:00  69.853507  79.432850
3  2023-01-01 03:00:00  62.855872  47.992442
4  2023-01-01 04:00:00  55.013834  69.511280
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Timestamp  1000 non-null   object 
 1   Feature1   900 non-null    float64
 2   Feature2   900 non-null    float64
dtypes: float64(2), object(1)
memory usage: 23.6+ KB
None
Timestamp      0
Feature1     100
Feature2     100
dtype: int64
                      Feature1   Feature2
Timestamp                                
2023-01-01 00:00:00  80.665638  61.746969
2023-01-01 01:00:00  75.401201  67.817525
2023-01-01 02:00:00  69.853507  79.432850
2023-01-01 03:00:00  62.855872  47.992442
2023-01-01 04:00:00  55.013834  69.511280
```

```
/tmp/ipython-input-1552314111.py:5: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h' instead.
  dates = pd.date_range(start='2023-01-01', periods=1000, freq='H')
/tmp/ipython-input-1552314111.py:28: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df.fillna(method='ffill', inplace=True)
```

In [ ]:

```
train_size = int(len(df) * 0.8)
train_df = df.iloc[:train_size]
test_df = df.iloc[train_size:]

print("\nTraining set shape:", train_df.shape)
print("Testing set shape:", test_df.shape)

scaler = MinMaxScaler()

train_scaled = scaler.fit_transform(train_df)
train_df_scaled = pd.DataFrame(train_scaled, index=train_df.index, columns=train_df.columns)

test_scaled = scaler.transform(test_df)
test_df_scaled = pd.DataFrame(test_scaled, index=test_df.index, columns=test_df.columns)

print("\nScaled Training set head:")
print(train_df_scaled.head())
print("\nScaled Testing set head:")
print(test_df_scaled.head())
```

```
Training set shape: (800, 2)
Testing set shape: (200, 2)

Scaled Training set head:
                     Feature1  Feature2
Timestamp
2023-01-01 00:00:00  0.808719  0.617137
2023-01-01 01:00:00  0.755772  0.677929
2023-01-01 02:00:00  0.699977  0.794247
2023-01-01 03:00:00  0.629598  0.479396
2023-01-01 04:00:00  0.550727  0.694890

Scaled Testing set head:
                     Feature1  Feature2
Timestamp
2023-02-03 08:00:00  0.372702  0.110983
2023-02-03 09:00:00  0.152667  0.534928
2023-02-03 10:00:00  0.173536  0.311897
2023-02-03 11:00:00  0.698400  0.663447
2023-02-03 12:00:00  0.505722  0.806891
```

In [ ]:

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

n_steps = 24

n_features = train_df_scaled.shape[1]
input_shape = (n_steps, n_features)
model = Sequential()

model.add(LSTM(units=50, activation='relu', input_shape=input_shape, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(units=50, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(units=n_features))
model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not p
ass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer usi
ng an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 24, 50)	10,600
dropout (Dropout)	(None, 24, 50)	0
lstm_1 (LSTM)	(None, 50)	20,200
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 2)	102

Total params: 30,902 (120.71 KB)

Trainable params: 30,902 (120.71 KB)

Non-trainable params: 0 (0.00 B)

In [ ]:
```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 24, 50)	10,600
dropout (Dropout)	(None, 24, 50)	0
lstm_1 (LSTM)	(None, 50)	20,200
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 2)	102

Total params: 30,902 (120.71 KB)

Trainable params: 30,902 (120.71 KB)

Non-trainable params: 0 (0.00 B)

In [ ]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

def create_sequences(data, n_steps):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data.iloc[i:(i + n_steps)].values)
        y.append(data.iloc[i + n_steps].values)
    return np.array(X), np.array(y)

X_train, y_train = create_sequences(train_df_scaled, n_steps)

X_test, y_test = create_sequences(test_df_scaled, n_steps)

history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_te

y_pred_scaled = model.predict(X_test)
y_test_actual = scaler.inverse_transform(y_test)
y_pred_actual = scaler.inverse_transform(y_pred_scaled)

rmse = np.sqrt(mean_squared_error(y_test_actual, y_pred_actual))
mae = mean_absolute_error(y_test_actual, y_pred_actual)

print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Mean Absolute Error (MAE): {mae}")

r2 = r2_score(y_test_actual, y_pred_actual)
print(f"R-squared (R2) Score: {r2}")
```

```
Epoch 1/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 7s 42ms/step - loss: 0.1931 - mae: 0.3507 - val_loss: 0.0947 - val_
mae: 0.2607
Epoch 2/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0935 - mae: 0.2571 - val_loss: 0.0923 - val_
mae: 0.2574
Epoch 3/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0872 - mae: 0.2474 - val_loss: 0.0943 - val_
mae: 0.2599
Epoch 4/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0863 - mae: 0.2441 - val_loss: 0.0883 - val_
mae: 0.2570
Epoch 5/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0845 - mae: 0.2423 - val_loss: 0.0889 - val_
mae: 0.2600
Epoch 6/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0862 - mae: 0.2488 - val_loss: 0.0895 - val_
mae: 0.2587
Epoch 7/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0850 - mae: 0.2439 - val_loss: 0.0886 - val_
mae: 0.2570
Epoch 8/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0822 - mae: 0.2385 - val_loss: 0.0893 - val_
mae: 0.2568
Epoch 9/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0805 - mae: 0.2332 - val_loss: 0.0892 - val_
mae: 0.2572
Epoch 10/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0855 - mae: 0.2398 - val_loss: 0.0881 - val_
mae: 0.2575
Epoch 11/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0828 - mae: 0.2379 - val_loss: 0.0883 - val_
mae: 0.2595
Epoch 12/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0831 - mae: 0.2397 - val_loss: 0.0879 - val_
mae: 0.2577
Epoch 13/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0822 - mae: 0.2378 - val_loss: 0.0885 - val_
mae: 0.2571
Epoch 14/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0847 - mae: 0.2422 - val_loss: 0.0923 - val_
mae: 0.2587
Epoch 15/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - loss: 0.0830 - mae: 0.2349 - val_loss: 0.0890 - val_
mae: 0.2575
Epoch 16/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - loss: 0.0859 - mae: 0.2395 - val_loss: 0.0885 - val_
mae: 0.2576
Epoch 17/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step - loss: 0.0826 - mae: 0.2359 - val_loss: 0.0889 - val_
mae: 0.2573
Epoch 18/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0819 - mae: 0.2354 - val_loss: 0.0877 - val_
mae: 0.2579
Epoch 19/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - loss: 0.0788 - mae: 0.2299 - val_loss: 0.0877 - val_
mae: 0.2585
Epoch 20/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0819 - mae: 0.2396 - val_loss: 0.0879 - val_
mae: 0.2578
Epoch 21/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0801 - mae: 0.2351 - val_loss: 0.0878 - val_
mae: 0.2583
```

```
Epoch 22/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0766 - mae: 0.2257 - val_loss: 0.0877 - val_
mae: 0.2591
Epoch 23/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0776 - mae: 0.2276 - val_loss: 0.0887 - val_
mae: 0.2609
Epoch 24/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 47ms/step - loss: 0.0827 - mae: 0.2374 - val_loss: 0.0874 - val_
mae: 0.2587
Epoch 25/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 56ms/step - loss: 0.0822 - mae: 0.2400 - val_loss: 0.0881 - val_
mae: 0.2577
Epoch 26/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0806 - mae: 0.2338 - val_loss: 0.0877 - val_
mae: 0.2595
Epoch 27/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0784 - mae: 0.2326 - val_loss: 0.0885 - val_
mae: 0.2611
Epoch 28/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - loss: 0.0788 - mae: 0.2324 - val_loss: 0.0876 - val_
mae: 0.2582
Epoch 29/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - loss: 0.0783 - mae: 0.2316 - val_loss: 0.0878 - val_
mae: 0.2582
Epoch 30/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - loss: 0.0794 - mae: 0.2329 - val_loss: 0.0894 - val_
mae: 0.2577
Epoch 31/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - loss: 0.0829 - mae: 0.2382 - val_loss: 0.0895 - val_
mae: 0.2579
Epoch 32/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0750 - mae: 0.2226 - val_loss: 0.0888 - val_
mae: 0.2612
Epoch 33/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0804 - mae: 0.2340 - val_loss: 0.0900 - val_
mae: 0.2635
Epoch 34/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0824 - mae: 0.2377 - val_loss: 0.0876 - val_
mae: 0.2589
Epoch 35/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0787 - mae: 0.2328 - val_loss: 0.0885 - val_
mae: 0.2579
Epoch 36/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0828 - mae: 0.2367 - val_loss: 0.0876 - val_
mae: 0.2585
Epoch 37/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0780 - mae: 0.2307 - val_loss: 0.0877 - val_
mae: 0.2585
Epoch 38/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0801 - mae: 0.2339 - val_loss: 0.0876 - val_
mae: 0.2587
Epoch 39/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - loss: 0.0789 - mae: 0.2306 - val_loss: 0.0882 - val_
mae: 0.2601
Epoch 40/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0809 - mae: 0.2350 - val_loss: 0.0878 - val_
mae: 0.2594
Epoch 41/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0787 - mae: 0.2315 - val_loss: 0.0878 - val_
mae: 0.2596
Epoch 42/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - loss: 0.0793 - mae: 0.2309 - val_loss: 0.0878 - val_
mae: 0.2583
```

```
Epoch 43/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0779 - mae: 0.2298 - val_loss: 0.0883 - val_
mae: 0.2578
Epoch 44/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0769 - mae: 0.2269 - val_loss: 0.0876 - val_
mae: 0.2589
Epoch 45/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - loss: 0.0791 - mae: 0.2334 - val_loss: 0.0881 - val_
mae: 0.2580
Epoch 46/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - loss: 0.0763 - mae: 0.2265 - val_loss: 0.0878 - val_
mae: 0.2583
Epoch 47/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step - loss: 0.0800 - mae: 0.2331 - val_loss: 0.0876 - val_
mae: 0.2586
Epoch 48/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0783 - mae: 0.2318 - val_loss: 0.0882 - val_
mae: 0.2598
Epoch 49/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0801 - mae: 0.2350 - val_loss: 0.0877 - val_
mae: 0.2597
Epoch 50/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - loss: 0.0792 - mae: 0.2317 - val_loss: 0.0877 - val_
mae: 0.2592
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 65ms/step
Root Mean Squared Error (RMSE): 29.50102036254032
Mean Absolute Error (MAE): 25.82512457419641
R-squared (R2) Score: 0.004177907580553308
```

Program 8

Write a program to read a dataset of text reviews. Classify the reviews as positive or negative.

In [ ]:
```
!pip install transformers datasets pandas
```

In [ ]:
```
from datasets import load_dataset
from transformers import pipeline

dataset = load_dataset("imdb")

print(dataset['train'][0])
print(dataset['train'][1])

classifier = pipeline("sentiment-analysis")
review = "This movie was amazing! I loved every part of it."
result = classifier(review)

print(f"Review: {review}")
print(f"Sentiment: {result[0]['label']}, Score: {result[0]['score']:.2f}")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggin
gface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or d
atasets.
  warnings.warn(
README.md: 0.00B [00:00, ?B/s]
plain_text/train-00000-of-00001.parquet:   0%|          | 0.00/21.0M [00:00<?, ?B/s]
plain_text/test-00000-of-00001.parquet:   0%|          | 0.00/20.5M [00:00<?, ?B/s]
plain_text/unsupervised-00000-of-00001.p(…):   0%|          | 0.00/42.0M [00:00<?, ?B/s]
Generating train split:   0%|          | 0/25000 [00:00<?, ? examples/s]
Generating test split:   0%|          | 0/25000 [00:00<?, ? examples/s]
Generating unsupervised split:   0%|          | 0/50000 [00:00<?, ? examples/s]
No model was supplied, defaulted to distilbert/distilbert-base-uncased-finetuned-sst-2-english
and revision 714eb0f (https://huggingface.co/distilbert/distilbert-base-uncased-finetuned-sst-
2-english).
Using a pipeline without specifying a model name and revision in production is not recommende
d.
```

```
{'text': 'I rented I AM CURIOUS-YELLOW from my video store because of all the controversy that surrounded it when it was first released in 1967. I also heard that at first it was seized by U.S. customs if it ever tried to enter this country, therefore being a fan of films considered "controversial" I really had to see this for myself.<br /><br />The plot is centered around a young Swedish drama student named Lena who wants to learn everything she can about life. In particular she wants to focus her attentions to making some sort of documentary on what the average Swede thought about certain political issues such as the Vietnam War and race issues in the United States. In between asking politicians and ordinary denizens of Stockholm about their opinions on politics, she has sex with her drama teacher, classmates, and married men.<br /><br />What kills me about I AM CURIOUS-YELLOW is that 40 years ago, this was considered pornographic. Really, the sex and nudity scenes are few and far between, even then it\'s not shot like some cheaply made porno. While my countrymen mind find it shocking, in reality sex and nudity are a major staple in Swedish cinema. Even Ingmar Bergman, arguably their answer to good old boy John Ford, had sex scenes in his films.<br /><br />I do commend the filmmakers for the fact that any sex shown in the film is shown for artistic purposes rather than just to shock people and make money to be shown in pornographic theaters in America. I AM CURIOUS-YELLOW is a good film for anyone wanting to study the meat and potatoes (no pun intended) of Swedish cinema. But really, this film doesn\'t have much of a plot.', 'label': 0}
{'text': '"I Am Curious: Yellow" is a risible and pretentious steaming pile. It doesn\'t matter what one\'s political views are because this film can hardly be taken seriously on any level. As for the claim that frontal male nudity is an automatic NC-17, that isn\'t true. I\'ve seen R-rated films with male nudity. Granted, they only offer some fleeting views, but where are the R-rated films with gaping vulvas and flapping labia? Nowhere, because they don\'t exist. The same goes for those crappy cable shows: schlongs swinging in the breeze but not a clitoris in sight. And those pretentious indie movies like The Brown Bunny, in which we\'re treated to the site of Vincent Gallo\'s throbbing johnson, but not a trace of pink visible on Chloe Sevigny. Before crying (or implying) "double-standard" in matters of nudity, the mentally obtuse should take into account one unavoidably obvious anatomical difference between men and women: there are no genitals on display when actresses appears nude, and the same cannot be said for a man. In fact, you generally won\'t see female genitals in an American film in anything short of porn or explicit erotica. This alleged double-standard is less a double standard than an admittedly depressing ability to come to terms culturally with the insides of women\'s bodies.', 'label': 0}
config.json:   0%|          | 0.00/629 [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
vocab.txt: 0.00B [00:00, ?B/s]
Device set to use cpu
Review: This movie was amazing! I loved every part of it.
Sentiment: POSITIVE, Score: 1.00
```

In [ ]:

```python
reviews = [
    "This movie was great!",
    "This movie was terrible.",
    "It was an okay movie, nothing special."
]
results = classifier(reviews)
for review, result in zip(reviews, results):
    print(f"Review: {review}")
    print(f"Sentiment: {result['label']}, Score: {result['score']:.2f}")
```

```
Review: This movie was great!
Sentiment: POSITIVE, Score: 1.00
Review: This movie was terrible.
Sentiment: NEGATIVE, Score: 1.00
Review: It was an okay movie, nothing special.
Sentiment: NEGATIVE, Score: 0.96
```