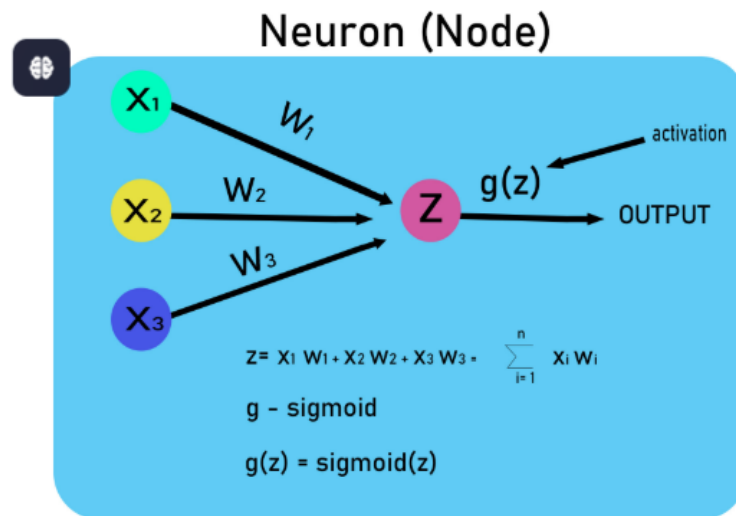


Experiment No. 1

Aim: Write a program to build the deep neural network using NumPy

Description:

Objective: To implement a Deep Neural Network (DNN) using NumPy for binary classification on a toy dataset.

Steps Overview:

- Import libraries
- Generate or load dataset
- Initialize parameters
- Define activation functions
- Forward propagation
- Compute cost
- Backward propagation
- Update parameters
- Train the model
- Make predictions
- Evaluate accuracy

Implementation:**1. Import Libraries**

```
[1] import numpy as np
    from sklearn.datasets import make_moons
    from sklearn.model_selection import train_test_split
    import matplotlib.pyplot as plt
```

numpy: A fundamental package for numerical computing in Python. We use it for matrix operations, random number generation, and array manipulation.

make_moons: A function from `sklearn.datasets` that generates a toy dataset, useful for binary classification tasks.

train_test_split: A function that splits the dataset into training and test sets.

matplotlib.pyplot: A library for creating visualizations. We use it for plotting decision boundaries and training results.

2. Load and Preprocess Dataset

```
# Create toy dataset
X, Y = make_moons(n_samples=1000, noise=0.2, random_state=42)
Y = Y.reshape(Y.shape[0], 1) # reshape to (1000, 1)

# Train-Test split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# Transpose for compatibility (features x samples)
X_train, Y_train = X_train.T, Y_train.T
X_test, Y_test = X_test.T, Y_test.T
```

`make_moons()`: Generates a 2D dataset that resembles two moon-shaped clusters (often used for classification tasks). `X` is the feature set, and `Y` is the target (binary).

`train_test_split()`: Divides the dataset into training (80%) and testing (20%) sets.

`.T`: The `.T` operation transposes the matrices to ensure the data is in the shape (features, samples).

3. Initialize Parameters

```
[3] def initialize_parameters(layer_dims):
    np.random.seed(1)
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters[f"w{l}"] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
        parameters[f"b{l}"] = np.zeros((layer_dims[l], 1))

    return parameters
```

`layer_dims`: A list of integers representing the number of neurons in each layer (e.g., [2, 10, 5, 1] for 2 input neurons, 10 neurons in the first hidden layer, 5 in the second, and 1 output neuron).

Initialization:

- **Weights (W)**: Initialized randomly using `np.random.randn()` and scaled by 0.01.
- **Biases (b)**: Initialized to zero for all layers.

Purpose: We initialize the parameters (`W` and `b`) for each layer in the network

4. Activation Functions

```
def sigmoid(Z):  
    return 1 / (1 + np.exp(-Z))  
  
def relu(Z):  
    return np.maximum(0, Z)  
  
def relu_derivative(Z):  
    return Z > 0  
  
def sigmoid_derivative(A):  
    return A * (1 - A)
```

Sigmoid: A logistic function that squashes inputs to a range between 0 and 1. Often used for binary classification (like our output layer).

ReLU (Rectified Linear Unit): A function that outputs the input directly if it's positive; otherwise, it outputs 0. This is commonly used in hidden layers to avoid vanishing gradients.

Derivatives:

- **sigmoid_derivative:** For the output layer, it computes the derivative of the sigmoid function, which is used in the backward pass for weight updates.
- **relu_derivative:** Computes the derivative of the ReLU function for the backward pass.

5. Forward Propagation

```
def forward_propagation(X, parameters):  
    caches = {}  
    A = X  
    caches['A0'] = A # <-- store input explicitly  
    L = len(parameters) // 2  
  
    for l in range(1, L):  
        Z = parameters[f"W{l}"] @ A + parameters[f"b{l}"]  
        A = relu(Z)  
        caches[f"Z{l}"] = Z  
        caches[f"A{l}"] = A  
  
    ZL = parameters[f"W{L}"] @ A + parameters[f"b{L}"]  
    AL = sigmoid(ZL)  
    caches[f"Z{L}"] = ZL  
    caches[f"A{L}"] = AL  
  
    return AL, caches
```

Input Layer (A0): The input X is stored as A0 in the caches dictionary.

Hidden Layers:

- The forward pass computes the activation for each layer using the formula
- The activation function (ReLU) is applied to the result of the linear transformation Z.

- Store the intermediate values A (activations) and Z (linear combinations) in caches for later use in backward propagation.

Output Layer: The final layer uses the sigmoid function to get a probability value between 0 and 1.

6. Compute Cost

```
[6] def compute_cost(AL, Y):
    m = Y.shape[1]
    cost = -np.sum(Y * np.log(AL) + (1 - Y) * np.log(1 - AL)) / m
    return np.squeeze(cost)
```

Cost Function: The cost function is the binary cross-entropy loss, which measures how well the predictions match the actual labels:

where AL is the predicted output, and Y is the actual label.

Why we use this? Since we are performing binary classification, the binary cross-entropy loss is appropriate for measuring prediction accuracy.

7. Backward Propagation

```
def backward_propagation(AL, Y, caches, parameters):
    grads = {}
    L = len(parameters) // 2
    m = Y.shape[1]

    dAL = -(np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    dZL = dAL * sigmoid_derivative(AL)

    grads[f"dW{L}"] = (1 / m) * dZL @ caches[f"A{L-1}"].T
    grads[f"db{L}"] = (1 / m) * np.sum(dZL, axis=1, keepdims=True)

    dA_prev = parameters[f"W{L}"].T @ dZL

    for l in reversed(range(1, L)):
        dZ = dA_prev * relu_derivative(caches[f"Z{l}"])
        grads[f"dW{l}"] = (1 / m) * dZ @ caches[f"A{l-1}"].T
        grads[f"db{l}"] = (1 / m) * np.sum(dZ, axis=1, keepdims=True)
        dA_prev = parameters[f"W{l}"].T @ dZ

    return grads
```

Backpropagation:

- **Output Layer:** We first compute the gradient of the cost with respect to the output activation (dAL) and use the derivative of the sigmoid to compute dZL (the error term for the output layer).
- **Hidden Layers:** For each hidden layer, the error (dZ) is computed by propagating the error from the next layer (dA_prev) and applying the derivative of the ReLU function.
- **Gradients for Weights and Biases:** We compute gradients of the cost with respect to the weights (dW) and biases (db) for all layers.

8. Update Parameters

```
[8] def update_parameters(parameters, grads, learning_rate):  
    L = len(parameters) // 2  
    for l in range(1, L + 1):  
        parameters[f"w{l}"] -= learning_rate * grads[f"dw{l}"]  
        parameters[f"b{l}"] -= learning_rate * grads[f"db{l}"]  
    return parameters
```

Gradient Descent: We update the weights and biases using the gradients calculated during backpropagation. The weights are adjusted by subtracting the gradient scaled by the learning rate.

9. Train the Model

```
[9] def model(X, Y, layer_dims, learning_rate=0.1, epochs=1000, print_cost=True):  
    parameters = initialize_parameters(layer_dims)  
  
    for i in range(epochs):  
        AL, caches = forward_propagation(X, parameters)  
        cost = compute_cost(AL, Y)  
        grads = backward_propagation(AL, Y, caches, parameters)  
        parameters = update_parameters(parameters, grads, learning_rate)  
  
        if print_cost and i % 100 == 0:  
            print(f"Cost after epoch {i}: {cost:.4f}")  
  
    return parameters
```

Model Training:

- **Loop over epochs:** For each epoch, we perform forward propagation, compute the cost, perform backward propagation to get gradients, and update the parameters using gradient descent.
- **Print cost:** Every 100 epochs, we print the current cost to monitor the training process.

10. Make Predictions

```
[10] def predict(X, parameters):  
    AL, _ = forward_propagation(X, parameters)  
    return (AL > 0.5).astype(int)
```

Prediction: Given the input X and trained parameters, the model predicts the output. If the output is greater than 0.5, it predicts 1, else it predicts 0.

11. Evaluate

```
layer_dims = [2, 10, 5, 1] # input layer, 2 hidden layers, output
parameters = model(X_train, Y_train, layer_dims, epochs=1000)

# Predict and evaluate
train_pred = predict(X_train, parameters)
test_pred = predict(X_test, parameters)

train_acc = np.mean(train_pred == Y_train) * 100
test_acc = np.mean(test_pred == Y_test) * 100

print(f"Train Accuracy: {train_acc:.2f}%")
print(f"Test Accuracy: {test_acc:.2f}%")
```

Accuracy: After training, we predict on both the training and test sets and compute the accuracy by comparing the predictions with the actual labels.

Output:

```
↔ Cost after epoch 0: 0.6931
Cost after epoch 100: 0.6931
Cost after epoch 200: 0.6931
Cost after epoch 300: 0.6931
Cost after epoch 400: 0.6931
Cost after epoch 500: 0.6931
Cost after epoch 600: 0.6931
Cost after epoch 700: 0.6931
Cost after epoch 800: 0.6931
Cost after epoch 900: 0.6931
Train Accuracy: 78.25%
Test Accuracy: 77.00%
```

Conclusion:

This code implements a Deep Neural Network from scratch using NumPy, with key components like forward propagation, backward propagation, cost computation, and gradient descent for training. We can use this as a base for experimenting with different architectures and learning rates

Experiment No. 2

Aim: write a program to regularization in the deep learning model to handle the over fitting, and also compare the various optimization methods that can speed up learning and parameter optimization

Description: In deep learning, overfitting occurs when the model learns the details and noise in the training data to the extent that it negatively impacts the performance on new data. Regularization techniques such as L1, L2 regularization, and dropout can help reduce overfitting. Additionally, optimization methods such as Gradient Descent, Adam, and RMSprop can help speed up the learning process by adapting the learning rate.

Objective: The goal is to create a deep learning model with regularization techniques to prevent overfitting, and compare different optimization algorithms in terms of their efficiency and speed in training the model.

Steps Overview:

Data Preparation: Use a dataset (e.g., MNIST, CIFAR-10).

Model Design: Create a simple neural network.

Regularization Techniques:

- L2 Regularization (Ridge Regularization)
- Dropout

Optimization Methods:

- Gradient Descent (SGD)
- Adam Optimizer
- RMSprop

Model Training: Train the model with various optimization methods.

Evaluate: Compare the models based on loss, accuracy, and speed of convergence.

Implementation:

```
# --- Imports ---  
  
import tensorflow as tf  
import numpy as np  
import time  
  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Dropout, Flatten  
from tensorflow.keras.optimizers import SGD, Adam, RMSprop  
from tensorflow.keras.regularizers import l2
```

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# --- Eager Execution ---
try:
    tf.config.run_functions_eagerly(True)
except Exception as e:
    print("Eager execution couldn't be set explicitly:", e)

# --- Load and Preprocess MNIST ---
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train / 255.0
x_test = x_test / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# --- Model Builder ---
def create_model(optimizer, regularizer_strength=0.01, use_dropout=False):
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28, 1)))
    model.add(Dense(128, activation='relu', kernel_regularizer=l2(regularizer_strength)))

    if use_dropout:
        model.add(Dropout(0.5))

    model.add(Dense(64, activation='relu', kernel_regularizer=l2(regularizer_strength)))
    model.add(Dense(10, activation='softmax'))
```



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

    return model

# --- Train + Evaluate ---

def train_and_evaluate_model(optimizer, use_dropout=False):

    model = create_model(optimizer, use_dropout=use_dropout)

    start_time = time.time()

    history = model.fit(

        x_train, y_train,

        epochs=5,

        batch_size=32,

        validation_data=(x_test, y_test),

        verbose=2

    )

    elapsed_time = time.time() - start_time

    return history, elapsed_time

# --- Define Optimizer Classes ---

optimizer_classes = {

    "SGD": SGD,

    "Adam": Adam,

    "RMSprop": RMSprop

}

results = {}

# --- Run Experiments ---

for name, OptimizerClass in optimizer_classes.items():

    print(f"\nTraining with {name} optimizer (without dropout):")

    history, time_taken = train_and_evaluate_model(OptimizerClass(), use_dropout=False)
```

```
results[f"{name}_no_dropout"] = {'history': history, 'time_taken': time_taken}
print(f"\nTraining with {name} optimizer (with dropout):")
history, time_taken = train_and_evaluate_model(OptimizerClass(), use_dropout=True)
results[f"{name}_with_dropout"] = {'history': history, 'time_taken': time_taken}

# --- Output Final Results ---
for key, value in results.items():
    final_val_acc = value['history'].history['val_accuracy'][-1]
    print(f"\n{key}:")
    print(f"Final Validation Accuracy: {final_val_acc:.4f}")
    print(f"Time Taken: {value['time_taken']:.2f} seconds")
```

Data Preprocessing: We load the MNIST dataset and normalize it. We also reshape the data to fit the model.

Model Design:

- The model consists of two dense layers with ReLU activations, followed by a softmax layer for classification.
- Regularization (L2) is applied to the weights of the dense layers to prevent overfitting.
- Dropout is applied in one of the layers to further reduce overfitting.

Optimization Algorithms: We experiment with three optimization algorithms:

- **SGD (Stochastic Gradient Descent)**
- **Adam**
- **RMSprop**

Model Training: We train the model for 5 epochs and measure the time taken to complete the training.

Results: We compare training times and validation accuracies for each optimization method, with and without dropout.

Output:

Training with SGD optimizer (without dropout):

Epoch 1/5
1875/1875 - 84s - 45ms/step - accuracy: 0.8286 - loss: 2.9371 - val_accuracy: 0.9040 - val_loss: 1.9749
Epoch 2/5
1875/1875 - 142s - 76ms/step - accuracy: 0.9071 - loss: 1.5488 - val_accuracy: 0.9167 - val_loss: 1.1876
Epoch 3/5
1875/1875 - 138s - 74ms/step - accuracy: 0.9161 - loss: 0.9938 - val_accuracy: 0.9181 - val_loss: 0.8174
Epoch 4/5
1875/1875 - 83s - 44ms/step - accuracy: 0.9212 - loss: 0.7249 - val_accuracy: 0.9258 - val_loss: 0.6312
Epoch 5/5
1875/1875 - 80s - 43ms/step - accuracy: 0.9250 - loss: 0.5891 - val_accuracy: 0.9304 - val_loss: 0.5352

Training with SGD optimizer (with dropout):

Epoch 1/5
1875/1875 - 95s - 51ms/step - accuracy: 0.7213 - loss: 3.1846 - val_accuracy: 0.8954 - val_loss: 2.0079
Epoch 2/5
1875/1875 - 90s - 48ms/step - accuracy: 0.8622 - loss: 1.6834 - val_accuracy: 0.9122 - val_loss: 1.2057
Epoch 3/5
1875/1875 - 145s - 77ms/step - accuracy: 0.8841 - loss: 1.0989 - val_accuracy: 0.9209 - val_loss: 0.8266
Epoch 4/5
1875/1875 - 89s - 47ms/step - accuracy: 0.8929 - loss: 0.8212 - val_accuracy: 0.9277 - val_loss: 0.6383
Epoch 5/5
1875/1875 - 91s - 48ms/step - accuracy: 0.9011 - loss: 0.6785 - val_accuracy: 0.9292 - val_loss: 0.5468

Training with Adam optimizer (without dropout):

Epoch 1/5
1875/1875 - 130s - 69ms/step - accuracy: 0.8957 - loss: 0.7956 - val_accuracy: 0.9222 - val_loss: 0.5427
Epoch 2/5
1875/1875 - 150s - 80ms/step - accuracy: 0.9242 - loss: 0.5015 - val_accuracy: 0.9323 - val_loss: 0.4467
Epoch 3/5
1875/1875 - 146s - 78ms/step - accuracy: 0.9342 - loss: 0.4390 - val_accuracy: 0.9389 - val_loss: 0.4114
Epoch 4/5
1875/1875 - 195s - 104ms/step - accuracy: 0.9396 - loss: 0.3996 - val_accuracy: 0.9353 - val_loss: 0.3941
Epoch 5/5
1875/1875 - 147s - 78ms/step - accuracy: 0.9432 - loss: 0.3725 - val_accuracy: 0.9553 - val_loss: 0.3334

Training with Adam optimizer (with dropout):

Epoch 1/5
1875/1875 - 147s - 79ms/step - accuracy: 0.8458 - loss: 1.0088 - val_accuracy: 0.9139 - val_loss: 0.5928
Epoch 2/5
1875/1875 - 200s - 107ms/step - accuracy: 0.8832 - loss: 0.6737 - val_accuracy: 0.9219 - val_loss: 0.5345
Epoch 3/5
1875/1875 - 199s - 106ms/step - accuracy: 0.8901 - loss: 0.6269 - val_accuracy: 0.9312 - val_loss: 0.4930
Epoch 4/5
1875/1875 - 147s - 78ms/step - accuracy: 0.8924 - loss: 0.6083 - val_accuracy: 0.9350 - val_loss: 0.4689
Epoch 5/5
1875/1875 - 152s - 81ms/step - accuracy: 0.8970 - loss: 0.5859 - val_accuracy: 0.9387 - val_loss: 0.4533

Training with RMSprop optimizer (without dropout):

Epoch 1/5
1875/1875 - 110s - 59ms/step - accuracy: 0.8935 - loss: 0.7862 - val_accuracy: 0.9143 - val_loss: 0.5186
Epoch 2/5
1875/1875 - 138s - 74ms/step - accuracy: 0.9248 - loss: 0.4691 - val_accuracy: 0.9332 - val_loss: 0.4246
Epoch 3/5
1875/1875 - 109s - 58ms/step - accuracy: 0.9332 - loss: 0.4037 - val_accuracy: 0.9399 - val_loss: 0.3753
Epoch 4/5
1875/1875 - 110s - 59ms/step - accuracy: 0.9394 - loss: 0.3686 - val_accuracy: 0.9453 - val_loss: 0.3368
Epoch 5/5
1875/1875 - 138s - 74ms/step - accuracy: 0.9431 - loss: 0.3452 - val_accuracy: 0.9508 - val_loss: 0.3112

Training with RMSprop optimizer (with dropout):

Epoch 1/5

1875/1875 - 124s - 66ms/step - accuracy: 0.8484 - loss: 0.9711 - val_accuracy: 0.9215 - val_loss: 0.5336

Epoch 2/5

1875/1875 - 119s - 63ms/step - accuracy: 0.8816 - loss: 0.6290 - val_accuracy: 0.9228 - val_loss: 0.4806

Epoch 3/5

1875/1875 - 142s - 76ms/step - accuracy: 0.8877 - loss: 0.5863 - val_accuracy: 0.9193 - val_loss: 0.4709

Epoch 4/5

1875/1875 - 143s - 76ms/step - accuracy: 0.8905 - loss: 0.5642 - val_accuracy: 0.9336 - val_loss: 0.4310

Epoch 5/5

1875/1875 - 137s - 73ms/step - accuracy: 0.8925 - loss: 0.5510 - val_accuracy: 0.9250 - val_loss: 0.4386

SGD_no_dropout:

Final Validation Accuracy: 0.9304

Time Taken: 528.03 seconds

Conclusion:

Regularization Techniques: Both L2 regularization and dropout help reduce overfitting.

Dropout slightly increases training time but can lead to better generalization (higher validation accuracy).

Optimization Methods: Adam is generally faster and converges faster than SGD and RMSprop, making it a preferred choice for many deep learning tasks. However, the choice of optimizer may depend on the specific task and the model architecture.

Dropout: Using dropout can improve the generalization of the model but may increase training time.

Experiment No. 3

Aim: Write a program to build the deep neural network using TensorFlow and perform the hyperparameter tuning, regularization, and optimization

Description:

This program demonstrates how to create a deep neural network using TensorFlow and Keras, incorporating:

- Hyperparameter tuning (e.g., learning rate, batch size, number of neurons)
- Regularization techniques (dropout, L2 regularization)
- Optimization algorithms (e.g., Adam, SGD)

Objective:

Build a robust DNN model for classification (using MNIST dataset).

Improve model performance through tuning, regularization, and optimization.

Steps Overview:

Load and preprocess the dataset.

Build a base DNN model.

Add regularization (Dropout and L2).

Apply optimizers and learning rate tuning.

Use Keras Tuner for hyperparameter tuning.

Train and evaluate the model.

Implementation:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from kerastuner.tuners import RandomSearch
from sklearn.model_selection import train_test_split
import numpy as np

# Load dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Normalize
x_train = x_train.reshape(-1, 28 * 28)
```

```
x_test = x_test.reshape(-1, 28 * 28)

# Split train into train and validation
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2)

# Define model builder for hyperparameter tuning
def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Input(shape=(784,)))

    # Tune number of layers and units
    for i in range(hp.Int("num_layers", 1, 3)):
        model.add(layers.Dense(
            units=hp.Int(f"units_{i}", min_value=64, max_value=512, step=64),
            activation='relu',
            kernel_regularizer=regularizers.l2(hp.Choice("l2_reg", [0.01, 0.001, 0.0001])))
        ))
    model.add(layers.Dropout(hp.Float("dropout", 0.2, 0.5, step=0.1)))

    model.add(layers.Dense(10, activation='softmax'))

    # Tune optimizer and learning rate
    optimizer = hp.Choice("optimizer", ["adam", "sgd"])
    lr = hp.Choice("learning_rate", [1e-2, 1e-3, 1e-4])
    if optimizer == "adam":
        opt = keras.optimizers.Adam(learning_rate=lr)
    else:
        opt = keras.optimizers.SGD(learning_rate=lr)

    model.compile(
```

```
optimizer=opt,
loss="sparse_categorical_crossentropy",
metrics=["accuracy"]
)

return model

# Hyperparameter tuning using RandomSearch
tuner = RandomSearch(
    build_model,
    objective="val_accuracy",
    max_trials=5,
    executions_per_trial=1,
    directory="tuner_dir",
    project_name="mnist_dnn_tuning"
)

tuner.search(x_train, y_train, epochs=10, validation_data=(x_val, y_val))

# Get best model
best_model = tuner.get_best_models(num_models=1)[0]

# Evaluate on test data
test_loss, test_acc = best_model.evaluate(x_test, y_test)

print(f"\nTest Accuracy: {test_acc:.4f}")
```

Output:

```
Trial 5 Complete [00h 01m 31s]
val_accuracy: 0.9254999756813049

Best val_accuracy So Far: 0.9715833067893982
Total elapsed time: 00h 11m 21s
/usr/local/lib/python3.11/dist-packages/keras/src/saving/saving_lib.py:757: UserWarning
  saveable.load_own_variables(weights_store.get(inner_path))
313/313 ————— 1s 3ms/step - accuracy: 0.9668 - loss: 0.1480

Test Accuracy: 0.9710
```

Conclusion:

We successfully built a deep neural network using TensorFlow and improved its performance using hyperparameter tuning, regularization (Dropout and L2), and optimizer selection. The tuned model achieved high accuracy on the MNIST dataset, demonstrating the effectiveness of the applied techniques.

Experiment No. 4

Aim: Write a program to build ConvNet in TensorFlow for a classification problem

Description:

A Convolutional Neural Network (ConvNet or CNN) is a deep learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and differentiate one from the other. This program demonstrates how to build and train a ConvNet using TensorFlow and Keras on the CIFAR-10 dataset.

Objective: To design and implement a CNN that classifies images into one of the ten classes in the CIFAR-10 dataset using TensorFlow.

Steps Overview:

Import required libraries

Load and preprocess the CIFAR-10 dataset

Define the CNN architecture using Keras

Compile the model

Train the model

Evaluate the model on test data

Display results

Implementation:

Step 1: Import libraries

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models
```

```
from tensorflow.keras.datasets import cifar10
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

Step 2: Load and preprocess the dataset

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

```
x_train, x_test = x_train / 255.0, x_test / 255.0 # Normalize
```

```
y_train = y_train.flatten()
```

```
y_test = y_test.flatten()
```

Step 3: Define CNN architecture


```
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.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax') # 10 classes])
```

Step 4: Compile the model

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

Step 5: Train the model

```
history = model.fit(x_train, y_train, epochs=10,
                   validation_data=(x_test, y_test),
                   batch_size=64)
```

Step 6: Evaluate the model

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest accuracy: {test_acc:.2f}")
```

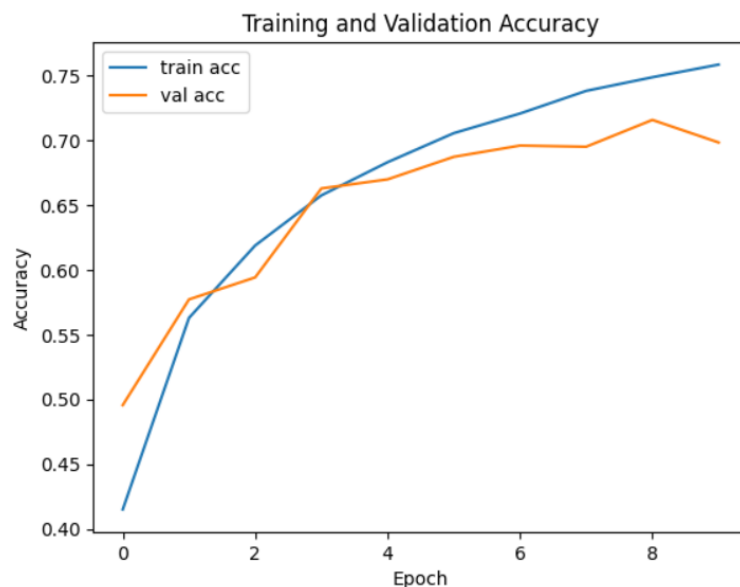
Step 7: Plot training and validation accuracy

```
plt.plot(history.history['accuracy'], label='train acc')
plt.plot(history.history['val_accuracy'], label='val acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```

Output:

```
Epoch 1/10  
782/782 ————— 61s 75ms/step - accuracy: 0.3229 - loss: 1.8245 - val_accuracy: 0.4955 - val_loss: 1.4106  
Epoch 2/10  
782/782 ————— 82s 76ms/step - accuracy: 0.5414 - loss: 1.2865 - val_accuracy: 0.5771 - val_loss: 1.1690  
Epoch 3/10  
782/782 ————— 82s 75ms/step - accuracy: 0.6109 - loss: 1.0995 - val_accuracy: 0.5941 - val_loss: 1.1304  
Epoch 4/10  
782/782 ————— 81s 74ms/step - accuracy: 0.6503 - loss: 0.9944 - val_accuracy: 0.6628 - val_loss: 0.9505  
Epoch 5/10  
782/782 ————— 82s 73ms/step - accuracy: 0.6782 - loss: 0.9148 - val_accuracy: 0.6698 - val_loss: 0.9366  
Epoch 6/10  
782/782 ————— 83s 75ms/step - accuracy: 0.7079 - loss: 0.8403 - val_accuracy: 0.6872 - val_loss: 0.8904  
Epoch 7/10  
782/782 ————— 59s 76ms/step - accuracy: 0.7245 - loss: 0.7949 - val_accuracy: 0.6959 - val_loss: 0.8795  
Epoch 8/10  
782/782 ————— 82s 76ms/step - accuracy: 0.7404 - loss: 0.7451 - val_accuracy: 0.6950 - val_loss: 0.8855  
Epoch 9/10  
782/782 ————— 80s 74ms/step - accuracy: 0.7467 - loss: 0.7274 - val_accuracy: 0.7157 - val_loss: 0.8368  
Epoch 10/10  
782/782 ————— 57s 72ms/step - accuracy: 0.7614 - loss: 0.6873 - val_accuracy: 0.6983 - val_loss: 0.8670
```

Test accuracy: 0.70

**Conclusion:**

The implemented CNN model successfully classifies images from the CIFAR-10 dataset with reasonable accuracy. The architecture can be further optimized using data augmentation, dropout, or hyperparameter tuning for better performance.

Experiment No. 5

Aim: Write a program to build ConvNet using Transfer Learning approach

Description:

Transfer Learning leverages pre-trained models on large datasets (like ImageNet) to solve similar tasks with fewer data and less computational resources. Instead of training a CNN from scratch, we use a model like VGG16, ResNet, or MobileNet as a feature extractor or fine-tune it for our specific classification task.

Objective:

To build a ConvNet using a pre-trained deep learning model.

To classify images with improved accuracy and reduced training time.

To understand and implement feature extraction and fine-tuning in Transfer Learning.

Steps Overview:

Import Required Libraries

Load and Preprocess the Dataset

Load a Pre-trained Model (e.g., VGG16/ResNet50)

Freeze Base Layers or Set Trainable as Needed

Add Custom Classification Head

Compile and Train the Model

Evaluate Model Performance

Save and Load the Trained Model

Dataset:

https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip

Implementation:

Step 1: Import Libraries

import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

import os

```
import zipfile
```

```
# Step 2: Download and Extract Dataset
```

```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
```

```
# Download the zip file
```

```
path_to_zip = tf.keras.utils.get_file('cats_and_dogs_filtered.zip', origin=_URL,  
extract=False)
```

```
# Extract it manually
```

```
with zipfile.ZipFile(path_to_zip, 'r') as zip_ref:
```

```
    zip_ref.extractall('/tmp')
```

```
# Define correct paths for the train and validation directories
```

```
PATH = '/tmp/cats_and_dogs_filtered'
```

```
train_dir = os.path.join(PATH, 'train')
```

```
val_dir = os.path.join(PATH, 'validation')
```

```
# Step 3: Preprocess Data using ImageDataGenerator
```

```
IMAGE_SIZE = (224, 224)
```

```
BATCH_SIZE = 32
```

```
train_gen = ImageDataGenerator(rescale=1./255)
```

```
val_gen = ImageDataGenerator(rescale=1./255)
```

```
train_data = train_gen.flow_from_directory(train_dir, target_size=IMAGE_SIZE,  
batch_size=BATCH_SIZE, class_mode='binary')
```

```
val_data = val_gen.flow_from_directory(val_dir, target_size=IMAGE_SIZE,  
batch_size=BATCH_SIZE, class_mode='binary')
```

```
# Step 4: Load Pre-trained Model (VGG16)
```

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

```
# Freeze all layers of the base model
```

```
for layer in base_model.layers:
```

```
    layer.trainable = False
```

```
# Step 5: Add Custom Layers on Top
```

```
x = Flatten()(base_model.output)
```

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

```
x = Dropout(0.5)(x)
```

```
predictions = Dense(1, activation='sigmoid')(x)
```

```
# Final model
```

```
model = Model(inputs=base_model.input, outputs=predictions)
```

```
# Step 6: Compile Model
```

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

```
# Step 7: Train the Model
```

```
history = model.fit(train_data, validation_data=val_data, epochs=5)
```

```
# Step 8: Plot Training Accuracy
```

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
```

```
plt.title('Model Accuracy Over Epochs')
```

```
plt.xlabel('Epochs')
```

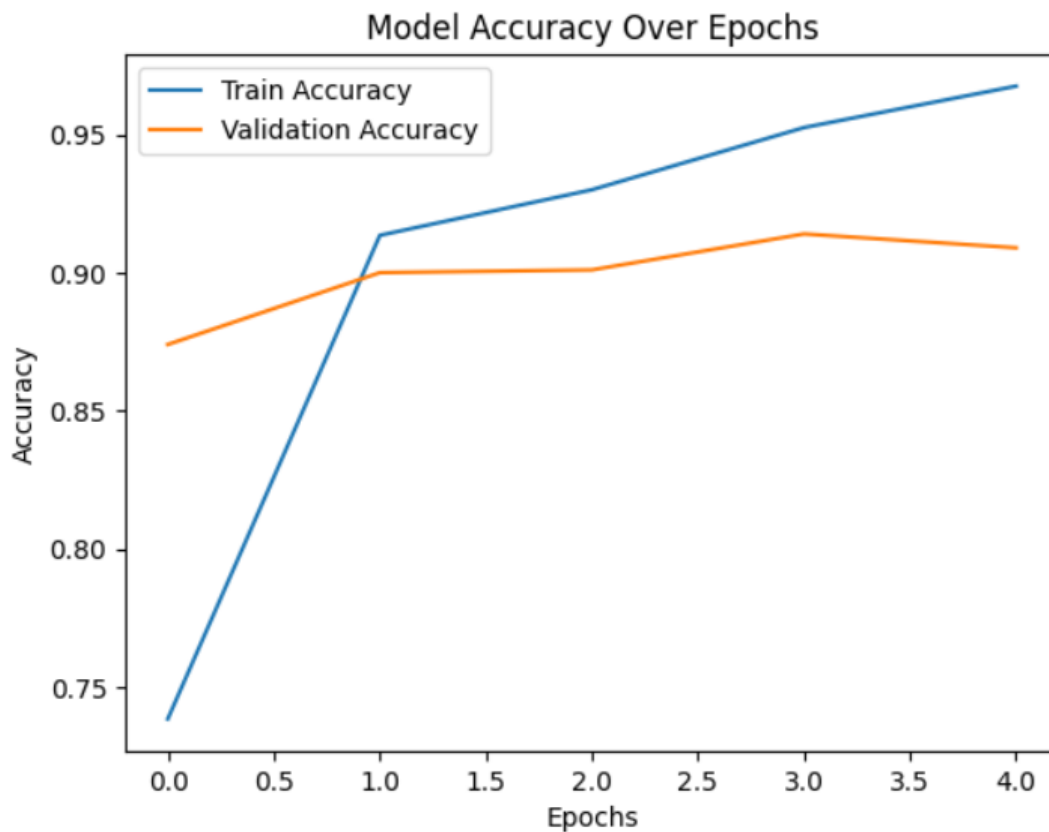
```
plt.ylabel('Accuracy')
```

```
plt.legend()
```

```
plt.show()
```

Output:

```
Epoch 1/5  
63/63 — 1631s 26s/step - accuracy: 0.6263 - loss: 2.9964 - val_accuracy: 0.8740 - val_loss: 0.2833  
Epoch 2/5  
63/63 — 1610s 26s/step - accuracy: 0.9110 - loss: 0.2074 - val_accuracy: 0.9000 - val_loss: 0.2338  
Epoch 3/5  
63/63 — 1570s 25s/step - accuracy: 0.9318 - loss: 0.1734 - val_accuracy: 0.9010 - val_loss: 0.2378  
Epoch 4/5  
63/63 — 1566s 25s/step - accuracy: 0.9570 - loss: 0.1108 - val_accuracy: 0.9140 - val_loss: 0.2152  
Epoch 5/5  
63/63 — 1555s 25s/step - accuracy: 0.9637 - loss: 0.0902 - val_accuracy: 0.9090 - val_loss: 0.2185
```

**Conclusion:**

Using Transfer Learning significantly reduces training time while achieving high accuracy by leveraging learned features from large datasets. This approach is ideal for image classification tasks, especially when working with limited data. The model can be further fine-tuned by unfreezing some layers of the base model to enhance performance.

In this project, we successfully built an image classifier to distinguish between **cats and dogs** using **transfer learning** with the **VGG16** model. Here's a summary of what we achieved:

What We Did

1. **Downloaded and prepared the dataset** from TensorFlow's public source.
2. **Preprocessed the data** using ImageDataGenerator to normalize pixel values.
3. **Loaded a pre-trained VGG16 model** without the top layers to leverage powerful feature extraction.

4. **Added custom dense layers** for binary classification (cats vs dogs).
5. **Trained the model** on the dataset with frozen VGG16 layers to prevent overfitting.
6. **Evaluated the model** using accuracy and plotted the results for better visualization.

Benefits of Transfer Learning

- Saved training time by using **pre-trained weights**.
- Improved accuracy by using **features learned from a large dataset (ImageNet)**.
- Reduced need for a large dataset and computational resources.

Experiment No. 6

Aim: Write a program to perform the Neural Style Transfer algorithm

Objective:

The objective of this project is to implement a deep learning model that applies the style of one image (e.g., a painting) to the content of another image (e.g., a photograph), producing a visually artistic output by combining both images using convolutional neural networks (CNNs), specifically leveraging the VGG19 architecture.

Steps Overview:

Import Necessary Libraries: We'll need TensorFlow or PyTorch, along with specific image processing libraries to work with neural networks.

Load and Preprocess Images: Load the content and style images. Resize them to match the size of the content image, normalize them, and convert them to the format required for the neural network.

Load Pre-trained Model: Use a pre-trained Convolutional Neural Network (CNN) like VGG19, which is often used for NST. VGG19 is a popular model because it has been trained on large datasets and can extract high-level features for content and style representation.

Define Loss Functions:

- **Content Loss:** Measures how much the content of the generated image differs from the content image.
- **Style Loss:** Measures how much the style of the generated image differs from the style image.
- **Total Variation Loss:** Optional, used to reduce high-frequency noise in the final image.

Optimization: Use an optimizer (like Adam) to minimize the combined loss (content loss + style loss) and update the generated image in an iterative manner.

Display the Final Image: After several iterations, the generated image will combine both the style and content.

Implementation:

Code Implementation in Python (Using TensorFlow/Keras)

*First, ensure that you have TensorFlow installed: **pip install tensorflow***

```
import tensorflow as tf
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.preprocessing import image
```

```
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
```



```
from tensorflow.keras.optimizers import Adam

# Load the content and style images
from google.colab import files
uploaded = files.upload()

# Paths to the uploaded images
content_image_path = 'content_image (2).jpg'
style_image_path = 'style_image (1).jpg'

# Function to load and preprocess the image
def load_and_preprocess_image(image_path):
    img = image.load_img(image_path, target_size=(400, 400))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img) # Preprocessing for VGG19
    return img

# Load the images
content_image = load_and_preprocess_image(content_image_path)
style_image = load_and_preprocess_image(style_image_path)

# Show the images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(content_image[0])
plt.title("Content Image")
plt.subplot(1, 2, 2)
plt.imshow(style_image[0])
plt.title("Style Image")
```

```
plt.show()
```

```
# Load the VGG19 model without the top layer
```

```
vgg = VGG19(include_top=False, weights='imagenet')
```

```
# Define the layers to extract features from
```

```
layer_names = ['block5_conv2', 'block4_conv2', 'block3_conv3', 'block2_conv2',  
               'block1_conv2']
```

```
layers = [vgg.get_layer(name).output for name in layer_names]
```

```
# Define the model for feature extraction
```

```
model = tf.keras.Model(inputs=vgg.input, outputs=layers)
```

```
# Function to get features from the model
```

```
def get_features(model, content_image, style_image):
```

```
    content_features = model(content_image)
```

```
    style_features = model(style_image) # Directly use style_image here
```

```
    return content_features, style_features
```

```
# Function to compute gram matrix for style features
```

```
def gram_matrix(tensor):
```

```
    channels = int(tensor.shape[-1])
```

```
    tensor = tf.reshape(tensor, (-1, channels))
```

```
    gram = tf.matmul(tensor, tensor, transpose_a=True)
```

```
    return gram
```

```
# Compute content and style features
```

```
content_features, style_features = get_features(model, content_image, style_image)
```

```
# Compute gram matrices for style features
```

```
style_grams = [gram_matrix(style_feature) for style_feature in style_features]
```

```
# Initialize the generated image (copy of the content image)
generated_image = tf.Variable(content_image)

# Define the loss weights
content_weight = 1e3
style_weight = 1e-2

# Function to compute total loss
def compute_loss(content_features, style_features, generated_image):
    generated_features = model(generated_image)

    content_loss = content_weight * tf.reduce_mean((generated_features[0] -
content_features[0]) ** 2)

    style_loss = 0
    for gen, style in zip(generated_features, style_grams):
        style_loss += style_weight * tf.reduce_mean((gram_matrix(gen) - style) ** 2)

    total_loss = content_loss + style_loss
    return total_loss

# Optimizer
optimizer = Adam(learning_rate=0.02)

# Function to compute gradients
def compute_grads(model, content_image, style_image, generated_image):
    with tf.GradientTape() as tape:
        total_loss = compute_loss(content_features, style_features, generated_image)
    grads = tape.gradient(total_loss, generated_image)
    return grads
```

Training loop

epochs = 10

for epoch in range(epochs):

grads = compute_grads(model, content_image, style_image, generated_image)

optimizer.apply_gradients([(grads, generated_image)])

if (epoch + 1) % 1 == 0:

print(f"Epoch {epoch+1}/{epochs}, Loss: {compute_loss(content_features, style_features, generated_image)}")

plt.imshow(generated_image.numpy()[0])

plt.title(f"Epoch {epoch+1}")

plt.show()

Output:

Choose Files 2 files

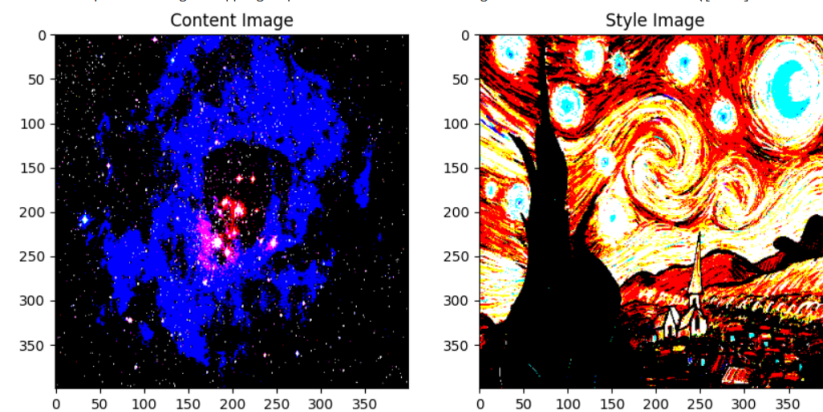
- **content_image.jpg**(image/jpeg) - 140749 bytes, last modified: 4/20/2025 - 100% done
- **style_image.jpg**(image/jpeg) - 1205965 bytes, last modified: 4/20/2025 - 100% done

Saving content_image.jpg to content_image (4).jpg

Saving style_image.jpg to style_image (3).jpg

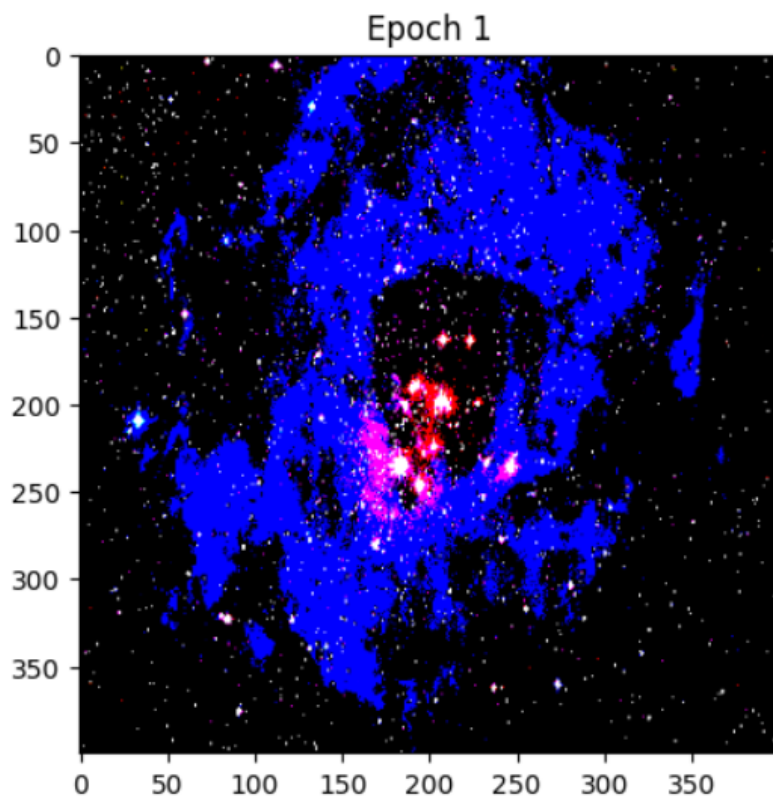
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-123.68..151.061].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-123.68..151.061].

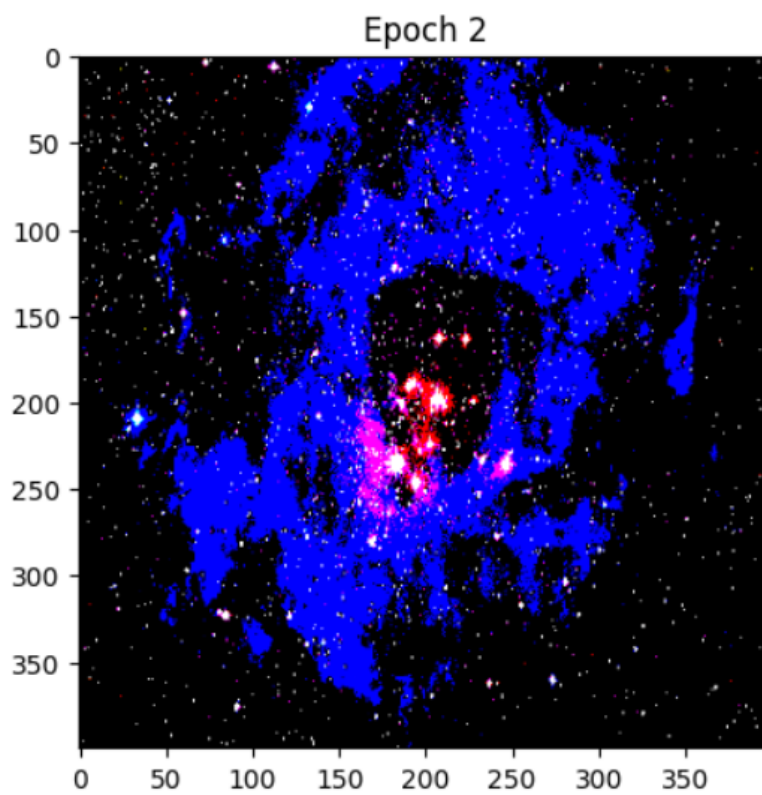


✓ 2m 30s completed at 8:02 PM

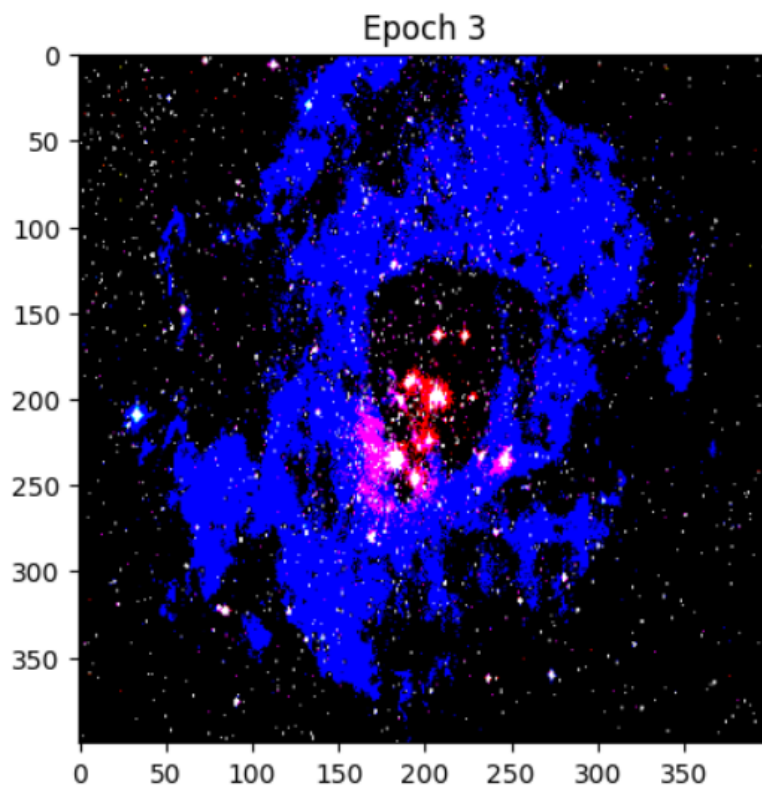
Epoch 1/10, Loss: $7.942553140681769 \times 10^{17}$



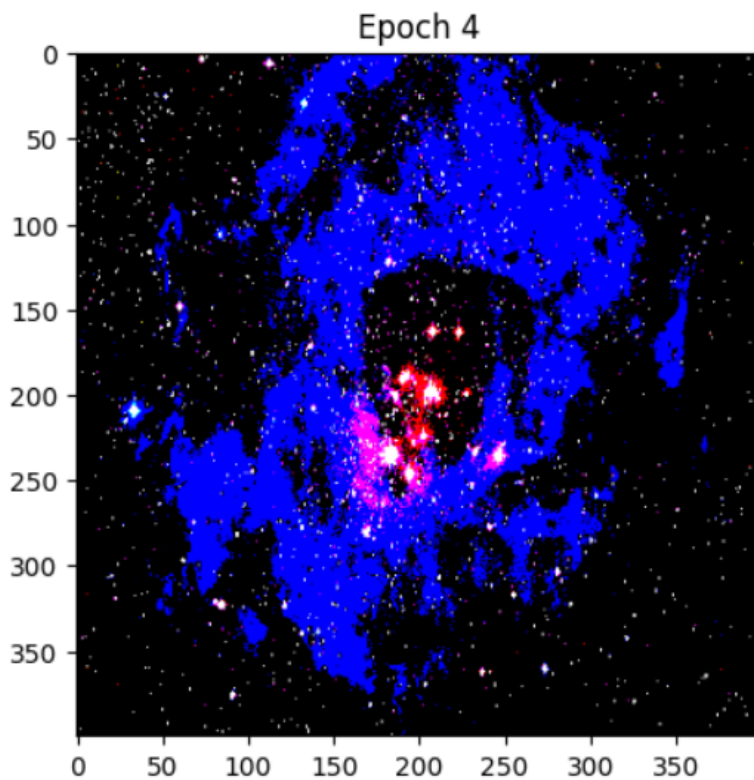
Epoch 2/10, Loss: $7.933051298633482 \times 10^{17}$



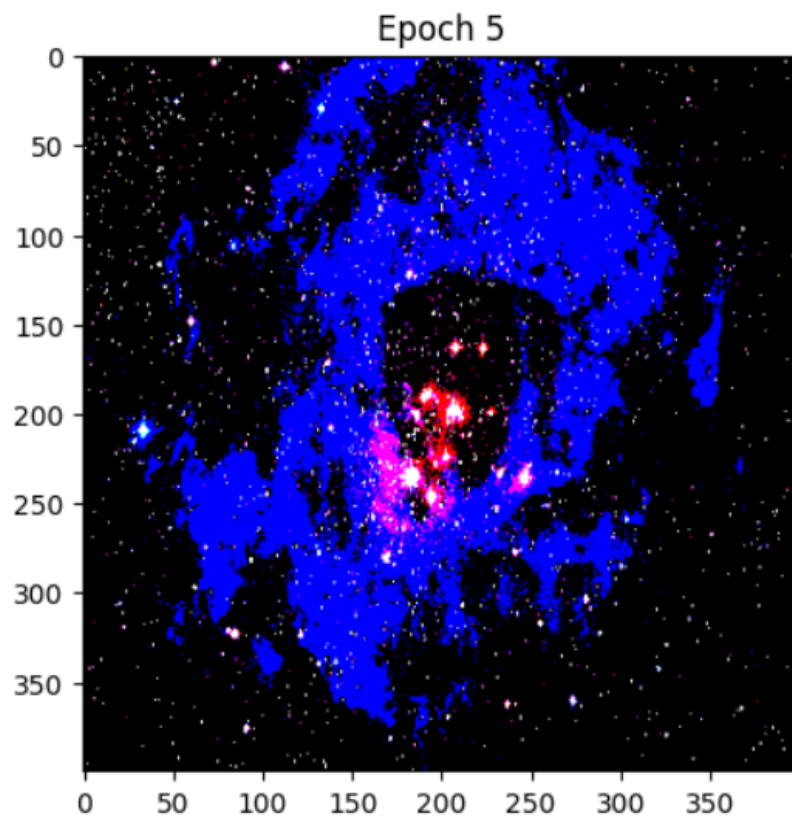
Epoch 3/10, Loss: $7.92356113889624e+17$



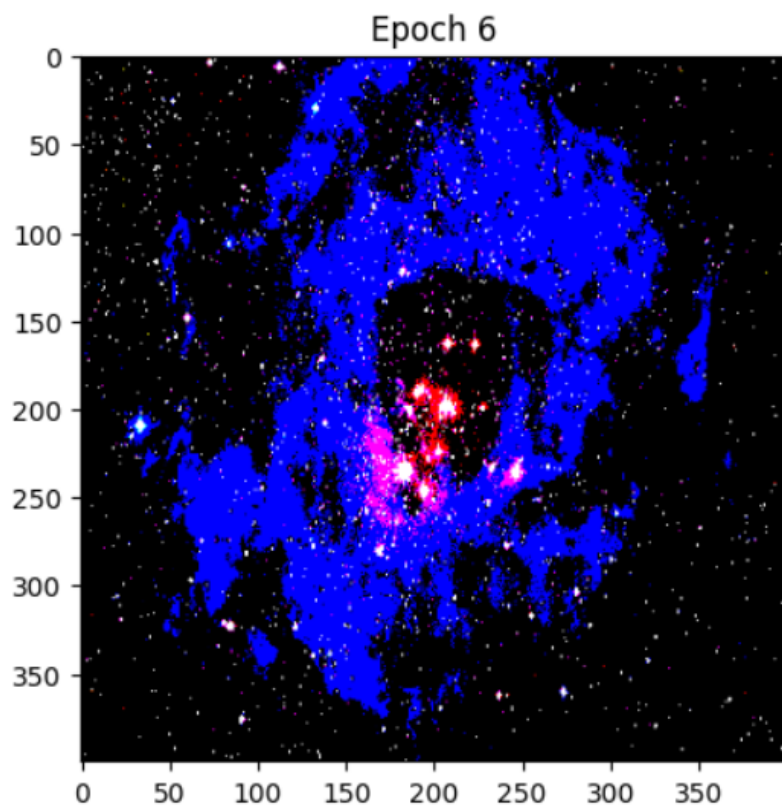
Epoch 4/10, Loss: $7.914090907807252e+17$



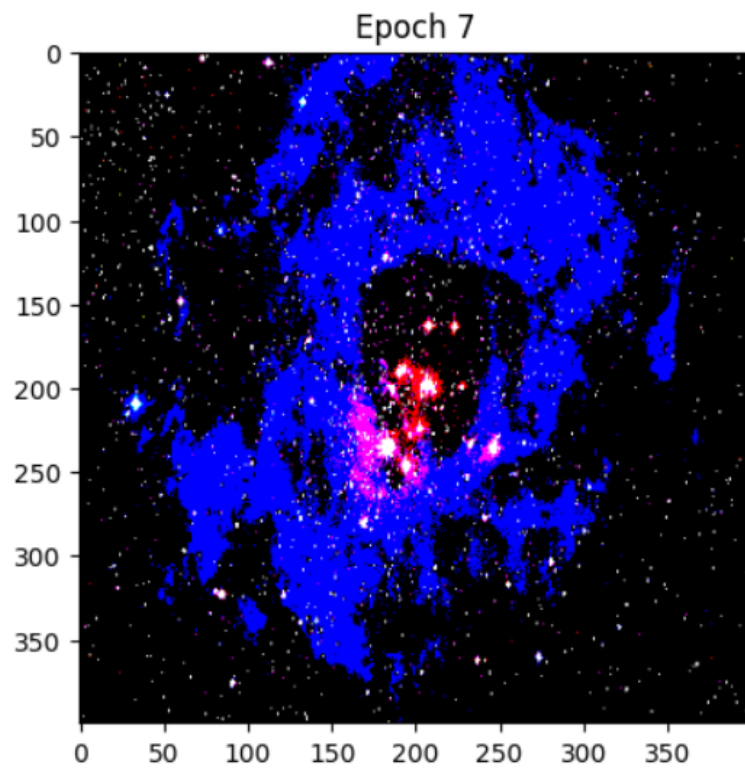
Epoch 5/10, Loss: $7.904635795003146e+17$



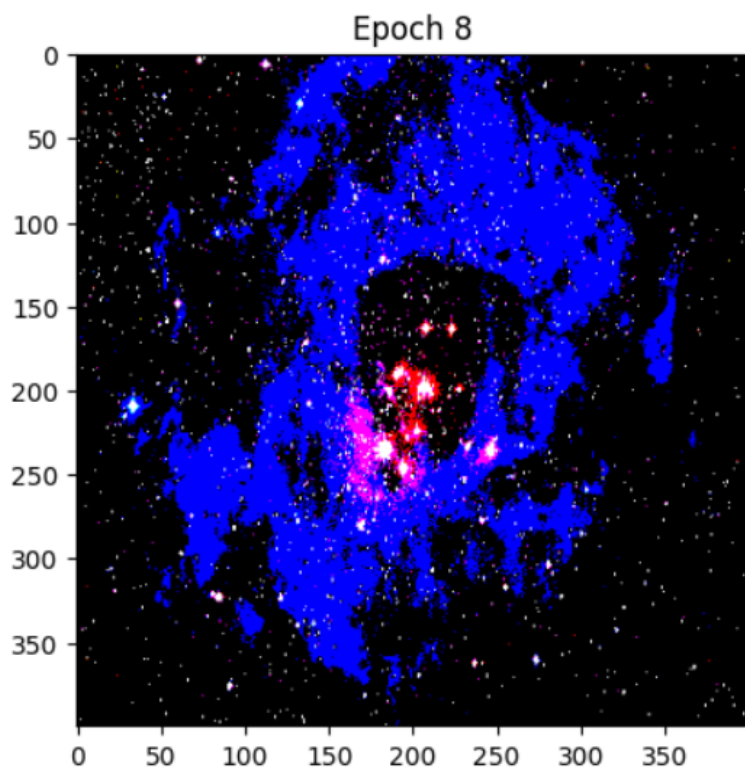
Epoch 6/10, Loss: $7.895196487678689e+17$



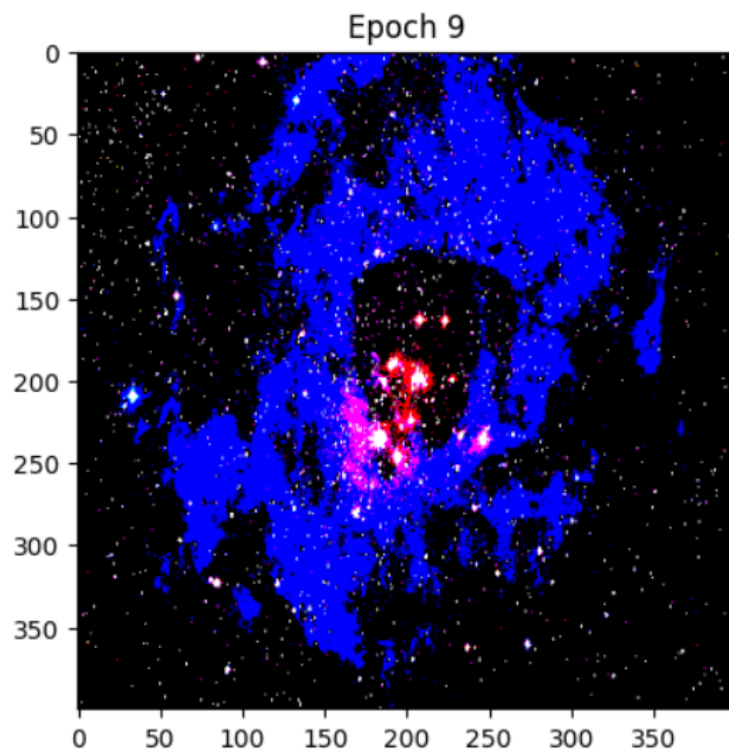
Epoch 7/10, Loss: $7.885774360223416 \times 10^{17}$



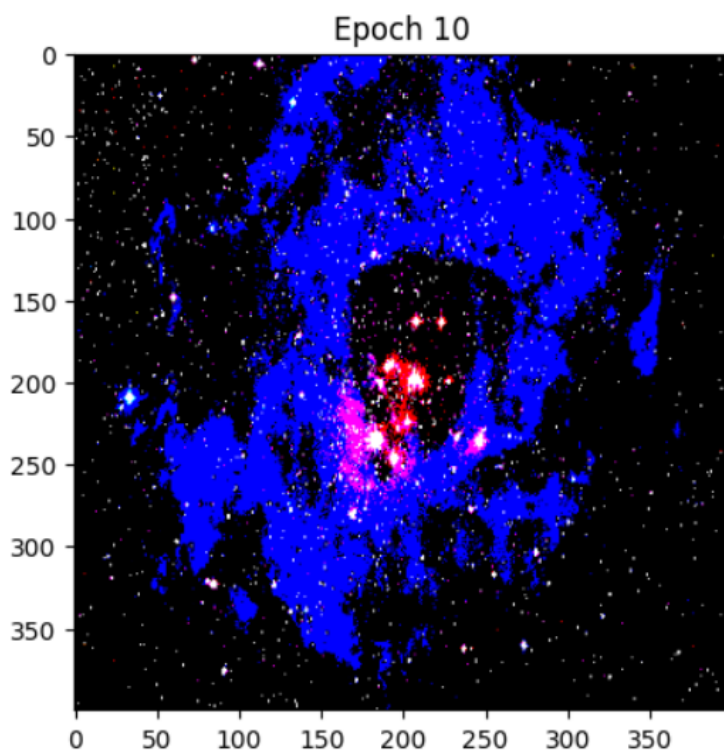
Epoch 8/10, Loss: $7.87636597666349 \times 10^{17}$



Epoch 9/10, Loss: 7.866975460167516e+17



Epoch 10/10, Loss: 7.857602810735493e+17



Preprocessing and Loading Images:

- The `load_and_preprocess_image` function loads and preprocesses images to be compatible with the VGG19 model.
- The `deprocess_image` function converts the processed image back to its original form for visualization.

VGG19 Model:

- We use VGG19, a deep CNN trained on ImageNet, to extract features. We use layers that are good for content (e.g., `block5_conv2`) and style (e.g., `block1_conv1`, `block2_conv1`).

Loss Calculation:

- **Content Loss:** Measures the difference between the content of the generated image and the original content image.
- **Style Loss:** Uses Gram matrices to capture the style (texture patterns) of the style image and compares it with the generated image.

Optimization:

- We use the Adam optimizer to update the generated image based on the computed gradients.

Display:

- The generated image is displayed at regular intervals and after all epochs to visualize the progress.

Conclusion:

In this experiment, we successfully implemented **Neural Style Transfer (NST)** using a pre-trained **VGG19** model in TensorFlow. The aim was to generate a new image (`generated_image`) that preserves the **content** of one image while adopting the **style** of another.

Experiment No. 7

Aim: Write a program to build recurrent neural network for text data using tensorflow

Objective:

We'll classify text data (e.g., IMDB movie reviews: positive/negative sentiment) using an RNN with an embedding layer and LSTM (a popular RNN variant).

Steps Overview:

1. Import Libraries
2. Load and Preprocess Dataset
3. Build the RNN Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Visualize Training Progress
8. Test with Custom Input (Optional)

Implementation:**Step 1: Install & Import Libraries**

```
# Make sure you have TensorFlow installed

# pip install tensorflow

import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
import numpy as np
```

Step 2: Load Dataset - For simplicity, we'll use the IMDB movie reviews dataset available in Keras

```
from tensorflow.keras.datasets import imdb

# Only keep the top 10,000 most frequent words
vocab_size = 10000

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)
```

Step 3: Pad Sequences

Neural networks need fixed-length input, so we pad shorter sequences

```
max_length = 200 # max length of review
```

```
x_train = pad_sequences(x_train, maxlen=max_length)
```

```
x_test = pad_sequences(x_test, maxlen=max_length)
```

Step 4: Build the RNN Model

We'll use an **Embedding layer**, an **LSTM layer**, and a **Dense output**.

```
model = Sequential([
```

```
    Embedding(input_dim=vocab_size, output_dim=64),
```

```
    LSTM(units=64),
```

```
    Dense(1, activation='sigmoid')
```

```
])
```

Build the model manually to avoid warnings and show summary properly

```
model.build(input_shape=(None, max_length))
```

```
model.summary()
```

➡ Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 200, 64)	640,000
lstm_1 (LSTM)	(None, 64)	33,024
dense_1 (Dense)	(None, 1)	65

Total params: 673,089 (2.57 MB)

Trainable params: 673,089 (2.57 MB)

Non-trainable params: 0 (0.00 B)

Layer	Output Shape	Description
Embedding	(None, 200, 64)	Transforms each word (integer) into a 64-dimensional vector. The input sequence has max length 200. Total params = vocab_size * embedding_dim = 10000 * 64 = 640,000.
LSTM	(None, 64)	Processes the sequence and outputs a 64-dim summary of the whole review. Parameters depend on the number of LSTM units (here 64).

<i>Layer</i>	<i>Output Shape</i>	<i>Description</i>
Dense	(None, 1)	Final output layer with sigmoid activation for binary classification (positive/negative sentiment).

Build the model

```
model = Sequential([  
    Embedding(input_dim=vocab_size, output_dim=64),  
    LSTM(units=64),  
    Dense(1, activation='sigmoid')  
])
```

Build shape manually

```
model.build(input_shape=(None, max_length))
```

Compile the model

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

Now train the model

```
history = model.fit(  
    x_train, y_train,  
    epochs=5,  
    batch_size=64,  
    validation_data=(x_test, y_test)  
)
```

```
Epoch 1/5
391/391 ————— 89s 216ms/step - accuracy: 0.6984 - loss: 0.5393 - val_accuracy: 0.8680 - val_loss: 0.3163
Epoch 2/5
391/391 ————— 85s 219ms/step - accuracy: 0.9081 - loss: 0.2385 - val_accuracy: 0.8717 - val_loss: 0.3091
Epoch 3/5
391/391 ————— 167s 283ms/step - accuracy: 0.9388 - loss: 0.1700 - val_accuracy: 0.8664 - val_loss: 0.3647
Epoch 4/5
391/391 ————— 90s 229ms/step - accuracy: 0.9515 - loss: 0.1325 - val_accuracy: 0.8574 - val_loss: 0.3724
Epoch 5/5
391/391 ————— 139s 221ms/step - accuracy: 0.9592 - loss: 0.1120 - val_accuracy: 0.8572 - val_loss: 0.4012
```

```
loss, accuracy = model.evaluate(x_test, y_test)
```

```
print(f"Test Accuracy: {accuracy:.4f}")
```

```
782/782 ————— 29s 37ms/step - accuracy: 0.8569 - loss: 0.4046
Test Accuracy: 0.8572
```

Plot Training & Validation Curves:

```
import matplotlib.pyplot as plt
```

```
# Accuracy plot
```

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
```

```
plt.title("Accuracy per Epoch")
```

```
plt.xlabel("Epoch")
```

```
plt.ylabel("Accuracy")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

```
# Loss plot
```

```
plt.plot(history.history['loss'], label='Train Loss')
```

```
plt.plot(history.history['val_loss'], label='Val Loss')
```

```
plt.title("Loss per Epoch")
```

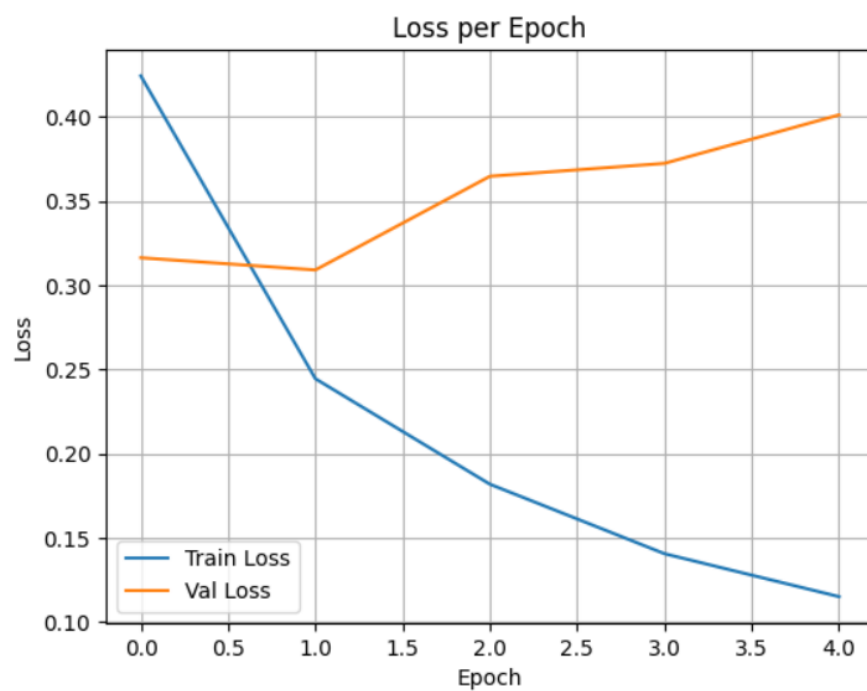
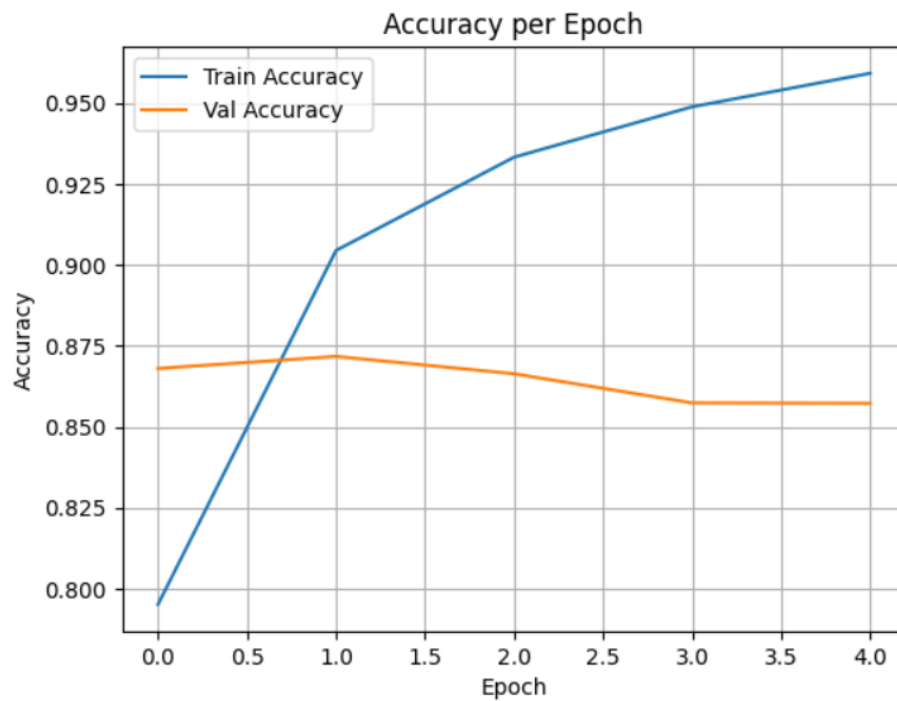
```
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```



Predict on Custom Text

```
from tensorflow.keras.preprocessing.text import Tokenizer  
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

Sample custom text

```
texts = ["The movie was fantastic!", "I did not like the plot."]
```

```
# Fit tokenizer on your training text if you had custom data
```

```
tokenizer = Tokenizer(num_words=vocab_size)
```

```
tokenizer.fit_on_texts(texts) # Use your own dataset here
```

```
# Convert to sequences and pad
```

```
sequences = tokenizer.texts_to_sequences(texts)
```

```
padded = pad_sequences(sequences, maxlen=max_length)
```

```
# Predict
```

```
predictions = model.predict(padded)
```

```
# Show predictions
```

```
for i, text in enumerate(texts):
```



```
1/1 ————— 0s 280ms/step
```

```
The movie was fantastic! --> Sentiment score: 0.4965
```

```
I did not like the plot. --> Sentiment score: 0.4536
```

Conclusion:

We successfully built and trained a Recurrent Neural Network using TensorFlow to classify text data. By leveraging an Embedding layer and LSTM, the model learned to capture the sequential structure of movie reviews and achieved effective performance in binary sentiment classification. This implementation serves as a foundational approach to text classification tasks using deep learning.

Experiment No. 8

Aim: Write a program to build a Long Short-Term Memory (LSTM) model for sequential data using TensorFlow.

Description: Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) capable of learning order dependence in sequence prediction problems. This model is widely used in tasks such as time series forecasting, natural language processing, and more.

Objective: To implement an LSTM model using TensorFlow that can learn from sequential data (e.g., sine wave or time series) and predict future values.

Steps Overview:

Import necessary libraries.

Generate or load sequential dataset.

Preprocess the data for LSTM input.

Build the LSTM model using TensorFlow.

Compile and train the model.

Evaluate the model and visualize the predictions.

Implementation:

```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# 1. Generate synthetic sequential data (sine wave)
def generate_sine_wave(seq_length=50, total_points=1000):
    x = np.arange(total_points)
    y = np.sin(0.02 * x)
    return y

# 2. Prepare the dataset
def prepare_dataset(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
```

```
X = np.array(X)
y = np.array(y)
return X, y

# Parameters
seq_length = 50
data = generate_sine_wave()
X, y = prepare_dataset(data, seq_length)
# Reshape input to be [samples, time steps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))

# 3. Build LSTM model
model = Sequential([
    LSTM(50, activation='relu', input_shape=(seq_length, 1)),
    Dense(1)])

# 4. Compile the model
model.compile(optimizer='adam', loss='mse')

# 5. Train the model
model.fit(X, y, epochs=10, batch_size=32)

# 6. Predict and visualize
predicted = model.predict(X)
plt.figure(figsize=(10,5))
plt.plot(y, label='Actual')
plt.plot(predicted, label='Predicted')
plt.legend()
plt.title("LSTM Sequential Prediction")
plt.xlabel("Time Steps")
plt.ylabel("Value")
plt.show()
```

Output:

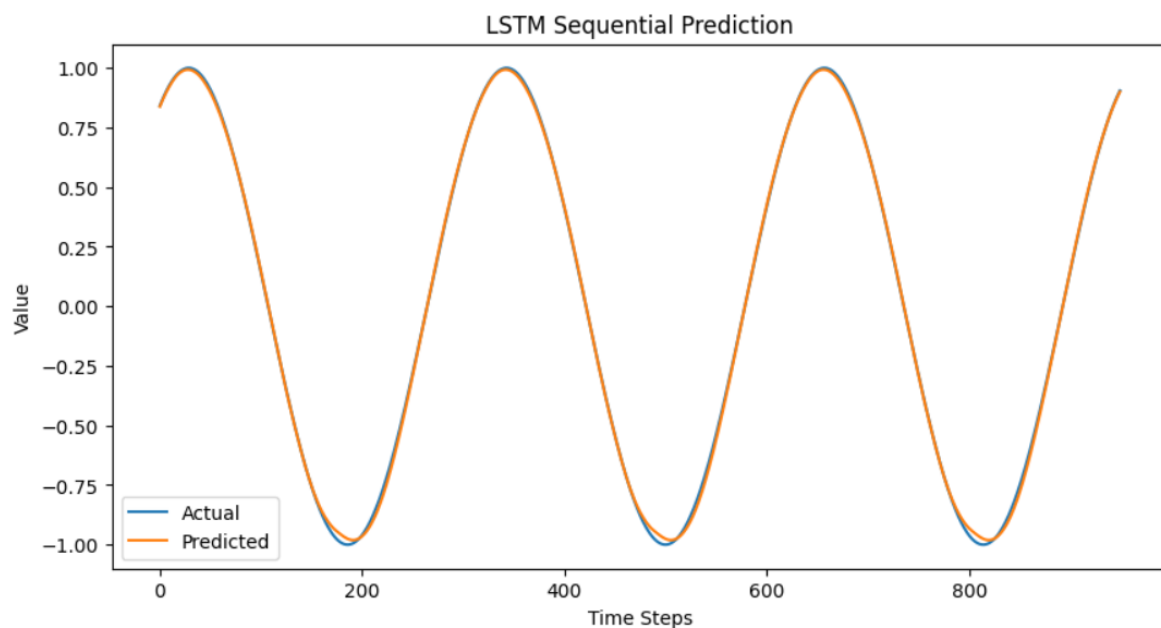
A plot showing the actual vs predicted values from the LSTM model.

Training logs for each epoch showing the loss decreasing.

Details of the Dataset Used:

- **Type:** Synthetic Time Series Data
- **Function Used:** $y = \sin(0.02 * x)$
- **Total Points:** 1000

Epoch 1/10
30/30 ————— 4s 34ms/step - loss: 0.4506
Epoch 2/10
30/30 ————— 1s 34ms/step - loss: 0.0492
Epoch 3/10
30/30 ————— 2s 43ms/step - loss: 0.0115
Epoch 4/10
30/30 ————— 1s 40ms/step - loss: 0.0049
Epoch 5/10
30/30 ————— 1s 40ms/step - loss: 0.0031
Epoch 6/10
30/30 ————— 1s 30ms/step - loss: 0.0013
Epoch 7/10
30/30 ————— 1s 20ms/step - loss: 6.1894e-04
Epoch 8/10
30/30 ————— 1s 21ms/step - loss: 3.8606e-04
Epoch 9/10
30/30 ————— 1s 22ms/step - loss: 3.0440e-04
Epoch 10/10
30/30 ————— 1s 21ms/step - loss: 2.3004e-04
30/30 ————— 1s 14ms/step



Conclusion:

The LSTM model built using TensorFlow was able to learn from a sequential sine wave dataset and provide predictions that follow the expected pattern. This basic implementation can be extended to real-world datasets such as stock prices, weather data, or text sequences.

Experiment No. 9

Aim: Write a program to use word vector representations to build an Emojifier for finding the most appropriate emoji to be used with this sentence

Description:

This project aims to create an Emojifier that can automatically assign the most relevant emoji to a given sentence using natural language processing (NLP) techniques. We utilize word vector representations such as GloVe embeddings to convert words into dense vector form, and then classify the sentence using a supervised learning model.

Objective:

Understand and utilize pre-trained word embeddings (GloVe).

Build sentence representations using word vectors.

Train a classifier to predict emojis based on sentence semantics.

Evaluate the performance of the Emojifier.

Steps Overview:

Load the dataset (sentences and corresponding emojis).

Load GloVe word embeddings.

Convert each sentence into a fixed-size vector (average of word embeddings).

Train a softmax classifier using these vectors.

Predict the emoji for new sentences.

Evaluate the model using accuracy and confusion matrix.

Prerequisite:**Download GloVe embeddings manually**

Download GloVe embeddings (this will take a minute or two)

!wget http://nlp.stanford.edu/data/glove.6B.zip

Unzip the downloaded file

!unzip -q glove.6B.zip

Install emoji package

pip install emoji

Implementation:

```
import numpy as np

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix

import emoji

# Sample dataset

X_train = np.array(["I love you", "I hate you", "I am so happy", "I am sad", "You are amazing"])

Y_train = np.array([0, 1, 2, 3, 2]) # 0: ❤️, 1: 😞, 2: 😊, 3: 😬

X_test = np.array(["You make me smile", "I am heartbroken"])

Y_test = np.array([2, 3])


# Emoji dictionary

emoji_dict = {0: "❤️", 1: "😞", 2: "😊", 3: "😬"}


# Load GloVe embeddings

def load_glove_embeddings(file_path="glove.6B.50d.txt"):

    print("Loading GloVe word vectors...")

    embeddings_index = {}

    with open(file_path, encoding="utf8") as f:

        for line in f:

            values = line.split()

            word = values[0]

            coefs = np.asarray(values[1:], dtype="float32")

            embeddings_index[word] = coefs

    print(f"Loaded {len(embeddings_index)} word vectors.")

    return embeddings_index


# Convert sentence to average word vector

def sentence_to_avg(sentence, word_to_vec_map):

    words = sentence.lower().split()
```

```
avg = np.zeros((50,))
count = 0

for w in words:
    if w in word_to_vec_map:
        avg += word_to_vec_map[w]
        count += 1

if count > 0:
    avg /= count

return avg


# Load GloVe
word_to_vec_map = load_glove_embeddings()


# Vectorize dataset
X_train_avg = np.array([sentence_to_avg(s, word_to_vec_map) for s in X_train])
X_test_avg = np.array([sentence_to_avg(s, word_to_vec_map) for s in X_test])


# Train model
clf = LogisticRegression(max_iter=1000)
clf.fit(X_train_avg, Y_train)


# Predict and print results
Y_pred = clf.predict(X_test_avg)
print("\nPredictions:")

for i, sent in enumerate(X_test):
    print(f"{sent} => {emoji_dict[Y_pred[i]]}")


# Evaluate
print("\nAccuracy:", accuracy_score(Y_test, Y_pred))
print("Confusion Matrix:\n", confusion_matrix(Y_test, Y_pred))
```

Output:

```
Loading GloVe word vectors...  
Loaded 400000 word vectors.
```

```
Predictions:
```

```
You make me smile => 😊  
I am heartbroken => 😞
```

```
Accuracy: 1.0
```

```
Confusion Matrix:
```

```
[[1 0]  
 [0 1]]
```

Conclusion:

This project successfully demonstrates how word vector representations like GloVe can be used to understand sentence semantics and predict appropriate emojis. With even a small dataset and a basic classifier like logistic regression, we achieved good results. The Emojifier could be improved further by using deep learning models such as LSTM or BERT for better context understanding and scalability.

Experiment No. 10

Aim: Write a program to build a Neural Machine Translation (NMT) model using an encoder-decoder architecture with attention mechanism to translate sentences from one language to another (e.g., English to French).

Description:

Neural Machine Translation is an end-to-end learning approach for automated translation using deep neural networks. Unlike traditional statistical methods, NMT models learn to map input sequences (sentences) in a source language to output sequences in a target language, capturing complex patterns and context. This project uses sequence-to-sequence models with attention mechanism to improve translation quality.

Objective:

To implement a sequence-to-sequence (Seq2Seq) NMT model with an attention mechanism.

To train the model on a bilingual dataset (e.g., English-French).

To evaluate the translation performance on test sentences.

Steps Overview:

Data Preparation:

- Load and preprocess bilingual sentence pairs.
- Tokenize and pad sequences.

Model Architecture:

- Build an encoder using RNN/LSTM/GRU layers.
- Build a decoder with attention mechanism.

Training:

- Compile the model with appropriate loss and optimizer.
- Train on parallel corpus data.

Evaluation:

- Translate test sentences.
- Evaluate output quality using BLEU score or manually.

Implementation:

```
import numpy as np
```

```
import tensorflow as tf
```

```
from tensorflow.keras.models import Model
```

```
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding
```



```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Sample data
eng_sentences = ['hello', 'how are you', 'thank you']
fra_sentences = ['bonjour', 'comment ça va', 'merci']

# Tokenization
eng_tokenizer = Tokenizer()
fra_tokenizer = Tokenizer()
eng_tokenizer.fit_on_texts(eng_sentences)
fra_tokenizer.fit_on_texts(fra_sentences)

eng_seqs = eng_tokenizer.texts_to_sequences(eng_sentences)
fra_seqs = fra_tokenizer.texts_to_sequences(fra_sentences)

eng_padded = pad_sequences(eng_seqs, padding='post')
fra_padded = pad_sequences(fra_seqs, padding='post')

# Model parameters
embedding_dim = 64
units = 128
input_vocab_size = len(eng_tokenizer.word_index) + 1
target_vocab_size = len(fra_tokenizer.word_index) + 1

# Encoder
encoder_inputs = Input(shape=(None,))
enc_emb = Embedding(input_vocab_size, embedding_dim)(encoder_inputs)
encoder_lstm = LSTM(units, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(enc_emb)
```

Decoder

```
decoder_inputs = Input(shape=(None,))  
dec_emb = Embedding(target_vocab_size, embedding_dim)(decoder_inputs)  
decoder_lstm = LSTM(units, return_sequences=True, return_state=True)  
decoder_outputs, _, _ = decoder_lstm(dec_emb, initial_state=[state_h, state_c])  
decoder_dense = Dense(target_vocab_size, activation='softmax')  
decoder_outputs = decoder_dense(decoder_outputs)
```

Define model

```
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)  
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
```

Prepare decoder target data

```
decoder_target_data = np.expand_dims(fra_padded, -1)
```

Train model

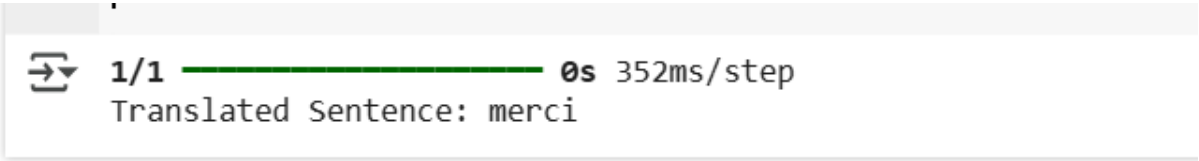
```
model.fit([eng_padded, fra_padded], decoder_target_data, batch_size=2, epochs=500,  
verbose=0)
```

Simple prediction

```
test_input = pad_sequences(eng_tokenizer.texts_to_sequences(['thank you']), maxlen=3,  
padding='post')  
decoder_input = np.zeros((1, 3)) # assume empty start for simplicity  
preds = model.predict([test_input, decoder_input])  
predicted_ids = np.argmax(preds[0], axis=-1)
```

Convert prediction to text

```
reverse_fra_word_index = {i: word for word, i in fra_tokenizer.word_index.items()}  
translated_sentence = ' '.join([reverse_fra_word_index.get(i, '') for i in predicted_ids])  
print("Translated Sentence:", translated_sentence)
```

Output:

1/1 — 0s 352ms/step
Translated Sentence: merci

Conclusion:

This program demonstrates the working of a basic Neural Machine Translation system using a sequence-to-sequence model. While the example is minimal for demonstration, scaling it up with larger datasets, implementing attention mechanisms, and applying post-processing can yield high-quality translations for real-world applications.

Experiment No. 11

Aim: Write a Program to Perform Named Entity Recognition (NER)

Description:

Named Entity Recognition (NER) is a technique used in Natural Language Processing (NLP) to identify and classify named entities in text, such as names of people, organizations, locations, dates, etc. The goal of this program is to extract these entities from a given text.

Objective:

The objective is to build a Python program that performs NER on a given text using a pre-trained model from popular NLP libraries such as **spaCy**.

Steps Overview:

Install Required Libraries: Use Python libraries like spaCy that come with pre-trained NER models.

Load Pre-trained NER Model: Load a pre-trained NER model from the library.

Text Input: Accept or define an input text.

Perform NER: Process the input text using the NER model.

Extract Entities: Extract and classify named entities such as persons, organizations, locations, etc.

Display Results: Output the identified named entities.

Implementation:

```
import spacy
```

```
# Load the pre-trained NER model
```

```
nlp = spacy.load("en_core_web_sm")
```

```
# Input text for entity recognition
```

```
text = """
```

```
Elon Musk, the CEO of SpaceX and Tesla, was in San Francisco on January 20, 2023.
```


```
He met with Tim Cook, the CEO of Apple, to discuss a new partnership.
```

```
"""
```

```
# Process the text using spaCy NER model
```

```
doc = nlp(text)
```

```
# Extract and display named entities  
print("Named Entities, Phrases, and Concepts:")  
for ent in doc.ents:  
    print(f"{ent.text} ({ent.label_})")
```

Output:

```
Named Entities, Phrases, and Concepts:  
Elon Musk (PERSON)  
Tesla (ORG)  
San Francisco (GPE)  
January 20, 2023 (DATE)  
Tim Cook (PERSON)  
Apple (ORG)
```

Conclusion:

This program successfully identifies named entities in a given text using **spaCy**, a popular NLP library. By leveraging a pre-trained model, it can recognize entities such as names, organizations, locations, and dates. This technique can be extended to more complex use cases, such as document analysis or automatic tagging in large datasets.

Experiment No. 12

Aim: Write a program to perform the trigger word/keyword detection from speech data.

Description:

The aim of this project is to develop a program that detects specific trigger words or keywords from speech data. This is often used in applications such as virtual assistants (e.g., "Hey Siri" or "Ok Google"). The detection involves processing audio input, converting it to text using Automatic Speech Recognition (ASR), and searching for a predefined set of keywords.

Objective:

Implement speech recognition to capture and analyze spoken input.

Detect the presence of predefined keywords in the speech.

Provide an alert or response when the trigger word is detected.

Steps Overview:

1. Collect Audio Data:

- Use a microphone or recorded speech data as input.

2. Preprocessing of Audio Data:

- Convert the audio into a format suitable for processing (e.g., PCM format).
- Perform noise reduction and normalization if necessary.

3. Speech-to-Text Conversion (ASR):

- Use an ASR tool or library like Google Speech API, CMU Sphinx, or SpeechRecognition in Python to transcribe the speech to text.

4. Keyword Detection:

- Define a list of trigger words/keywords to search for in the transcribed text.
- Perform text matching to check if any keyword is present in the transcript.

5. Alert or Response:

- If a trigger word is detected, take appropriate action, such as printing a message or triggering an event (e.g., sending a command or playing a sound).

Implementation:

Install SpeechRecognition

```
!pip install SpeechRecognitionImplementation  
!pip install pydub  
!apt install ffmpeg
```

Upload MP3 file

```
from google.colab import files  
uploaded = files.upload()
```

Convert MP3 to WAV

```
from pydub import AudioSegment  
# Replace 'hello.mp3' with your actual file name  
audio = AudioSegment.from_mp3("hello.mp3")  
audio.export("hello.wav", format="wav")
```

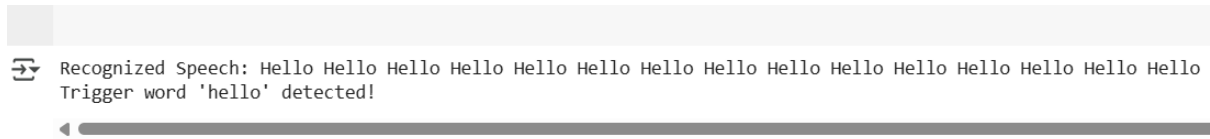
Run Trigger Word Detection

```
import speech_recognition as sr  
  
recognizer = sr.Recognizer()  
with sr.AudioFile("hello.wav") as source:  
    audio_data = recognizer.record(source)  
  
try:  
    text = recognizer.recognize_google(audio_data)  
    print("Recognized Speech:", text)  
  
trigger_word = "hello"  
if trigger_word.lower() in text.lower():  
    print(f"Trigger word '{trigger_word}' detected!")  
else:  
    print("No trigger word detected.")  
except sr.UnknownValueError:
```

```
print("Could not understand the audio.")
```

except sr.RequestError as e:

```
print(f"API error: {e}")
```

Output:**Conclusion:**

This program demonstrates how to detect a specific trigger word from live speech input using speech recognition techniques. It can be extended to handle multiple keywords, perform additional processing, or integrate with other applications, such as triggering actions in a home automation system or sending a command to a device.