1. <u>Demonstrate use of tensorflow and pytorch by implementing simple code in python</u>

TENSORFLOW

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

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

# Normalize the images to the range of 0 to 1

x_train = x_train.astype('float32') / 255.0

x_test = x_test.astype('float32') / 255.0

# Reshape the data to (num_samples, 28 * 28)

x_train = x_train.reshape((x_train.shape[0], 28 * 28))

x_test = x_test.reshape((x_test.shape[0], 28 * 28))

# Create a simple feedforward neural network model

model = models.Sequential([

    layers.Dense(128, activation='relu', input_shape=(28 * 28,)),

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

    layers.Dense(10, activation='softmax')  # Output layer for 10 classes

])

model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)

test_loss, test_accuracy = model.evaluate(x_test, y_test)

print(f'Test accuracy (TensorFlow): {test_accuracy:.4f}')

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

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

plt.title('Training History (TensorFlow)')

plt.xlabel('Epochs')
```

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

Pytorch

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
class FeedForwardNN(nn.Module):
    def __init__(self):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)  # Output layer for 10 classes

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten the input
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)
```

```python
# Instantiate the model, define the loss function and the optimizer
model = FeedForwardNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# Training settings
num_epochs = 5
train_losses = []

# Train the model
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()  # Clear the gradients
        outputs = model(images)  # Forward pass
        loss = criterion(outputs, labels)  # Compute the loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update the weights
        running_loss += loss.item()

    # Record the average loss for the epoch
    average_loss = running_loss / len(train_loader)
    train_losses.append(average_loss)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {average_loss:.4f}')

# Evaluate the model
model.eval()
correct = 0
```

```python
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Test accuracy (PyTorch): {correct / total:.4f}')

# Plot the training loss
plt.plot(train_losses, label='Training Loss', color='blue')
plt.title('Training Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

2. <u>Implement Feedforward neural networks with Keras and TensorFlow MNIST Digit dataset</u>

Importing Dataset from local PC :

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load data from CSV file
data = pd.read_csv(r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\mnist_784_csv.csv")

# Separate features and labels
x = data.iloc[:, :-1].values  # All columns except the last one are pixel values
y = data['class'].values      # The last column is the class label

# Normalize the pixel values
x = x / 255.0

# Reshape x to (number of samples, 28, 28) to match the input shape for the model
x = x.reshape(-1, 28, 28)

# One-hot encode the labels
y = tf.keras.utils.to_categorical(y, 10)

# Split data into training and testing sets
split_index = int(0.8 * len(x))
x_train, x_test = x[:split_index], x[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

# Build a simple feedforward neural network
model = Sequential([
    Flatten(input_shape=(28, 28)),       # Flatten input images to 1D vectors
    Dense(128, activation='relu'),       # Hidden layer with 128 neurons
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')      # Output layer with 10 neurons (one for each class)
])

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

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on test data
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

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

# Display a few random test images with their predictions
num_images = 5
random_indices = np.random.choice(x_test.shape[0], num_images, replace=False)

for i, idx in enumerate(random_indices):
    test_image = x_test[idx]
    true_label = np.argmax(y_test[idx])

    # Predict the label for the test image
    test_image_reshaped = np.expand_dims(test_image, axis=0)  # Reshape for prediction
    predicted_label = np.argmax(model.predict(test_image_reshaped))

    # Plot the image and prediction
    plt.subplot(1, num_images, i + 1)
    plt.imshow(test_image, cmap='gray')
    plt.title(f"True: {true_label}\nPred: {predicted_label}")
    plt.axis('off')
plt.show()
```

Importing Dataset Internet :

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers, models
import tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)

model = Sequential([
    Flatten(input_shape = (28, 28)),
    Dense(256, activation = ‘relu’),
    Dense(128, activation = ‘relu’),
    Dense(64, activation = ‘relu’),
    Dense(10, activation = ‘softmax’)
```

```
])
model.compile(optimizer = ‘adam’, loss = ‘categorical_crossentropy’, metrics = [’accuracy’])

history = model.fit(x_train, y_train, epochs = 10, batch_size=32, validation_data = (x_test, y_test))
test_loss, test_accuracy = model.evaluate(x_test, y_test)

print(f’Test accuracy: {test_accuracy: .4f}’)
plt.plot(history.history[’accuracy’])
plt.plot(history.history[’val_accuracy’])
plt.title(’Model Accuracy’)
plt.xlabel(’Epoch’)
plt.ylabel(’Accuracy’)
plt.legend([’Train’, ‘Test’], loc = ‘upper left’ )
plt.show()

random_index = np.random.randint(0, x_test.shape[0])
test_image = x_test[random_index]
true_label = np.argmax(y_test[random_index])
test_image_reshaped = np.expand_dims(test_image, axis = 0)
predicted_probabilities = model.predict(test_image_reshaped)
predicted_label = np.argmax(predicted_probabilities)

plt.imshow(test_image, cmap = ‘gray’)
plt.title(f”True Label: {true_label}, Predicted Label: {predicted_label}”)
plt.axis(’off’)
plt.show()
```

3. Implement Feedforward neural networks with Keras and TensorFlow CIFAR dataset

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix, classification_report

import seaborn as sns


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

train_data = pd.read_csv('path_to_your_training_file.csv')

x_train = train_data.iloc[:, :-1].values  # All columns except the last one are pixel values

y_train = train_data['label'].values      # The last column is the label

test_data = pd.read_csv('path_to_your_testing_file.csv')

x_test = test_data.iloc[:, :-1].values

y_test = test_data['label'].values


# Normalize the pixel values

x_train = x_train / 255.0

x_test = x_test / 255.0


# Reshape the data to match CIFAR-10 image dimensions (32, 32, 3)

x_train = x_train.reshape(-1, 32, 32, 3)

x_test = x_test.reshape(-1, 32, 32, 3)


# One-hot encode the labels

y_train_encoded = tf.keras.utils.to_categorical(y_train, 10)

y_test_encoded = tf.keras.utils.to_categorical(y_test, 10)
```

```
model = Sequential([
    Flatten(input_shape=(32, 32, 3)),  # Flatten 32x32x3 images
    Dense(512, activation='relu'),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # 10 output classes for CIFAR-10
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x_train, y_train_encoded, epochs=10, batch_size=32, validation_split=0.2)

test_loss, test_accuracy = model.evaluate(x_test, y_test_encoded)
print(f'Test accuracy: {test_accuracy:.4f}')

plt.figure(figsize=(12, 4))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Function to visualize a random prediction
def visualize_random_prediction(images, true_labels, model):
    random_index = np.random.randint(0, len(images))
    image = images[random_index]
    true_label = true_labels[random_index]
```

```
# Prepare the image for prediction
    image_for_prediction = np.expand_dims(image, axis=0)
    prediction = model.predict(image_for_prediction)

    # Get class names for true and predicted labels
    true_class = class_names[np.argmax(true_label)]
    predicted_class = class_names[np.argmax(prediction)]

    # Plot the image and prediction
    plt.figure(figsize=(4, 4))
    plt.imshow(image)
    plt.title(f"True Label: {true_class}\nPredicted Label: {predicted_class}",
              color='green' if true_class == predicted_class else 'red')
    plt.axis('off')
    plt.show()

# Visualize a random test image and its prediction
visualize_random_prediction(x_test, y_test_encoded, model)

# Function to plot confusion matrix and classification report
def plot_confusion_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
```

```
    plt.tight_layout()
    plt.show()

# Predict on the test set and display confusion matrix
predictions = model.predict(x_test)
y_pred_classes = np.argmax(predictions, axis=1)
y_true_classes = y_test  # Use original test labels for evaluation

plot_confusion_matrix(y_true_classes, y_pred_classes)

# Print classification report
print("\nClassification Report:")
print(classification_report(y_true_classes, y_pred_classes, target_names=class_names))
```

4. Build image classification model using CNN on fashion MNIST dataset.

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.optimizers import Adam

train_data = pd.read_csv(r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\fashion-mnist_train.csv\fashion-mnist_train.csv")
test_data = pd.read_csv(r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\fashion-mnist_test.csv\fashion-mnist_test.csv")

# Prepare the data
X_train = train_data.iloc[:, 1:].values  # Get pixel values
y_train = train_data.iloc[:, 0].values    # Get labels
X_test = test_data.iloc[:, 1:].values     # Get pixel values
y_test = test_data.iloc[:, 0].values      # Get labels

# Normalize the data
X_train = X_train / 255.0
X_test = X_test / 255.0

# Reshape the data to match the model input
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Define the model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
# Train the model
history = model.fit(X_train, y_train, epochs=30, validation_split=0.2)

# Plot training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')

# Predictions and classification report
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred_classes))

def show_random_prediction(X_test, y_test, model):
    random_index = random.randint(0, len(X_test) - 1)
    image = X_test[random_index]
    true_label = y_test[random_index]
    image_reshaped = np.expand_dims(image, axis=0)

    # Predict the class of the image
    prediction = model.predict(image_reshaped)
    predicted_label = np.argmax(prediction)

    plt.imshow(image.squeeze(), cmap='gray')
    plt.title(f"True Label: {true_label}\nPredicted Label: {predicted_label}")
    plt.axis('off')
    plt.show()

# Call the function to display a random prediction
show_random_prediction(X_test, y_test, model)
```

5. Build image classification model using CNN on pneumonia X RAY IMAGE dataset.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
import os

train_dir = ‘’
val_dir = ‘’
test_dir = ‘’

# Image data generators for loading images in batches
train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Generate batches of image data from directories
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),  # Resize images to 150x150 pixels
    batch_size=32,
    class_mode='binary'  # Binary classification for 'Normal' and 'Pneumonia'
)

val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)

model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
```

```
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid')  # Sigmoid for binary classification
])

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

history = model.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=10,
    validation_data=val_generator,
    validation_steps=len(val_generator)
)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Model Accuracy')
plt.tight_layout()
plt.show()

test_loss, test_accuracy = model.evaluate(test_generator)
print(f'Test accuracy: {test_accuracy:.4f}')

from tensorflow.keras.preprocessing import image

def predict_image(img_path, model):
    img = image.load_img(img_path, target_size=(150, 150))
    img_array = image.img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)  # Add batch dimension

    prediction = model.predict(img_array)
    class_label = 'Pneumonia' if prediction[0] > 0.5 else 'Normal'
    print(f"Prediction: {class_label}")

    plt.imshow(img)
    plt.title(f"Predicted Label: {class_label}")
    plt.axis('off')
    plt.show()

# Call the function with the path to a test image
predict_image('')
```

6. <u>Build image classification model using CNN on FOOD dataset.</u>
7. <u>Build Brain tumor classification model with CNN</u>

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

from tensorflow.keras.callbacks import EarlyStopping

import numpy as np

from tensorflow.keras.preprocessing import image

import matplotlib.pyplot as plt


train_dir = r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\brain tumour\Training"

test_dir = r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\brain tumour\Testing"


train_datagen = ImageDataGenerator(

    rescale=1.0/255,

    rotation_range=30,

    width_shift_range=0.2,

    height_shift_range=0.2,

    shear_range=0.2,

    zoom_range=0.2,

    horizontal_flip=True,

    vertical_flip=True,

    fill_mode='nearest',

    validation_split=0.2

)

test_datagen = ImageDataGenerator(rescale=1.0/255)

train_generator = train_datagen.flow_from_directory(

    train_dir,

    target_size=(150, 150),
```

```
    batch_size=32,
    class_mode='sparse',
    subset='training'
)
val_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='sparse',
    subset='validation'
)
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='sparse'
)
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 3)),
    layers.MaxPooling2D(2, 2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Conv2D(128, (3,3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(4, activation='softmax')
])
```

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
history = model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=30,
    callbacks=[early_stopping]
)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
plt.show()

import numpy as np

from tensorflow.keras.preprocessing import image

import matplotlib.pyplot as plt


# Define a function to predict a single image

def predict_single_image(model, img_path, class_labels):

    # Load the image with target size as expected by the model

    img = image.load_img(img_path, target_size=(150, 150))


    # Convert the image to an array and rescale it

    img_array = image.img_to_array(img) / 255.0

    img_array = np.expand_dims(img_array, axis=0)  # Expand dimensions for batch shape


    # Make a prediction

    prediction = model.predict(img_array)

    predicted_class = class_labels[np.argmax(prediction)]

    confidence = np.max(prediction)


    # Display the image with prediction

    plt.imshow(img)

    plt.title(f"Predicted: {predicted_class} ({confidence:.2f})")

    plt.axis('off')

    plt.show()

# Class labels in order (change these if necessary)

class_labels = ['no_tumor', 'pituitary_tumor', 'glioma', 'meningioma']

#Example

predict_single_image(model, r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\brain
tumour\Testing\pituitary_tumor\image(45).jpg", class_labels)
```

8. Build Recurrent Neural Network by using the numpy library

```
import numpy as np
import matplotlib.pyplot as plt
```

Dataset Creation

```
def create_dataset():
    data = "hello world hello world hello world"
    chars = list(set(data))
    char_to_idx = {ch: i for i, ch in enumerate(chars)}
    idx_to_char = {i: ch for i, ch in enumerate(chars)}

    # Prepare training data
    X = []  # Input sequences
    y = []  # Target sequences

    for i in range(len(data) - 1):
        input_char = np.zeros((len(chars)))
        input_char[char_to_idx[data[i]]] = 1
        target_char = np.zeros((len(chars)))
        target_char[char_to_idx[data[i + 1]]] = 1

        X.append(input_char)
        y.append(target_char)

    return np.array(X), np.array(y), char_to_idx, idx_to_char, data
```

Model Parameters

```
class SimpleRNN:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        # Initialize weights
        self.Wxh = np.random.randn(hidden_size, input_size) * 0.01   # Input to hidden
        self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01  # Hidden to hidden
        self.Why = np.random.randn(output_size, hidden_size) * 0.01  # Hidden to output

        # Initialize biases
        self.bh = np.zeros((hidden_size, 1))  # Hidden bias
        self.by = np.zeros((output_size, 1))  # Output bias

        self.learning_rate = learning_rate

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def tanh(self, x):
        return np.tanh(x)

    def tanh_derivative(self, x):
        return 1 - np.tanh(x)**2
```

```
    def forward(self, inputs):
        # Initialize lists to store states
        self.hidden_states = []
        self.outputs = []
        h_prev = np.zeros((self.Whh.shape[0], 1))  # Initial hidden state

        # Forward pass for each time step
        for x in inputs:
            # Convert input to column vector
            x = x.reshape(-1, 1)

            # Calculate hidden state
            h = self.tanh(np.dot(self.Wxh, x) + np.dot(self.Whh, h_prev) + self.bh)

            # Calculate output
            y = self.sigmoid(np.dot(self.Why, h) + self.by)

            # Store states for backpropagation
            self.hidden_states.append(h)
            self.outputs.append(y)
            h_prev = h

        return self.outputs

    def backward(self, inputs, targets, outputs, hidden_states):
        # Initialize gradients
        dWxh = np.zeros_like(self.Wxh)
        dWhh = np.zeros_like(self.Whh)
        dWhy = np.zeros_like(self.Why)
        dbh = np.zeros_like(self.bh)
        dby = np.zeros_like(self.by)

        dh_next = np.zeros_like(hidden_states[0])

        # Backpropagate through time
        for t in reversed(range(len(outputs))):
            dy = outputs[t] - targets[t].reshape(-1, 1)

            # Gradients for Why and by
            dWhy += np.dot(dy, hidden_states[t].T)
            dby += dy

            # Gradient for hidden state
            dh = np.dot(self.Why.T, dy) + dh_next

            # Gradient through tanh
            dh_raw = self.tanh_derivative(hidden_states[t]) * dh

            dbh += dh_raw
            dWxh += np.dot(dh_raw, inputs[t].reshape(1, -1))
            dWhh += np.dot(dh_raw, hidden_states[t-1].T) if t > 0 else np.dot(dh_raw,
np.zeros_like(hidden_states[0]).T)
```

```
            dh_next = np.dot(self.Whh.T, dh_raw)

        # Clip gradients to prevent exploding gradients
        for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
            np.clip(dparam, -5, 5, out=dparam)

        # Update weights and biases
        self.Wxh -= self.learning_rate * dWxh
        self.Whh -= self.learning_rate * dWhh
        self.Why -= self.learning_rate * dWhy
        self.bh -= self.learning_rate * dbh
        self.by -= self.learning_rate * dby
```

Training and History

```
def train_and_demonstrate():
    # Create dataset
    X, y, char_to_idx, idx_to_char, original_data = create_dataset()

    # Initialize RNN
    input_size = len(char_to_idx)
    hidden_size = 50
    output_size = len(char_to_idx)
    rnn = SimpleRNN(input_size, hidden_size, output_size)

    # Training loop
    epochs = 100
    losses = []

    print("Training the RNN...")
    print("Original sequence:", original_data)
    print("\nTraining Progress:")

    for epoch in range(epochs):
        # Forward pass
        outputs = rnn.forward(X)

        # Calculate loss (mean squared error)
        loss = np.mean([(output - target.reshape(-1, 1))**2 for output, target in zip(outputs, y)])
        losses.append(loss)

        # Backward pass
        rnn.backward(X, y, outputs, rnn.hidden_states)

        if epoch % 20 == 0 or epoch == epochs - 1:
            print(f'Epoch {epoch}, Loss: {loss:.4f}')
```

Demonstration and Visualization

```
    print("\nDemonstrating predictions for each character in sequence:")
    for i in range(len(original_data) - 1):
        input_char = original_data[i]
        actual_next_char = original_data[i + 1]
```

```
        # Prepare input
        input_vector = np.zeros((len(char_to_idx)))
        input_vector[char_to_idx[input_char]] = 1

        # Get prediction
        output = rnn.forward([input_vector])[0]
        predicted_char = idx_to_char[np.argmax(output)]

        print(f"Input: '{input_char}' → Predicted next: '{predicted_char}' (Actual: '{actual_next_char}')")

    # Plot training loss
    plt.figure(figsize=(10, 5))
    plt.plot(losses)
    plt.title('Training Loss over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.show()

    return rnn, char_to_idx, idx_to_char
```

Run the Demonstration

```
rnn, char_to_idx, idx_to_char = train_and_demonstrate()
```

9. Implement simple autoencoder to reconstruct MNIST digits. Add sparsity constraint on the encoded representations

```
import numpy as np
import pandas as pd  # Import pandas for data loading
import tensorflow as tf
from tensorflow.keras import layers, regularizers, models
import matplotlib.pyplot as plt

data = pd.read_csv(r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\mnist_784_csv.csv")
x_data = data.iloc[:, 1:].values  # Assuming the first column is labels and the rest are pixel values

# Normalize the pixel values
x_data = x_data.astype('float32') / 255.

# Assuming each image is 28x28
num_samples = x_data.shape[0]  # Get the number of samples
x_data = np.reshape(x_data, (num_samples, 28, 28, 1))  # Reshape to (num_samples, 28, 28, 1)

# Split into training and testing datasets
x_train, x_test = x_data[:int(len(x_data) * 0.8)], x_data[int(len(x_data) * 0.8):]

# Parameters
input_shape = (28, 28, 1)  # Change this according to your data's shape
encoding_dim = 128
sparsity_penalty = 1e-5  # L1 sparsity regularization parameter

# Define encoder
def build_encoder(input_shape, encoding_dim, sparsity_penalty):
    inputs = layers.Input(shape=input_shape)
    x = layers.Flatten()(inputs)
    x = layers.Dense(256, activation='relu')(x)
    x = layers.BatchNormalization()(x)

    # Apply L1 regularization to encourage sparsity
    encoded = layers.Dense(encoding_dim,
                           activation='relu',
                           activity_regularizer=regularizers.L1(sparsity_penalty))(x)
    return models.Model(inputs, encoded, name="encoder")

# Define decoder
def build_decoder(encoding_dim, original_shape):
    encoded_input = layers.Input(shape=(encoding_dim,))
    x = layers.Dense(256, activation='relu')(encoded_input)
    x = layers.BatchNormalization()(x)
    x = layers.Dense(np.prod(original_shape), activation='sigmoid')(x)
    decoded = layers.Reshape(original_shape)(x)

    return models.Model(encoded_input, decoded, name="decoder")

# Build autoencoder
encoder = build_encoder(input_shape, encoding_dim, sparsity_penalty)
```

```
decoder = build_decoder(encoding_dim, input_shape)
autoencoder = models.Model(encoder.input, decoder(encoder.output), name="autoencoder")

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

# Train autoencoder
history = autoencoder.fit(x_train, x_train,
                    epochs=30,
                    batch_size=256,
                    shuffle=True,
                    validation_data=(x_test, x_test))

# Generate reconstructions
decoded_imgs = autoencoder.predict(x_test)

# Visualize results
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title('Original')
    plt.axis('off')

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.title('Reconstructed')
    plt.axis('off')

plt.show()

# Plot training history
plt.figure(figsize=(10, 4))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

10. <u>Use Autoencoder to implement anomaly detection on credit card dataset</u>

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

data = pd.read_csv(r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\creditcard.csv")

# Separate features and labels
X = data.drop('Class', axis=1)
y = data['Class']

# Normalize the feature data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Filter normal transactions (Class == 0) for training
X_train_normal = X_train[y_train == 0]
print(f"Number of normal transactions in training set: {X_train_normal.shape[0]}")

input_dim = X_train_normal.shape[1]

autoencoder = models.Sequential([
    layers.Input(shape=(input_dim,)),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(8, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(input_dim, activation='sigmoid')
])

autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder on normal transactions
history = autoencoder.fit(X_train_normal, X_train_normal,
                          epochs=50,
                          batch_size=256,
                          validation_split=0.2,
                          shuffle=True)

# Calculate reconstruction error on the test set
X_test_predictions = autoencoder.predict(X_test)
mse = np.mean(np.square(X_test - X_test_predictions), axis=1)
```

```python
# Set a threshold for anomaly detection (95th percentile of MSE on normal transactions)
threshold = np.percentile(mse[y_test == 0], 95)
print(f"Threshold for anomaly detection: {threshold}")

# Classify anomalies
y_pred = (mse > threshold).astype(int)

# Evaluate the model
print("Classification Report:")
print(classification_report(y_test, y_pred))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Visualization of reconstruction error distribution
plt.figure(figsize=(10, 6))
plt.hist(mse[y_test == 0], bins=50, alpha=0.6, label='Normal')
plt.hist(mse[y_test == 1], bins=50, alpha=0.6, label='Fraud')
plt.axvline(threshold, color='r', linestyle='--', label='Threshold')
plt.title("Reconstruction Error Distribution")
plt.xlabel("Reconstruction error")
plt.ylabel("Frequency")
plt.legend()
plt.show()

# Show reconstruction error and prediction for a few samples
sample_indices = [0, 1, 2, 3, 4]  # Choose some random test indices for demonstration
for index in sample_indices:
    sample = X_test[index]
    reconstruction = autoencoder.predict(sample.reshape(1, -1))
    error = np.mean(np.square(sample - reconstruction))
    prediction = "Fraud" if error > threshold else "Normal"
    print(f"Sample {index}:")
    print(f"  Reconstruction Error: {error}")
    print(f"  Prediction: {prediction} (Threshold: {threshold})")
```

#OUTPUT

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.95      0.97     56864
           1       0.03      0.90      0.06        98

    accuracy                           0.95     56962
   macro avg       0.51      0.92      0.52     56962
weighted avg       1.00      0.95      0.97     56962
```

- **Precision**:

- For class 0 (normal transactions), precision is 1.00, meaning that when the model predicts a transaction is normal, it is correct 100% of the time.
- For class 1 (fraudulent transactions), precision is only 0.03. This means that of all the transactions predicted as fraudulent, only 3% were actually fraudulent. This indicates a high number of false positives.

- **Recall**:

  - For class 0, recall is 0.95, meaning the model correctly identifies 95% of the actual normal transactions.
  - For class 1, recall is 0.90, indicating that 90% of the fraudulent transactions are correctly identified by the model. This shows that the model is effective at identifying actual fraud cases, despite the low precision.

- **F1-Score**: This is the harmonic mean of precision and recall.

  - The F1-score for normal transactions is quite high (0.97), indicating that the model performs very well in detecting normal transactions.
  - The F1-score for fraudulent transactions is low (0.06), highlighting the difficulty in identifying fraud without a significant number of false positives.

```
Confusion Matrix:
[[54020  2844]
 [   10    88]]
```

 The first row corresponds to normal transactions (Class 0):

- **54020** true negatives (correctly predicted as normal).
- **2844** false positives (incorrectly predicted as fraud).

 The second row corresponds to fraudulent transactions (Class 1):

- **10** false negatives (incorrectly predicted as normal).
- **88** true positives (correctly predicted as fraud).

```
1/1 [==============================] - 0s 22ms/step
Sample 0:
  Reconstruction Error: 65.5436218925837
  Prediction: Fraud (Threshold: 1.4906384262057126)
1/1 [==============================] - 0s 19ms/step
Sample 1:
  Reconstruction Error: 0.8376235550297751
  Prediction: Normal (Threshold: 1.4906384262057126)
1/1 [==============================] - 0s 30ms/step
Sample 2:
  Reconstruction Error: 0.31452216628550395
  Prediction: Normal (Threshold: 1.4906384262057126)
1/1 [==============================] - 0s 18ms/step
Sample 3:
  Reconstruction Error: 0.2853646274277654
  Prediction: Normal (Threshold: 1.4906384262057126)
```

11. <u>Implement the concept of image denoising using autoencoders on MNIST data set</u>

****Import Data from Local PC****

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.layers import Input, Conv2D, Conv2DTranspose
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('your_file_path.csv')

# Extract features and labels
X = data.iloc[:, :-1].values.reshape(-1, 28, 28, 1)
y = data.iloc[:, -1].values

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Add noise to the images
noise_factor = 0.5
X_train_noisy = X_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=X_train.shape)
X_test_noisy = X_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
X_train_noisy = np.clip(X_train_noisy, 0.0, 1.0)
X_test_noisy = np.clip(X_test_noisy, 0.0, 1.0)

# Define the autoencoder model
input_img = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(encoded)
x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
decoded = Conv2DTranspose(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = models.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()

# Train the autoencoder
autoencoder.fit(X_train_noisy, X_train, epochs=10, batch_size=128, shuffle=True,
validation_data=(X_test_noisy, X_test))
```

```
# Predict denoised images
denoised_images = autoencoder.predict(X_test_noisy)

# Visualize results
n = 10
plt.figure(figsize=(20, 6))
for i in range(n):
    # Display Original
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis("off")

    # Display Noisy
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(X_test_noisy[i].reshape(28, 28), cmap='gray')
    plt.title("Noisy")
    plt.axis("off")

    # Display Denoised
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
    plt.title("Denoised")
    plt.axis("off")

plt.show()
```

<u>***Import from Internet***</u>

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.layers import Input, Conv2D, Conv2DTranspose
import matplotlib.pyplot as plt

(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)

input_img = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
```

```
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)  # Removed strides here
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)  # Removed strides here
encoded = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

# Decoder
x = Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(encoded)
x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)
decoded = Conv2DTranspose(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = models.Model(input_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()

autoencoder.fit(x_train_noisy, x_train, epochs=30, batch_size=128, shuffle=True,
validation_data=(x_test_noisy, x_test))

denoised_images = autoencoder.predict(x_test_noisy)

n = 10
plt.figure(figsize=(20, 6))
for i in range(n):
    # Display Original
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis("off")

    # Display Noisy
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')
    plt.title("Noisy")
    plt.axis("off")

    # Display Denoised
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
    plt.title("Denoised")
    plt.axis("off")

plt.show()
```

12. <u>Implement object detection using Transfer learning on food dataset</u>

```
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.applications import VGG16
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np

train_dir = r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\animal10\train_dataset"
val_dir = r"C:\Users\AVNISH\Desktop\Sem 7\z_DL_Dataset\animal10\val_dataset"
img_size = (224, 224)
batch_size = 16

train_dataset = image_dataset_from_directory(
    train_dir,
    image_size=img_size,
    batch_size=batch_size
)

val_dataset = image_dataset_from_directory(
    val_dir,
    image_size=img_size,
    batch_size=batch_size
)

base_model = VGG16(input_shape=img_size + (3,), include_top=False, weights='imagenet')
base_model.trainable = False

model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(len(train_dataset.class_names), activation='softmax')
])

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

early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=10,
    callbacks=[early_stopping]
)

loss, accuracy = model.evaluate(val_dataset)
print(f"Validation accuracy: {accuracy * 100:.2f}%")
```

```
y_true = np.concatenate([y for x, y in val_dataset], axis=0)
y_pred = np.argmax(model.predict(val_dataset), axis=1)
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(10, 8))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=val_dataset.class_names)
disp.plot(cmap='Blues', values_format='d')
plt.title("Confusion Matrix")
plt.show()

plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Training Accuracy', 'Validation Accuracy'])
plt.title("Training and Validation Accuracy")
plt.show()
```

13. Implement the Continuous Bag of Words (CBOW) Model.

```
import numpy as np, import tensorflow as tf, import matplotlib.pyplot as plt
from collections import Counter
from sklearn.decomposition import PCA

# Larger sample text to train on
text = """
Knowledge is the pathway to discovery. Learning transforms our understanding of the world…
"""

# Tokenize and preprocess the text
words = text.lower().split()
vocab = set(words)
vocab_size = len(vocab)

# Create word to index and index to word dictionaries
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for word, i in word_to_ix.items()}

# Define context window size
context_size = 2  # Larger context window would provide richer context

# Generate context-target pairs
def generate_context_target_pairs(words, context_size):
    pairs = []
    for i in range(context_size, len(words) - context_size):
        context = words[i - context_size:i] + words[i + 1:i + context_size + 1]
        target = words[i]
        pairs.append((context, target))
    return pairs

# Create context-target pairs
data = generate_context_target_pairs(words, context_size)

# Prepare the training data
X = []
y = []

for context, target in data:
    X.append([word_to_ix[w] for w in context])
    y.append(word_to_ix[target])

X = np.array(X)
y = np.array(y)

# CBOW Model Definition
class CBOW(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(CBOW, self).__init__()
        self.embeddings = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.linear = tf.keras.layers.Dense(vocab_size)
```

```
    def call(self, context_words):
        embedded = self.embeddings(context_words)
        context_vector = tf.reduce_mean(embedded, axis=1)  # Average the embeddings of context
words
        out = self.linear(context_vector)
        return out

embedding_dim = 50  # Increase embedding dimension for more nuanced representations
model = CBOW(vocab_size, embedding_dim)
loss_function = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

epochs = 100
for epoch in range(epochs):
    with tf.GradientTape() as tape:
        predictions = model(X)
        loss = loss_function(y, predictions)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    if epoch % 20 == 0:
        print(f"Epoch: {epoch}, Loss: {loss.numpy():.4f}")

embeddings = model.embeddings.weights[0].numpy()

# Function to find similar words based on cosine similarity
def find_similar_words(word, embeddings, word_to_ix, ix_to_word, top_n=5):
    word_vec = embeddings[word_to_ix[word]]
    similarities = np.dot(embeddings, word_vec) / (
        np.linalg.norm(embeddings, axis=1) * np.linalg.norm(word_vec) + 1e-9
    )
    closest_words = similarities.argsort()[-(top_n + 1):-1]  # Exclude the word itself
    return [ix_to_word[idx] for idx in closest_words]

test_word = "creativity"
similar_words = find_similar_words(test_word, embeddings, word_to_ix, ix_to_word)
print(f"Words similar to '{test_word}': {similar_words}")

def visualize_embeddings(embeddings, word_to_ix):
    pca = PCA(n_components=2)
    reduced_embeddings = pca.fit_transform(embeddings)
    plt.figure(figsize=(10, 10))
    for word, idx in word_to_ix.items():
        x, y = reduced_embeddings[idx]
        plt.scatter(x, y)
        plt.text(x, y, word, fontsize=12)
    plt.show()

visualize_embeddings(embeddings, word_to_ix)
```

14. <u>Implement object detection using Transfer learning on food dataset</u>

```
pip install tensorflow
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np

train_dir = "path/to/train"
val_dir = "path/to/val"
test_dir = "path/to/test"

# Define parameters
img_size = (224, 224)  # MobileNetV2 expects 224x224 images
batch_size = 32

# Load datasets
train_dataset = image_dataset_from_directory(train_dir, image_size=img_size, batch_size=batch_size)
val_dataset = image_dataset_from_directory(val_dir, image_size=img_size, batch_size=batch_size)
test_dataset = image_dataset_from_directory(test_dir, image_size=img_size, batch_size=batch_size)

# Load the base model (MobileNetV2) with pre-trained weights
base_model = tf.keras.applications.MobileNetV2(input_shape=(224, 224, 3),
                                               include_top=False,
                                               weights='imagenet')
base_model.trainable = False  # Freeze the base model layers

# Build the model
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(1, activation='sigmoid')  # Binary classification output layer
])

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

model.summary()
epochs = 10
history = model.fit(train_dataset, validation_data=val_dataset, epochs=epochs)
test_loss, test_accuracy = model.evaluate(test_dataset)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")
import cv2
import matplotlib.pyplot as plt

def predict_and_display(model, img_path):
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=img_size)
    img_array = tf.keras.preprocessing.image.img_to_array(img)
```

```
    img_array = tf.expand_dims(img_array, 0)  # Create a batch

    prediction = model.predict(img_array)
    class_name = 'Dog' if prediction[0] > 0.5 else 'Cat'

    plt.imshow(img)
    plt.title(f"Prediction: {class_name}")
    plt.axis('off')
    plt.show()

# Test the function with an example image
predict_and_display(model, "path/to/some_test_image.jpg")
# Plot training & validation accuracy values
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')

plt.show()
```

FOOD DATASET

```
import tensorflow as tf
import numpy as np
from object_detection.utils import config_util
from object_detection.builders import model_builder
from object_detection.utils import visualization_utils as viz_utils
import cv2
import matplotlib.pyplot as plt
# Define paths
pipeline_config_path = 'path/to/your/pipeline.config'  # Customize with your pipeline config path
model_checkpoint = 'path/to/your/pre-trained-model/checkpoint'  # Model checkpoint folder

# Load the pipeline config and build the model
configs = config_util.get_configs_from_pipeline_file(pipeline_config_path)
model_config = configs['model']
detection_model = model_builder.build(model_config=model_config, is_training=True)
# Restore checkpoint for transfer learning
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
ckpt.restore(model_checkpoint).expect_partial()
```

```
# Load dataset (example with TFRecord files, adjust accordingly if using COCO format)
train_dataset = tf.data.TFRecordDataset("path/to/train.record")
val_dataset = tf.data.TFRecordDataset("path/to/val.record")

# Parse the dataset using a parser function
def parse_tfrecord_fn(example):
    # Define the parsing schema as per your dataset
    pass  # Complete this based on your TFRecord schema

train_dataset = train_dataset.map(parse_tfrecord_fn)
val_dataset = val_dataset.map(parse_tfrecord_fn)
# Define optimizer and metrics
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# Training loop
epochs = 10
for epoch in range(epochs):
    print(f"Epoch {epoch + 1}/{epochs}")

    for batch in train_dataset:
        with tf.GradientTape() as tape:
            predictions = detection_model(batch['image'], training=True)
            loss_value = detection_model.compute_loss(predictions, batch['groundtruth_boxes'],
batch['groundtruth_classes'])

        gradients = tape.gradient(loss_value, detection_model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, detection_model.trainable_variables))

    print(f"Loss: {loss_value.numpy()}")

    # Validate on validation dataset if desired
def load_image_into_numpy_array(path):
    return np.array(cv2.imread(path))

def detect_objects_in_image(image_path):
    image_np = load_image_into_numpy_array(image_path)
    input_tensor = tf.convert_to_tensor(image_np)
    input_tensor = input_tensor[tf.newaxis, ...]

    detections = detection_model(input_tensor, training=False)

    # Visualize results
    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np,
        detections['detection_boxes'][0].numpy(),
        detections['detection_classes'][0].numpy().astype(np.int32),
        detections['detection_scores'][0].numpy(),
        category_index,  # Dictionary mapping class IDs to names (e.g., {1: "apple", 2: "banana"})
        use_normalized_coordinates=True,
        line_thickness=8
    )
```

```
    plt.figure(figsize=(12, 8))
    plt.imshow(image_np)
    plt.axis('off')
    plt.show()

# Test the function
detect_objects_in_image("path/to/test/image.jpg")
# Example placeholder for evaluation
test_loss = 0
num_batches = 0

for batch in val_dataset:
    predictions = detection_model(batch['image'], training=False)
    test_loss += detection_model.compute_loss(predictions, batch['groundtruth_boxes'],
batch['groundtruth_classes'])
    num_batches += 1

test_loss /= num_batches
print(f"Test Loss: {test_loss.numpy()}")
```