| **S.No.** | **Index** | **Page No.** | **Date** | **Total Credits(10)** | **Signature** |
| --- | --- | --- | --- | --- | --- |
|  | **Prerequisites** |  |  |  |  |
|  | **Instruction to open the Kaggle online Platform** |  |  |  |  |
| **Part I : BASIC LINEAR ALGEBRA EXERCISES IN PYTHON** | | | | | |
| **1** | 1. Matrix Multiplication **(2 Credits)** 2. Eigen Decomposition **(2 Credits)** 3. Singular Value Decomposition (SVD) **(2 Credits)** 4. Determinant Calculation **(2 Credits)** 5. Norm Calculation **(2 Credits)** |  |  |  |  |
| **Part II : DEEP LEARNING APPLICATIONS IN PYTHON** | | | | | |
| **2** | **Image Classification**   1. Basic Image Classification using CIFAR-10 Dataset **( 6 Credits)** 2. Handwritten Digit Recognition using CNNs (MNIST Dataset) **( +2 Credits)** 3. Fine-Tuning Pre-trained Models for Cat vs. Dog Classification **( +2 Credits)** |  |  |  |  |
| **3** | **Object Detection**   1. Basic Object Detection using OpenCV and Haar Cascades **( 6 Credits)** 2. Object Detection with YOLO (You Only Look Once) **( +2 Credits)** 3. Mask R-CNN for Instance Segmentation **( +2 Credits)** |  |  |  |  |
| **4** | **Object Classification**   1. Basic Object Classification using Pre-trained VGG16 Model **( 6 Credits)** 2. ImageNet Classification with Deep Residual Networks (ResNet) **( +2 Credits)** 3. Classifying Species of Flowers using Transfer Learning **( +2 Credits)** |  |  |  |  |
| **5** | **Text Classification**   1. Basic Text Classification using Naive Bayes and Bag-of-Words **( 6 Credits)** 2. News Article Categorization using BERT **( +2 Credits)** 3. Spam Detection in Email using LSTM Networks **( +2 Credits)** |  |  |  |  |
| **6** | **Sentiment Analysis**   1. Basic Sentiment Analysis using Logistic Regression **( 6 Credits)** 2. Twitter Sentiment Analysis using LSTM and GloVe Embeddings **( +2 Credits)** 3. Movie Reviews Sentiment Classification with BERT **( +2 Credits)** |  |  |  |  |
| **7** | **Text Summarization**   1. Basic Text Summarization using TF-IDF and Cosine Similarity **( 6 Credits)** 2. Abstractive Text Summarization with Transformers **( +2 Credits)** 3. Extractive Summarization using BERT and SpaCy **( +2 Credits)** |  |  |  |  |
| **8** | **Text Entailment**   1. Basic Text Entailment using Simple Rule-Based Methods **( 6 Credits)** 2. Natural Language Inference with BERT **( +2 Credits)** 3. Sentence Pair Classification using Siamese Networks **( +2 Credits)** |  |  |  |  |
| **9** | **Word and Sentence Embedding**   1. Basic Word Embeddings with TF-IDF **( 6 Credits)** 2. Generating Word Embeddings using Word2Vec and GloVe **( +2 Credits)** 3. Sentence Embeddings with Universal Sentence Encoder **(10 Credits)** |  |  |  |  |
| **10** | **Question Answering**   1. Basic Q&A System using Keyword Matching **( 6 Credits)** 2. Building a Q&A System with BERT **( +2 Credits)** 3. Question Answering on SQuAD Dataset using Transformers **( +2 Credits)** |  |  |  |  |
| **11** | **Machine Translation**   1. Basic Machine Translation using Rule-Based Methods **( 6 Credits)** 2. English to French Translation using Seq2Seq with Attention **( +2 Credits)** 3. Neural Machine Translation with Transformers (English to German) **( +2 Credits)** |  |  |  |  |
| **12** | **Dialogue System**   1. Basic Rule-Based Chatbot using Python NLTK **( 6 Credits)** 2. Building a Chatbot using Seq2Seq Models **( +2 Credits)** 3. Conversational AI with Transformer-based Models **( +2 Credits)** |  |  |  |  |

**PREREQUISITES**

Before starting, ensure you have a basic understanding of Python programming, deep learning concepts, and some experience with linear algebra. You should also have a working knowledge of **NumPy, pandas, TensorFlow, and matplotlib** libraries in Python. You should also have a working knowledge of **Kaggle online platform**.

**INSTRUCTION TO OPEN THE KAGGLE ONLINE PLATFORM**

**Navigating from Google to Kaggle**

- If you're starting from Google, simply search for "Kaggle".

- Click on the first link, which should direct you to Kaggle’s homepage.

- Follow the steps above to sign up or log in, and explore the various sections of Kaggle.

**1. Sign Up / Log In:**

- Go to [Kaggle](https://www.kaggle.com/).

- Click on the “Sign Up” button to create an account using your Google account or email address.

- If you already have an account, simply log in.

**2. Explore the Dashboard:**

- Once logged in, you’ll be directed to the Kaggle dashboard.

- Familiarize yourself with the main sections:

* Competitions,
* Datasets,
* Notebooks,
* Code,
* Discussions,
* Courses.

**3. Join a Competition:**

- Navigate to the “Competitions” tab to find ongoing competitions.

- Click on any competition to read the description, rules, and download the dataset.

**4. Download Datasets:**

- Go to the “Datasets” tab.

- Use the search bar to find datasets that interest you.

- Click on a dataset to view details and click the “Download” button to download the dataset to your local machine.

**5. Start a New Notebook:**

- Go to the “Notebooks” tab.

- Click on “New Notebook” to create a new notebook. You can write code in Python.

- You can also choose to upload existing notebooks or explore others' notebooks for inspiration.

**6. Kaggle Kernels:**

- Kaggle Kernels is the platform’s in-browser coding environment.

- To create a new Kernel, go to the “Notebooks” tab and click “New Kernel”.

- Select the environment (Python 3 or R) and start writing your code. You can also import datasets directly into your kernel.

**7. Use Kaggle’s Features:**

- Kernels: Write and run code in Python directly on Kaggle’s servers. Kernels can include data visualization, analysis, and model training.

- Discussion Forums: Engage with the community to discuss problems, share solutions, or ask for help.

- Courses: Kaggle offers free courses on data science and machine learning. Access them via the “Courses” tab.

**8. Collaborate and Share:**

- You can share your notebooks and datasets with the community.

- Click the “Share” button on your notebook to make it public or share it with selected users.

**PART I : BASIC LINEAR ALGEBRA EXERCISES IN PYTHON**

**Week 1: Exercise 1 - Matrix Multiplication**

**Objective:** Understand matrix multiplication using NumPy.

**Expected Outcome:** Ability to perform matrix multiplication and understand its properties.

1. **Import NumPy Library:**

import numpy as np

**2. Define Matrices:**

Create two matrices `A` and `B` using `np.array`.

A = np.array([[1, 2], [3, 4]])

B = np.array([[5, 6], [7, 8]])

**3. Perform Matrix Multiplication:**

Use `np.dot` to multiply the matrices `A` and `B`.

C = np.dot(A, B)

**4. Print the Matrices and Result:**

Display the matrices and the result.

print("Matrix A:")

print(A)

print("Matrix B:")

print(B)

print("Matrix C = A \* B:")

print(C)

**Week 1: Exercise 2- Eigen Decomposition**

**Objective:** Decompose a matrix into its eigenvalues and eigenvectors.

**Expected Outcome:** Ability to compute and understand eigenvalues and eigenvectors.

1. **Import NumPy Library:**

import numpy as np

**2. Define a Square Matrix:**

Create a square matrix `A` using `np.array`.

A = np.array([[4, 2], [1, 3]])

**3. Perform Eigen Decomposition:**

Use `np.linalg.eig` to compute the eigenvalues and eigenvectors of matrix `A`.

eigenvalues, eigenvectors = np.linalg.eig(A)

**4. Print the Matrix, Eigenvalues, and Eigenvectors:**

Display the matrix `A`, its eigenvalues, and eigenvectors.

print("Matrix A:")

print(A)

print("Eigenvalues:")

print(eigenvalues)

print("Eigenvectors:")

print(eigenvectors)

**Week 1:**  **Exercise 3- Singular Value Decomposition (SVD)**

**Objective:** Perform SVD on a matrix and understand its components.

**Expected Outcome:** Ability to perform SVD and understand its application in dimensionality reduction.

**1. Import NumPy Library:**

import numpy as np

**2. Define a Matrix:**

Create a matrix `A` using `np.array`.

A = np.array([[3, 2, 2], [2, 3, -2]])

**3. Perform SVD:**

Use `np.linalg.svd` to decompose matrix `A` into `U`, `S`, and `V`.

U, S, V = np.linalg.svd(A)

**4. Print the Matrices and Singular Values:**

Display the matrix `A`, and the components `U`, `S`, and `V`.

print("Matrix A:")

print(A)

print("U matrix:")

print(U)

print("Singular values:")

print(S)

print("V matrix:")

print(V)

**Week 1: Exercise 4- Determinant Calculation**

**Objective:** Calculate the determinant of a matrix.

**Expected Outcome:** Ability to compute the determinant and understand its significance.

**1. Import NumPy Library:**

import numpy as np

**2. Define a Square Matrix:**

Create a square matrix `A` using `np.array`.

A = np.array([[6, 1], [2, 3]])

**3. Calculate the Determinant:**

Use `np.linalg.det` to compute the determinant of matrix `A`.

det = np.linalg.det(A)

**4. Print the Matrix and Its Determinant:**

Display the matrix `A` and its determinant.

print("Matrix A:")

print(A)

print("Determinant of A:")

print(det)

**Week 1: Exercise 5 - Norm Calculation**

**Objective:** Calculate L1 and L2 norms of a vector.

**Expected Outcome:** Ability to compute and understand L1 and L2 norms.

**1. Import NumPy Library:**

import numpy as np

**2. Define a Vector:**

Create a vector `v` using `np.array`.

v = np.array([3, 4])

**3. Calculate L1 Norm:**

Use `np.linalg.norm` with `ord=1` to compute the L1 norm of vector `v`.

L1\_norm = np.linalg.norm(v, ord=1)

**4. Calculate L2 Norm:**

Use `np.linalg.norm` to compute the L2 norm of vector `v`.

L2\_norm = np.linalg.norm(v)

**5. Print the Vector and Its Norms:**

Display the vector `v` and its L1 and L2 norms.

print("Vector v:")

print(v)

print("L1 norm of v:")

print(L1\_norm)

print("L2 norm of v:")

print(L2\_norm)

## 

## 

## 

## **Part II: DEEP LEARNING APPLICATION**

**WEEK 2: I. IMAGE CLASSIFICATION**

1. **BASIC IMAGE CLASSIFICATION USING CIFAR-10 DATASET**

**Objective:**

To classify images from the CIFAR-10 dataset into 10 different classes using a simple Convolutional Neural Network (CNN).

**Expected Outcomes:**

- Understand the basics of image classification.

- Learn how to build and train a simple CNN.

- Evaluate the model's performance.

**Required Dataset:**

CIFAR-10 dataset.

**How to Import the Dataset:**

The CIFAR-10 dataset can be imported directly from the `keras.datasets` module.

1. **Import Necessary Libraries:**

*import tensorflow as tf*

*from tensorflow.keras import datasets, layers, models*

*import matplotlib.pyplot as plt*

**Explanation:**

- `tensorflow`: The main library for deep learning tasks.

- `datasets, layers, models`: Submodules of `tensorflow.keras` used for loading datasets, building neural network layers, and creating models.

- `matplotlib.pyplot`: Used for plotting and visualizing data.

2. **Load and Preprocess the Data:**

*# Load CIFAR-10 dataset*

*(train\_images, train\_labels), (test\_images, test\_labels) = datasets.cifar10.load\_data()*

*# Normalize pixel values to be between 0 and 1*

*train\_images, test\_images = train\_images / 255.0, test\_images / 255.0*

**Explanation:**

- The CIFAR-10 dataset is loaded, which consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class.

- The pixel values are normalized to be between 0 and 1 to make the training process more efficient.

3. **Visualize the Data:**

*class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']*

*# Display the first 25 images from the training set*

*plt.figure(figsize=(10,10))*

*for i in range(25):*

*plt.subplot(5, 5, i+1)*

*plt.xticks([])*

*plt.yticks([])*

*plt.grid(False)*

*plt.imshow(train\_images[i])*

*plt.xlabel(class\_names[train\_labels[i][0]])*

*plt.show()*

**Explanation:**

- The class names are defined to label the images.

- A 5x5 grid of the first 25 training images is displayed with their corresponding class labels.

4. **Build the CNN Model:**

*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)*

*])*

**Explanation:**

- A Sequential model is built layer by layer.

- Three convolutional layers are added, each followed by a max-pooling layer.

- The Flatten layer converts the 2D matrix to a 1D vector.

- A Dense layer with 64 neurons is added for learning complex representations.

- The final Dense layer has 10 neurons corresponding to the 10 classes, without activation as it will be handled by the loss function.

5. **Compile the Model:**

*model.compile(optimizer='adam',*

*loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),*

*metrics=['accuracy'])*

**Explanation:**

- The model is compiled with the Adam optimizer for efficient gradient descent.

- The loss function used is Sparse Categorical Crossentropy, suitable for integer labels.

- Accuracy is specified as the metric to evaluate during training and testing.

6. **Train the Model:**

*history = model.fit(train\_images, train\_labels, epochs=10,*

*validation\_data=(test\_images, test\_labels))*

**Explanation:**

- The model is trained for 10 epochs using the training data.

- Validation data is used to monitor the model's performance on unseen data during training.

7. **Evaluate the Model:**

*test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2)*

*print(f'Test accuracy: {test\_acc}')*

**Explanation:**

- The model's performance is evaluated on the test dataset.

- The test accuracy is printed to see how well the model generalizes to new data.

8. **Plot Training History:**

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

*plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')*

*plt.xlabel('Epoch')*

*plt.ylabel('Accuracy')*

*plt.ylim([0, 1])*

*plt.legend(loc='lower right')*

*plt.show()*

**Explanation:**

- The training and validation accuracy are plotted over epochs.

- This helps visualize the model's learning progress and detect any overfitting or underfitting issues.

**Sample Input and Expected Output:**

- **Sample Input:** An image from the CIFAR-10 dataset.

- **Expected Output:** The predicted class label of the image (e.g., "cat", "dog", etc.).

1. **HANDWRITTEN DIGIT RECOGNITION USING CNNS (MNIST DATASET)**

**Objective:**

To recognize handwritten digits (0-9) using a Convolutional Neural Network (CNN) on the MNIST dataset.

**Expected Outcomes:**

- Understand the architecture of CNNs.

- Learn how to preprocess image data for model training.

- Evaluate model performance on a test set.

**Required Dataset:**

MNIST dataset.

**How to Import the Dataset:**

The MNIST dataset can be imported directly from the `keras.datasets` module.

**Step-by-Step Instructions for Implementation:**

1. **Import Necessary Libraries:**

**Explanation:**

- The necessary libraries are imported, similar to the previous example.

2. **Load and Preprocess the Data:**

*# Load MNIST dataset*

*# Normalize pixel values to be between 0 and 1*

*# Reshape data to fit the model*

**Explanation:**

- The MNIST dataset is loaded, which consists of 70,000 grayscale images of handwritten digits (28x28 pixels).

- The pixel values are normalized.

- The images are reshaped to include a single channel, as required by the CNN.

3. **Build the CNN Model:**

*# Write the code here*

**Explanation:**

- The model architecture is similar to the CIFAR-10 example but adapted for grayscale images.

- The final Dense layer uses the softmax activation function for multi-class classification.

4. **Compile the Model:**

*# Write the code here*

**Explanation:**

- The model is compiled with the same optimizer and a suitable loss function for integer labels.

5. **Train the Model:**

*# Write the code here*

**Explanation:**

- The model is trained for 5 epochs, and validation data is used to monitor performance.

6. **Evaluate the Model:**

*# Write the code here*

**Explanation:**

- The model's performance is evaluated on the test dataset, and the test accuracy is printed.

7. **Plot Training History:**

*# Write the code here*

**Explanation:**

- The training and validation accuracy are plotted to visualize the model's learning progress.

**Sample Input and Expected Output:**

- **Sample Input:** An image of a handwritten digit from the MNIST dataset.

- **Expected Output:** The predicted digit (0-9).

1. **FINE-TUNING PRE-TRAINED MODELS FOR CAT VS. DOG CLASSIFICATION**

**Objective:**

To classify images of cats and dogs using a pre-trained model (VGG16) and fine-tuning it on a custom Cats vs. Dogs dataset.

**Expected Outcomes:**

- Understand transfer learning and fine-tuning of pre-trained models.

- Learn how to adapt a pre-trained model to a specific classification task.

- Evaluate model performance on a custom dataset.

**Required Dataset:**

Cats vs. Dogs dataset (can be downloaded from Kaggle or another source).

**How to Import the Dataset:**

Load the dataset using `tf.keras.preprocessing.image\_dataset\_from\_directory` and split it into training and validation sets.

**Step-by-Step Instructions for Implementation:**

1. **Import Necessary Libraries:**

***# Write the code here***

**Explanation:**

- Necessary libraries are imported for deep learning tasks and data visualization.

2. **Load and Preprocess the Data:**

*# Define paths to the train and validation datasets*

***# Write the code here***

*# Load the Cats vs. Dogs dataset*

***# Write the code here***

*# Define the class names*

***# Write the code here***

**Explanation:**

- The dataset is loaded using `tf.keras.preprocessing.image\_dataset\_from\_directory`, which automatically splits the dataset into training and validation sets.

- Images are resized to 150x150 pixels to fit the input requirements of the pre-trained model.

- `class\_names` are defined based on the subdirectories in the dataset.

3. **Visualize the Data:**

***# Write the code here***

**Explanation:**

- Displays a grid of 9 images from the training dataset with their corresponding class labels.

- Helps in understanding the content and distribution of the dataset.

4. **Build the Model Using a Pre-trained Base:**

***# Write the code here***

**Explanation:**

- Loads the VGG16 model as the base, excluding the top classification layers (`include\_top=False`).

- Freezes the weights of the pre-trained layers (`base\_model.trainable = False`) to prevent them from being updated during training.

5. **Compile the Model:**

***# Write the code here***

**Explanation:**

- Compiles the model with Adam optimizer and binary cross-entropy loss for binary classification (cat vs. dog).

- Accuracy is chosen as the metric to monitor during training.

6. **Train the Model:**

***# Write the code here***

**Explanation:**

- Trains the model for 5 epochs using the training dataset.

- Evaluates the model on the validation dataset during training to monitor performance.

7. **Fine-Tune the Model (Optional):**

***# Write the code here***

**Explanation:**

- Optionally unfreezes the base model layers (`base\_model.trainable = True`) for fine-tuning.

- Compiles the model again with a lower learning rate (`1e-5`) for fine-tuning.

- Trains the model for an additional 5 epochs to fine-tune the weights.

**8. Evaluate the Model:**

***# Write the code here***

**Explanation:**

- Evaluates the model's performance on the validation dataset after training.

- Prints the validation accuracy to assess how well the model performs on unseen data.

**9. Plot Training History:**

***# Write the code here***

**Explanation:**

- Plots the training and validation accuracies over epochs.

- Helps visualize the model's learning progress and performance during initial training and fine-tuning stages.

**Sample Input and Expected Output:**

- **Sample Input:** An image of a cat or dog from the custom dataset.

- **Expected Output:** The predicted class label ("cat" or "dog").

**WEEK 3: II. OBJECT DETECTION**

1. **BASIC OBJECT DETECTION USING OPENCV AND HAAR CASCADES**

**Objective:**

To detect faces in an image using the Haar Cascade classifier provided by OpenCV.

**Expected Outcomes:**

- Understand the basics of object detection.

- Learn how to use Haar Cascades for face detection.

- Develop a simple application for real-time face detection.

**Required Dataset:**

Pre-trained Haar Cascade XML files for face detection (provided by OpenCV).

**How to Import the Dataset:**

*import cv2*

**Step-by-Step Instructions for Implementation:**

**1. Import Libraries:**

*import cv2*

**Explanation:**  We import the OpenCV library to use its functions for image processing and object detection.

**2. Load the Haar Cascade Classifier:**

*face\_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')*

**Explanation:**  Load the pre-trained Haar Cascade classifier for face detection. OpenCV provides the `haarcascade\_frontalface\_default.xml` file which contains the pre-trained model.

**3. Read the Input Image:**

*img = cv2.imread('path\_to\_image.jpg')*

*gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)*

**Explanation:**  Read the input image using `cv2.imread` and convert it to grayscale using `cv2.cvtColor` as the Haar Cascade classifier works better on grayscale images.

**4. Detect Faces:**

*faces = face\_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))*

**Explanation:** Use the `detectMultiScale` method to detect faces in the image. `scaleFactor` controls the image size reduction at each image scale. `minNeighbors` defines how many neighbors each candidate rectangle should have to retain it. `minSize` defines the minimum possible object size.

**5. Draw Rectangles around Detected Faces:**

*for (x, y, w, h) in faces:*

*cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)*

**Explanation:** Loop through the detected faces and draw rectangles around them using `cv2.rectangle`. The parameters `(x, y, w, h)` define the position and size of the rectangle.

**6. Display the Output:**

cv2.imshow('Face Detection', img)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Explanation:** Display the output image with detected faces using `cv2.imshow`. The `cv2.waitKey(0)` function waits for a key event indefinitely, and `cv2.destroyAllWindows` closes all OpenCV windows.

**Sample Input and Output:**

**Input:** An image with one or more faces.

**Output:** The same image with rectangles drawn around detected faces.

1. **OBJECT DETECTION WITH YOLO (YOU ONLY LOOK ONCE)**

**Objective:**

To perform real-time object detection using the YOLO algorithm.

**Expected Outcomes:**

Understand the YOLO object detection algorithm.

Learn how to use YOLO for real-time object detection.

Develop a real-time object detection application.

**Required Dataset:**

Pre-trained YOLO model weights and configuration files.

**How to Import the Dataset:**

*import cv2*

*import numpy as np*

**Step-by-Step Instructions for Implementation:**

**1. Import Libraries:**

Import the OpenCV and NumPy libraries. OpenCV is used for image processing and object detection, while NumPy is used for handling arrays.

***#Write the Code Here***

**2. Load YOLO:**

Load the pre-trained YOLO model using the `cv2.dnn.readNet` method. Load the class names from the `coco.names` file which contains the names of the 80 object classes that YOLO can detect.

**3. Load the Input Image:**

Read the input image using `cv2.imread` and get the image dimensions (height and width).

***#Write the Code Here***

**4. Prepare the Image for YOLO:**

Convert the image to a blob using `cv2.dnn.blobFromImage`. The blob is a 4D array used as input to the YOLO model. The parameters include the image, scale factor, size, mean subtraction values, swapping of the Red and Blue channels, and cropping.

***#Write the Code Here***

**5. Get Output Layer Names:**

Get the names of the output layers using `net.getLayerNames` and `net.getUnconnectedOutLayers`. YOLO has multiple output layers, each responsible for detecting objects at different scales.

***#Write the Code Here***

**6. Forward Pass through YOLO:**

Perform a forward pass through the YOLO network using the `net.forward` method. The output is a list of detections from each output layer.

***#Write the Code Here***

**7. Process the Output:**

Process the output detections. For each detection, extract the scores, class ID, and confidence. If the confidence is above a threshold (0.5), calculate the bounding box coordinates and add them to the list of boxes, confidences, and class IDs.

***#Write the Code Here***

**8. Apply Non-Maximum Suppression:**

Apply Non-Maximum Suppression (NMS) to remove overlapping bounding boxes using `cv2.dnn.NMSBoxes`. Draw rectangles and labels around the detected objects.

***#Write the Code Here***

**9. Display the Output:**

Display the output image with detected objects using `cv2.imshow`. The `cv2.waitKey(0)` function waits for a key event indefinitely, and `cv2.destroyAllWindows` closes all OpenCV windows.

***#Write the Code Here***

**Sample Input and Output:**

Input: An image with multiple objects.

Output: The same image with rectangles and labels around detected objects.

1. **MASK R-CNN FOR INSTANCE SEGMENTATION**

**Objective:**

To perform instance segmentation using the Mask R-CNN algorithm.

**Expected Outcomes:**

Understand the Mask R-CNN algorithm.

Learn how to use Mask R-CNN for instance segmentation.

Develop an instance segmentation application.

**Required Dataset:**

Pre-trained Mask R-CNN model weights and configuration files.

**How to Import the Dataset:**

*import cv2*

*import numpy as np*

**Step-by-Step Instructions for Implementation:**

**1. Import Libraries:**

Import the OpenCV and NumPy libraries. OpenCV is used for image processing and instance segmentation, while NumPy is used for handling arrays.

***#Write the Code Here***

**2. Load Mask R-CNN:**

Load the pre-trained Mask R-CNN model using the `cv2.dnn.readNetFromTensorflow` method. The model files include the frozen inference graph (`frozen\_inference\_graph.pb`) and the configuration file (`mask\_rcnn\_inception\_v2\_coco\_2018\_01\_28.pbtxt`).

***#Write the Code Here***

**3. Load the Input Image:**

Read the input image using `cv2.imread` and get the image dimensions (height and width).

***#Write the Code Here***

**4. Prepare the Image for Mask R-CNN:**

Convert the image to a blob using `cv2.dnn.blobFromImage`. The blob is a 4D array used as input to the Mask R-CNN model. The parameters include the image, scale factor, size, mean subtraction values, swapping of the Red and Blue channels, and cropping.

***#Write the Code Here***

**5. Forward Pass through Mask R-CNN:**

Perform a forward pass through the Mask R-CNN network using the `net.forward` method. The output includes bounding boxes and masks for detected objects.

***#Write the Code Here***

**6. Process the Output:**

Process the output detections. For each detection, extract the bounding box coordinates and apply the mask to the region of interest (ROI) if the confidence score is above a threshold (0.5). Draw rectangles around detected objects and apply the mask to highlight the segmented area.

***#Write the Code Here***

**7. Display the Output:**

Display the output image with detected objects using `cv2.imshow`. The `cv2.waitKey(0)` function waits for a key event indefinitely, and `cv2.destroyAllWindows` closes all OpenCV windows.

***#Write the Code Here***

**Sample** Input: An image with multiple objects.

**Sample** Output: The same image with masks applied to detected objects and rectangles around them.

**WEEK 4 : III.** **OBJECT CLASSIFICATION**

1. **BASIC OBJECT CLASSIFICATION USING PRE-TRAINED VGG16 MODEL**

**Objective:**

To classify objects in images using a pre-trained VGG16 model.

**Expected Outcomes:**

- Learn to use a pre-trained model for object classification.

- Understand the concept of transfer learning.

- Gain practical experience in image preprocessing and model evaluation.

**Required Dataset:**

CIFAR-10 dataset (available in Keras).

**How to Import the Dataset:**

*from keras.datasets import cifar10*

*from keras.utils import to\_categorical*

*# Load the dataset*

*(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()*

*# One-hot encode the labels*

*y\_train = to\_categorical(y\_train)*

*y\_test = to\_categorical(y\_test)*

**Step-by-Step Instructions for Implementation:**

**1. Import Necessary Libraries:**

*import numpy as np*

*from keras.applications.vgg16 import VGG16, preprocess\_input*

*from keras.models import Model*

*from keras.layers import Dense, Flatten*

*from keras.preprocessing.image import ImageDataGenerator*

*from keras.optimizers import Adam*

*from keras.callbacks import EarlyStopping*

**Explanation:**

- `numpy` is used for numerical operations.

- `VGG16` is the pre-trained model we will use.

- `Model`, `Dense`, and `Flatten` are Keras layers for building our model.

- `ImageDataGenerator` is used for data augmentation.

- `Adam` is the optimizer used for training.

- `EarlyStopping` is a callback to stop training when validation loss stops improving.

**2. Load the Pre-trained VGG16 Model:**

*# Load the VGG16 model without the top fully connected layers*

*base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(32, 32, 3))*

**Explanation:**

- `weights='imagenet'` loads the weights pre-trained on the ImageNet dataset.

- `include\_top=False` excludes the top fully connected layers.

- `input\_shape=(32, 32, 3)` specifies the input shape of the images.

**3. Add Custom Layers for CIFAR-10 Classification:**

*# Add custom layers on top of the base model*

*x = base\_model.output*

*x = Flatten()(x)*

*x = Dense(256, activation='relu')(x)*

*predictions = Dense(10, activation='softmax')(x)*

*# Define the complete model*

*model = Model(inputs=base\_model.input, outputs=predictions)*

**Explanation:**

- `Flatten` converts the 3D output from the base model to 1D.

- `Dense(256, activation='relu')` adds a fully connected layer with 256 units and ReLU activation.

- `Dense(10, activation='softmax')` adds the output layer with 10 units (for 10 classes) and softmax activation.

**4. Freeze the Base Model Layers:**

*for layer in base\_model.layers:*

*layer.trainable = False*

**Explanation:**

- Freezes the weights of the pre-trained layers so they are not updated during training, focusing training on the new layers.

**5. Compile the Model:**

*model.compile(optimizer=Adam(), loss='categorical\_crossentropy', metrics=['accuracy'])*

**Explanation:**

- `Adam` optimizer is used for training.

- `categorical\_crossentropy` is the loss function for multi-class classification.

- `accuracy` is used as the evaluation metric.

**6. Preprocess the Data:**

*# Preprocess the data*

*x\_train = preprocess\_input(x\_train)*

*x\_test = preprocess\_input(x\_test)*

**Explanation:**

- `preprocess\_input` scales the pixel values to the range used during model training.

**7. Create Data Generators:**

*# Create data generators for data augmentation*

*datagen = ImageDataGenerator(*

*rotation\_range=15,*

*width\_shift\_range=0.1,*

*height\_shift\_range=0.1,*

*horizontal\_flip=True*

*)*

*datagen.fit(x\_train)*

**Explanation:**

- `ImageDataGenerator` is used for data augmentation to prevent overfitting.

- `rotation\_range=15` allows images to be rotated up to 15 degrees.

- `width\_shift\_range=0.1` and `height\_shift\_range=0.1` allow horizontal and vertical shifts.

- `horizontal\_flip=True` allows horizontal flipping of images.

**8. Train the Model:**

*# Train the model*

*early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)*

*history = model.fit(*

*datagen.flow(x\_train, y\_train, batch\_size=32),*

*validation\_data=(x\_test, y\_test),*

*epochs=50,*

*callbacks=[early\_stopping]*

*)*

**Explanation:**

- `EarlyStopping` stops training if validation loss does not improve for 5 epochs.

- `model.fit` trains the model with augmented data from `datagen`.

- `validation\_data` is used to evaluate the model during training.

**9. Evaluate the Model:**

*# Evaluate the model on test data*

*loss, accuracy = model.evaluate(x\_test, y\_test)*

*print(f'Test Accuracy: {accuracy \* 100:.2f}%')*

**Explanation:**

- `model.evaluate` evaluates the model on the test data.

- Prints the test accuracy.

**Sample Input and Output:**

Sample Input: An image from the CIFAR-10 dataset (e.g., a small 32x32 image of a frog).

Expected Output: The model predicts the class of the image (e.g., 'frog').

1. **IMAGENET CLASSIFICATION WITH DEEP RESIDUAL NETWORKS (RESNET)**

**Objective:**

To classify images from the ImageNet dataset using a deep residual network (ResNet).

**Expected Outcomes:**

- Understand the architecture of ResNet.

- Learn to use pre-trained models for large-scale image classification.

- Gain experience in handling large datasets.

**Required Dataset:**

ImageNet dataset (or a smaller subset for practice).

**How to Import the Dataset:**

f*rom keras.applications.resnet50 import ResNet50, preprocess\_input, decode\_predictions*

*from keras.preprocessing import image*

*import numpy as np*

**Step-by-Step Instructions for Implementation:**

**1. Import Necessary Libraries:**

***#Write the Code Here***

- `ResNet50` is the pre-trained ResNet model.

- `preprocess\_input` prepares the input images.

- `decode\_predictions` decodes the model's predictions.

- `image` helps in loading and processing images.

- `numpy` is used for numerical operations.

**2. Load the Pre-trained ResNet50 Model:**

*model = ResNet50(weights='imagenet')*

**Explanation:**

- Loads the ResNet50 model pre-trained on the ImageNet dataset.

**3. Preprocess the Input Image:**

*# Load an image file that contains an image to be classified*

***#Write the Code Here***

*# Convert the image to a numpy array*

*x = image.img\_to\_array(img)*

*# Add a batch dimension*

*# Preprocess the input image*

- `image.load\_img` loads an image and resizes it to 224x224 pixels.

- `image.img\_to\_array` converts the image to a numpy array.

- `np.expand\_dims` adds a batch dimension.

- `preprocess\_input` scales the pixel values to the range used during model training.

**4. Predict the Class of the Image:**

***#Write the Code Here***

- `model.predict` generates predictions for the input image.

- `decode\_predictions` decodes the predictions to human-readable class names.

- Prints the top 3 predicted classes with their probabilities.

**Sample Input and Output:**

- Sample Input: An image of an elephant.

- Expected Output: The top 3 predicted classes with their probabilities (e.g., 'African elephant', 'tusker', 'Indian elephant').

1. **CLASSIFYING SPECIES OF FLOWERS USING TRANSFER LEARNING**

**Objective:**

To classify different species of flowers using transfer learning with a pre-trained model.

**Expected Outcomes:**

- Understand the concept and application of transfer learning.

- Learn to fine-tune pre-trained models for specific tasks.

- Gain practical experience in image preprocessing and model evaluation.

**Required Dataset:**

Flowers dataset (available from various sources like Kaggle).

**How to Import the Dataset:**

*from keras.preprocessing.image import ImageDataGenerator*

*# Assuming the dataset is in a directory with subdirectories for each class*

*train\_dir = 'flowers/train'*

*valid\_dir = 'flowers/valid'*

*# Create data generators*

*train\_datagen = ImageDataGenerator(rescale=1./255)*

*valid\_datagen = ImageDataGenerator(rescale=1./255)*

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

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

ImageDataGenerator` is used to preprocess and augment the images.

- `rescale=1./255` scales pixel values to the [0, 1] range.

- `flow\_from\_directory` loads images from the specified directories.

**Step-by-Step Instructions for Implementation:**

**1. Import Necessary Libraries:**

***#Write the Code Here***

- `VGG16` is the pre-trained model we will use.

- `Model`, `Dense`, and `Flatten` are Keras layers for building our model.

- `Adam` is the optimizer used for training.

- `EarlyStopping` is a callback to stop training when validation loss stops improving.

**2. Load the Pre-trained VGG16 Model:**

*# Load the VGG16 model without the top fully connected layers*

- `weights='imagenet'` loads the weights pre-trained on the ImageNet dataset.

- `include\_top=False` excludes the top fully connected layers.

- `input\_shape=(224, 224, 3)` specifies the input shape of the images.

**3. Add Custom Layers for Flower Classification:**

*# Add custom layers on top of the base model*

*# Define the complete model*

- `Flatten` converts the 3D output from the base model to 1D.

- `Dense(256, activation='relu')` adds a fully connected layer with 256 units and ReLU activation.

- `Dense(len(train\_generator.class\_indices), activation='softmax')` adds the output layer with units equal to the number of classes and softmax activation.

**4. Freeze the Base Model Layers:**

*for layer in base\_model.layers:*

*layer.trainable = False*

**Explanation:**

- Freezes the weights of the pre-trained layers so they are not updated during training, focusing training on the new layers.

**5. Compile the Model:**

***#Write the Code Here***

- `Adam` optimizer is used for training.

- `categorical\_crossentropy` is the loss function for multi-class classification.

- `accuracy` is used as the evaluation metric.

**6. Train the Model:**

*# Train the model*

***#Write the Code Here***

- `EarlyStopping` stops training if validation loss does not improve for 5 epochs.

- `model.fit` trains the model with augmented data from `train\_generator`.

- `validation\_data` is used to evaluate the model during training.

**7. Evaluate the Model:**

*# Evaluate the model on validation data*

- `model.evaluate` evaluates the model on the validation data.

- Prints the validation accuracy.

**Sample Input and Output:**

- Sample Input: An image of a flower from the validation set.

- Expected Output: The model predicts the class of the flower (e.g., 'daisy', 'rose', 'sunflower').

**WEEK 5: IV. TEXT CLASSIFICATION**

1. **BASIC TEXT CLASSIFICATION USING NAIVE BAYES AND BAG-OF-WORDS**

**Objective:**

To classify text documents into predefined categories using the Naive Bayes classifier with Bag-of-Words representation.

**Expected Outcomes:**

- Learn the basics of text classification using the Naive Bayes algorithm.

- Understand how Bag-of-Words representation works for text data.

- Evaluate the model's accuracy on a test dataset.

**Required Dataset:**

A dataset with text documents and their corresponding categories (e.g., news articles categorized into topics like sports, politics, technology).

**How to Import the Dataset:**

Assuming you have a CSV file named `news\_dataset.csv` where each row contains a text article and its category label:

*import pandas as pd*

*# Load dataset*

*df = pd.read\_csv('news\_dataset.csv')*

*# Display first few rows to understand the structure*

*print(df.head())*

**Step-by-Step Implementation:**

**1. Preprocessing:**

*from sklearn.feature\_extraction.text import CountVectorizer*

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

*from sklearn.naive\_bayes import MultinomialNB*

*from sklearn.metrics import classification\_report*

*# Assuming df contains 'text' column for articles and 'category' column for labels*

*X = df['text']*

*y = df['category']*

*# Splitting the dataset into training and testing sets*

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)*

*# Creating Bag-of-Words representation*

*vectorizer = CountVectorizer()*

*X\_train\_counts = vectorizer.fit\_transform(X\_train)*

*X\_test\_counts = vectorizer.transform(X\_test)*

**Explanation:**

- Clean the text data (remove punctuation, stopwords, etc.).

- Tokenize the text into words.

**2. Model Training:**

*# Initialize Naive Bayes classifier*

*clf = MultinomialNB()*

*# Train the classifier*

*clf.fit(X\_train\_counts, y\_train)*

**Explanation:**

- Initialize a Naive Bayes classifier (MultinomialNB in this case).

- Train the classifier on the training data.

**3. Model Evaluation:**

# Predict on the test set

y\_pred = clf.predict(X\_test\_counts)

# Evaluate performance

print(classification\_report(y\_test, y\_pred))

**Explanation:**

- Evaluate the trained model on the test set using metrics like accuracy, precision, recall, and F1-score.

- CountVectorizer: Converts a collection of text documents to a matrix of token counts.

- MultinomialNB: Implements the Naive Bayes algorithm suitable for classification with discrete features.

- train\_test\_split: Splits data into random train and test subsets.

- classification\_report: Computes precision, recall, F1-score, and support for each class.

**Sample Input (sample news article):** "Researchers have discovered a new species of butterfly in the Amazon rainforest."

**Expected Output (category prediction):** "Science & Environment"

1. **NEWS ARTICLE CATEGORIZATION USING BERT**

**Objective:**

To categorize news articles into predefined topics using BERT (Bidirectional Encoder Representations from Transformers), a state-of-the-art pre-trained language model.

**Expected Outcomes:**

- Understand the application of pre-trained language models like BERT for text classification.

- Achieve higher accuracy compared to traditional methods like Naive Bayes.

- Explore fine-tuning techniques for domain-specific tasks.

**Required Dataset:**

A dataset similar to the one used for Naive Bayes, ideally with more diverse and larger text samples for better performance with BERT.

**How to Import the Dataset:**

Same as the Naive Bayes example, load the dataset into a pandas DataFrame.

**Step-by-Step Implementation:**

**1. Data Preprocessing:**

*# Assuming df contains 'text' column for articles and 'category' column for labels*

***#Write the Code Here***

*# Splitting the dataset into training and testing sets*

***#Write the Code Here***

*# Load BERT tokenizer*

***#Write the Code Here***

*# Tokenize and encode sequences*

***#Write the Code Here***

*# Sentence to encode*

*add\_special\_tokens=True,*

*# Add '[CLS]' and '[SEP]'*

*max\_length=max\_length,*

*# Pad & truncate all sentences.*

*pad\_to\_max\_length=True,*

*return\_attention\_mask=True,*

*# Construct attn. masks.*

*# Return pytorch tensors.*

***#Write the Code Here***

*# Tokenize and encode training and validation sets*

***#Write the Code Here***

*# Convert labels to tensors*

***#Write the Code Here***

- Tokenization and padding of text sequences to ensure uniform input size.

- Encode labels into numerical format suitable for model training.

**2. Fine-Tuning BERT Model:**

- Load a pre-trained BERT model from a library like Hugging Face's `transformers`.

- Fine-tune the model on the news article dataset using techniques like transfer learning.

*# Load BERT model for sequence classification*

*# Create DataLoader for efficient batch processing*

*# Fine-tuning BERT model*

*# Optimizer and learning rate scheduler (not shown in simplified outline)*

**3. Training and Evaluation:**

- Split the dataset into training and validation sets.

- Train the BERT model on the training set.

- Evaluate the model's performance on the validation set using metrics appropriate for multi-class classification (accuracy, F1-score, etc.).

- BertTokenizer: Converts text into tokens that BERT understands.

- BertForSequenceClassification: BERT model fine-tuned for text classification tasks.

- TensorDataset, DataLoader: Prepares data for efficient batch processing.

- get\_linear\_schedule\_with\_warmup: Adjusts learning rate during training.

*# Train the model (not shown in simplified outline)*

*# Evaluation on validation set*

*model.eval()*

*# Predictions on validation set*

***#Write the Code Here***

*# Compute evaluation metrics*

**Sample Input (sample news article):** "Researchers have discovered a new species of butterfly in the Amazon rainforest."

**Expected Output (category prediction):** "Science & Environment"

1. **SPAM DETECTION IN EMAIL USING LSTM NETWORKS**

**Objective:**

To detect spam emails using Long Short-Term Memory (LSTM) networks, which are effective for sequential data like text.

**Expected Outcomes:**

- Implement a deep learning model (LSTM) for binary text classification.

- Understand the use of recurrent neural networks (RNNs) for sequence modeling.

- Achieve high accuracy in distinguishing between spam and non-spam emails.

**Required Dataset:**

A labeled dataset containing emails classified as spam or non-spam.

**How to Import the Dataset:**

Load the dataset into a pandas DataFrame or any suitable data structure.

**Step-by-Step Implementation:**

**1. Data Preprocessing:**

- Clean and preprocess the

- Convert text data into sequences of numerical tokens suitable for LSTM input.

*# Assuming df contains 'text' column for emails and 'label' column for spam or non-spam # 0 for non-spam, 1 for spam*

*# Tokenization and padding*

*# Maximum number of words to tokenize*

*# Splitting the dataset into training and testing sets*

**2. Model Architecture:**

- Design an LSTM model for text classification.

- Incorporate embedding layers to learn representations of words.

- Use dropout regularization to prevent overfitting.

*# LSTM Model*

*# Compile the model*

**3. Training and Evaluation:**

- Train the LSTM model on the training set.

- Evaluate the model's performance on the test set using metrics like accuracy, precision, recall, and F1-score.

- Tokenizer: Converts text into tokens and sequences.

- Embedding Layer: Learns representations of words in a dense vector space.

- LSTM Layer: Processes sequences and captures long-term dependencies.

- Model Compilation and Training: Defines loss function, optimizer, and trains the model.

*# Train the model*

*# Evaluation on test set*

**Sample (sample email):** "Congratulations! You've won a free vacation. Click here to claim your prize."

**Expected Output (classification):**  "Spam"

**WEEK 6: V. SENTIMENT ANALYSIS**

1. **BASIC SENTIMENT ANALYSIS USING LOGISTIC REGRESSION**

**Objective:**

To classify text sentiment (positive or negative) using a basic logistic regression model.

**Expected Outcomes:**

- Build a basic understanding of sentiment analysis techniques.

- Implement a simple logistic regression model for sentiment classification.

- Evaluate the model's accuracy in classifying sentiments.

**Required Dataset:**

Any labeled sentiment dataset. For example, you can use the IMDb movie reviews dataset which consists of reviews labeled as positive or negative.

**How to Import the Dataset:**

You can download the IMDb dataset from [here](https://ai.stanford.edu/~amaas/data/sentiment/). It contains 25,000 labeled movie reviews for training and another 25,000 for testing.

**Step-by-Step Instructions:**

**1. Data Preprocessing:**

- Load the dataset.

- Clean the text (remove stopwords, punctuation, lowercase).

- Tokenize the text (convert words into tokens).

- Convert text tokens into numerical vectors (TF-IDF vectors or word embeddings).

**2. Model Building:**

- Implement a logistic regression classifier using libraries like scikit-learn.

- Train the model on the preprocessed data.

**3. Model Evaluation:**

- Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

- Test the model on unseen data to measure its generalization ability.

**Code:**

*# Import necessary libraries*

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

*from sklearn.feature\_extraction.text import TfidfVectorizer*

*from sklearn.linear\_model import LogisticRegression*

*from sklearn.metrics import classification\_report, accuracy\_score*

*import pandas as pd*

*# Step 1: Load and preprocess the dataset*

*# Assuming 'reviews.csv' contains 'text' and 'sentiment' columns*

*df = pd.read\_csv('path\_to\_dataset/reviews.csv')*

*# Clean text*

*def clean\_text(text):*

*# Implement your text cleaning process*

*cleaned\_text = text.lower() # Convert to lowercase for uniformity*

*return cleaned\_text*

*df['cleaned\_text'] = df['text'].apply(clean\_text)*

*# Step 2: Convert text data into numerical vectors*

*vectorizer = TfidfVectorizer(max\_features=5000) # Use TF-IDF for vectorization*

*X = vectorizer.fit\_transform(df['cleaned\_text'])*

*y = df['sentiment'] # Assuming 'sentiment' column has labels (0 for negative, 1 for positive)*

*# Step 3: Split data into training and testing sets*

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)*

*# Step 4: Build and train the logistic regression model*

*model = LogisticRegression(max\_iter=1000)*

*model.fit(X\_train, y\_train)*

*# Step 5: Evaluate the model*

*y\_pred = model.predict(X\_test)*

*accuracy = accuracy\_score(y\_test, y\_pred)*

*print(f'Accuracy: {accuracy:.2f}')*

*print(classification\_report(y\_test, y\_pred))*

**Sample Input (Review Text):**  "This movie was amazing! I loved every minute of it."

**Expected Output:** Positive sentiment (1)

1. **TWITTER SENTIMENT ANALYSIS USING LSTM AND GLOVE EMBEDDINGS**

**Objective:**

To classify sentiment (positive, negative, or neutral) from Twitter data using LSTM (Long Short-Term Memory) with pre-trained GloVe word embeddings.

**Expected Outcomes:**

- Understand the use of deep learning models (LSTM) for sentiment analysis.

- Utilize pre-trained word embeddings (GloVe) for improved text representation.

- Classify sentiment from Twitter data accurately.

**Required Dataset:**

A labeled dataset of tweets with sentiment annotations. You can use the Sentiment140 dataset, which contains 1.6 million tweets labeled as 0 (negative), 2 (neutral), and 4 (positive).

**How to Import the Dataset:**

You can download the Sentiment140 dataset from [here](http://cs.stanford.edu/people/alecmgo/trainingandtestdata.zip).

**Step-by-Step Instructions:**

**1. Data Preprocessing:**

Load and clean the dataset (remove noise, normalize text).

Tokenization and Padding: Text data is tokenized and sequences are padded to a fixed length for input to the LSTM model.

GloVe Embeddings: Pre-trained GloVe embeddings are loaded and used to create an embedding matrix for the embedding layer of the LSTM model.

**2. Model Building:**

- Build an LSTM model using Keras.

- Use GloVe embeddings as an embedding layer in the LSTM model.

- Compile the model with appropriate loss and optimizer.

**3. Model Training:**

- Train the LSTM model on the preprocessed and tokenized text data.

**4. Model Evaluation:**

- Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

- Test the model on unseen Twitter data to measure its effectiveness.

**Code:**

*# Import necessary libraries*

*# Step 1: Load and preprocess the dataset*

*# Assuming 'tweets.csv' contains 'text' and 'sentiment' columns*

*# Clean text (you can define your cleaning function here)*

*# Implement your text cleaning process*

*# Step 2: Tokenize text and pad sequences*

*# Convert sentiment labels to numerical format*

*# Step 3: Split data into training and testing sets*

*# Step 4: Load pre-trained GloVe embeddings*

*# Create embedding matrix*

*# Step 5: Build LSTM model with GloVe embeddings*

*# Compile the model*

*# Step 6: Train the model*

*# Step 7: Evaluate the model*

**Sample Input (Tweet Text):** "I really enjoyed the movie, it was fantastic!"

**Expected Output: P**ositive sentiment

1. **MOVIE REVIEWS SENTIMENT CLASSIFICATION WITH BERT**

**Objective:**

To classify sentiment (positive or negative) from movie reviews using the BERT (Bidirectional Encoder Representations from Transformers) model.

**Expected Outcomes:**

- Understand the application of state-of-the-art language models (BERT) for sentiment analysis.

- Fine-tune a pre-trained BERT model for sentiment classification.

- Achieve high accuracy in classifying sentiment from movie reviews.

**Required Dataset:**

IMDb movie reviews dataset, which contains 50,000 labeled movie reviews for training and testing (25,000 each).

**How to Import the Dataset:**

You can download the IMDb dataset from [here](https://ai.stanford.edu/~amaas/data/sentiment/).

**Step-by-Step Instructions:**

**1. Data Preprocessing:**

- Load and preprocess the IMDb dataset (tokenization, padding).

- Convert labels to numerical format (0 for negative, 1 for positive).

**2. Model Fine-Tuning:**

- Load a pre-trained BERT model using Hugging Face's `transformers` library.

- Tokenize and encode the movie reviews using BERT tokenizer.

- Fine-tune the BERT model on the IMDb dataset for sentiment classification.

**3. Model Evaluation:**

- Evaluate the fine-tuned BERT model on the test set.

- Measure accuracy and other metrics to assess model performance.

**Code:**

*# Import necessary libraries*

*# Step 1: Load and preprocess the dataset*

*# Assuming 'imdb\_reviews.csv' contains 'review' and 'sentiment' columns*

*# Clean text (if necessary)*

*# Implement your text cleaning process if needed*

*# Step 2: Tokenize and encode reviews using BERT tokenizer*

*# Step 3: Split data into training and testing sets*

*# Step 4: Load pre-trained BERT model and fine-tune for sentiment classification*

*# Train the model*

*# Step 5: Evaluate the model*

**Sample Input (Movie Review):** "The movie was boring and uninteresting."

**Expected Output:** Negative sentiment (0)

**WEEK 7: VI. TEXT SUMMARIZATION**

1. **BASIC TEXT SUMMARIZATION USING TF-IDF AND COSINE SIMILARITY**

**Objective:**

To create a basic text summarization tool using TF-IDF (Term Frequency-Inverse Document Frequency) and Cosine Similarity.

**Expected Outcomes:**

- Understand how to use TF-IDF for feature extraction.

- Learn how to compute cosine similarity to find important sentences in the text.

- Develop a simple extractive summarization method.

**Required Dataset:**

Any text dataset. For demonstration, we can use a sample text from Wikipedia or a news article.

**How to Import the Dataset:**

For simplicity, we'll use a sample text defined in the code.

**Step-by-Step Instructions for Implementation:**

**1. Import Required Libraries:**

- `numpy` is used for numerical operations.

- `nltk` is used for natural language processing tasks.

- `stopwords` from `nltk.corpus` provides a list of common words to be ignored.

- `TfidfVectorizer` from `sklearn.feature\_extraction.text` converts text to TF-IDF features.

- `cosine\_similarity` from `sklearn.metrics.pairwise` computes the cosine similarity between TF-IDF vectors.

- `nltk.download('punkt')` and `nltk.download('stopwords')` download necessary datasets for tokenization and stopwords.

*import numpy as np*

*import nltk*

*from nltk.corpus import stopwords*

*from sklearn.feature\_extraction.text import TfidfVectorizer*

*from sklearn.metrics.pairwise import cosine\_similarity*

*nltk.download('punkt')*

*nltk.download('stopwords')*

**2. Define Sample Text:**

**-**text` is a sample paragraph used for summarization.

*text = """*

*Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.*

*Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.*

*"""*

**3. Preprocess the Text:**

- `sentences` splits the text into sentences using `nltk.sent\_tokenize`.

- `stop\_words` stores a set of English stopwords.

- `preprocess\_sentence` removes stopwords from a sentence.

- `preprocessed\_sentences` stores the preprocessed sentences.

*sentences = nltk.sent\_tokenize(text)*

*stop\_words = set(stopwords.words('english'))*

*def preprocess\_sentence(sentence):*

*return ' '.join([word for word in sentence.split() if word.lower() not in stop\_words])*

*preprocessed\_sentences = [preprocess\_sentence(sentence) for sentence in sentences]*

**4. Compute TF-IDF Matrix:**

- `TfidfVectorizer` is instantiated.

- `tfidf\_matrix` stores the TF-IDF representation of the preprocessed sentences.

*vectorizer = TfidfVectorizer()*

*tfidf\_matrix = vectorizer.fit\_transform(preprocessed\_sentences)*

**5. Compute Cosine Similarity:**

- `cosine\_sim\_matrix` stores the cosine similarity between TF-IDF vectors of the sentences.

*cosine\_sim\_matrix = cosine\_similarity(tfidf\_matrix, tfidf\_matrix)*

**6. Generate Summary:**

- `generate\_summary` ranks sentences based on their similarity scores and returns the top sentences as the summary.

- `scores` stores the sum of similarity scores for each sentence.

- `ranked\_sentences` stores the top `top\_n` sentences.

- `summary` stores the final summary.

*def generate\_summary(sentences, sim\_matrix, top\_n=2):*

*scores = sim\_matrix.sum(axis=1)*

*ranked\_sentences = [sentences[i] for i in scores.argsort()[-top\_n:]]*

*return ' '.join(ranked\_sentences)*

*summary = generate\_summary(sentences, cosine\_sim\_matrix)*

*print("Summary:")*

*print(summary)*

**Sample Input:**

text = """

Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.

Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.

"""

**Expected Output:**

***plaintext***

***Summary:***

Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.

1. **ABSTRACTIVE TEXT SUMMARIZATION WITH TRANSFORMERS**

**Objective:**

To create an abstractive text summarization tool using Transformer models, specifically BART (Bidirectional and Auto-Regressive Transformers).

**Expected Outcomes:**

- Understand how to use pre-trained Transformer models for text summarization.

- Learn how to fine-tune a Transformer model for the summarization task.

- Develop an abstractive summarization method that generates novel sentences.

**Required Dataset:**

A text dataset. For this demonstration, we'll use the `CNN/DailyMail` dataset available in the Hugging Face library.

**How to Import the Dataset:**

We'll use the `datasets` library from Hugging Face to load the `CNN/DailyMail` dataset.

**Step-by-Step Instructions for Implementation:**

**1. Install Required Libraries:**

- `transformers` provides state-of-the-art Transformer models.

- `datasets` allows easy access to various datasets, including the `CNN/DailyMail` dataset.

***# Write the code here***

**2. Import Required Libraries:**

- `BartForConditionalGeneration` and `BartTokenizer` are used to load and use the BART model.

- `load\_dataset` is used to load the `CNN/DailyMail` dataset.

***# Write the code here***

3. Load the Dataset:

- `load\_dataset` loads a small portion (1%) of the `CNN/DailyMail` test split.

***# Write the code here***

**4. Load Pre-trained BART Model and Tokenizer:**

- `BartForConditionalGeneration.from\_pretrained` loads the pre-trained BART model.

- `BartTokenizer.from\_pretrained` loads the corresponding tokenizer.

***# Write the code here***

**5. Summarize Text:**

- `summarize` function tokenizes the input text, generates the summary, and decodes it.

- `inputs` stores the tokenized input text.

- `summary\_ids` stores the generated summary token IDs.

- `tokenizer.decode` decodes the token IDs into human-readable text.

***# Write the code here***

**Sample Input:**

Text from the `CNN/DailyMail` dataset.

**Expected Output:**

A concise summary generated by the BART model, which is abstractive in nature.

1. **EXTRACTIVE SUMMARIZATION USING BERT AND SPACY**

**Objective:**

To create an extractive text summarization tool using BERT embeddings and SpaCy.

**Expected Outcomes:**

- Understand how to use BERT embeddings for text summarization.

- Learn how to integrate BERT with SpaCy for extractive summarization.

- Develop a method to extract important sentences from the text.

**Required Dataset:**

Any text dataset. For demonstration, we'll use a sample text from Wikipedia or a news article.

**How to Import the Dataset:**

For simplicity, we'll use a sample text defined in the code.

**Step-by-Step Instructions for Implementation:**

**1. Install Required Libraries:**

- `spacy` is used for NLP tasks.

- `torch` provides the PyTorch framework for deep learning.

- `transformers` provides state-of-the-art Transformer models.

- `en\_core\_web\_md` is a medium-sized English model for SpaCy.

***# Write the code here***

**2. Import Required Libraries:**

- `spacy` is used for NLP tasks.

- `torch` provides the PyTorch framework.

- `BertTokenizer` and `BertModel` are used to load and use the BERT model.

***# Write the code here***

**3. Load SpaCy Model and BERT Model:**

- `spacy.load` loads the SpaCy model.

- `BertTokenizer.from\_pretrained` loads the pre-trained BERT tokenizer.

- `BertModel.from\_pretrained` loads the pre-trained BERT model.

***# Write the code here***

**4. Define Sample Text:**

- `text` is a sample paragraph used for summarization.

**5. Preprocess and Tokenize Sentences:**

- `doc` processes the text using SpaCy.

- `sentences` stores the sentences extracted from the text.

- `tokenized\_sentences` stores the tokenized sentences.

***# Write the code here***

**6. Compute BERT Embeddings:**

- `with torch.no\_grad()` disables gradient computation for efficiency.

- `embeddings` stores the BERT embeddings for each tokenized sentence.

***# Write the code here***

**7. Compute Sentence Scores and Generate Summary:**

- `sentence\_scores` stores the mean embeddings for each sentence.

- `top\_sentence\_indices` stores the indices of the top `top\_n` sentences.

- `summary` stores the final summary.

***# Write the code here***

**Sample Input:**

text = """ """

**Expected Output:**

***plaintext***

**Summary:**

**WEEK 8: VII. TEXT ENTAILMENT APPLICATIONS IN PYTHON**

1. **BASIC TEXT ENTAILMENT USING SIMPLE RULE-BASED METHODS**

**Objective:**

Develop a simple text entailment model using rule-based methods to determine if one sentence entails another.

**Expected Outcomes:**

A basic understanding of text entailment.

Ability to implement simple rule-based approaches for text entailment.

**Required Dataset:**

A dataset containing sentence pairs labeled as "entailment" or "non-entailment."

**How to Import the Dataset:**

*import pandas as pd*

*# Load the dataset containing sentence pairs and labels*

*data = pd.read\_csv('text\_entailment\_dataset.csv')*

**Step-by-Step Instructions for Implementation:**

**1. Preprocess the Data:**

Tokenize the sentences and convert them to lowercase for uniformity.

*import nltk*

*nltk.download('punkt')*

*# Function to preprocess text by tokenizing and converting to lowercase*

*def preprocess(text):*

*return nltk.word\_tokenize(text.lower())*

*# Apply preprocessing to both sentences in each pair*

*data['sentence1\_tokens'] = data['sentence1'].apply(preprocess)*

*data['sentence2\_tokens'] = data['sentence2'].apply(preprocess)*

Here, the `preprocess` function tokenizes the sentences using NLTK's `word\_tokenize` and converts all text to lowercase to ensure case-insensitivity during comparison.

**2. Define Simple Rule-Based Methods:**

Check if all words in sentence2 are present in sentence1.

*# Function to check if all tokens in s2 are in s1*

*def simple\_rule\_based\_entailment(s1, s2):*

*return set(s2).issubset(set(s1))*

*# Apply the rule-based method to each sentence pair*

*data['prediction'] = data.apply(lambda row: simple\_rule\_based\_entailment(row['sentence1\_tokens'], row['sentence2\_tokens']), axis=1)*

This function checks if the tokens in the second sentence (`s2`) are a subset of the tokens in the first sentence (`s1`). If they are, it indicates a possible entailment.

**3. Evaluate the Model:**

Compare the predictions with the actual labels to determine accuracy.

*from sklearn.metrics import accuracy\_score*

*# Calculate the accuracy of the predictions*

*accuracy = accuracy\_score(data['label'], data['prediction'])*

*print(f'Accuracy: {accuracy}')*

The `accuracy\_score` function from `sklearn.metrics` is used to measure the accuracy of the rule-based model by comparing the predicted entailment labels with the actual labels.

**Sample Input:**

Sentence1: "The cat is on the mat."

Sentence2: "The mat has a cat."

**Expected Output:**

Entailment: False (since the presence of a cat on a mat doesn't necessarily entail that the mat has a cat)

**b.NATURAL LANGUAGE INFERENCE WITH BERT**

**Objective:**

Utilize BERT to build a text entailment model that classifies sentence pairs into entailment, contradiction, or neutral.

**Expected Outcomes:**

Understanding of using BERT for Natural Language Inference (NLI).

Implementation of a BERT-based model for NLI.

**Required Dataset:**

A pre-processed dataset like the SNLI or MultiNLI dataset.

**How to Import the Dataset:**

from datasets import load\_dataset

*# Load the SNLI dataset*

**Step-by-Step Instructions for Implementation:**

**1. Preprocess the Data:**

Tokenize the sentences using a BERT tokenizer.

*from transformers import BertTokenizer*

*# Initialize the BERT tokenizer*

*# Function to tokenize and prepare the data for BERT*

*# Apply preprocessing to the dataset*

Here, we use the `BertTokenizer` to tokenize the sentences and prepare them for the BERT model. The `preprocess` function tokenizes the "premise" and "hypothesis" pairs and returns them in a format compatible with BERT, including padding and truncation to ensure uniform input size.

**2. Load the Pre-Trained BERT Model:**

Use a pre-trained BERT model with a classification head for NLI tasks.

*from transformers import BertForSequenceClassification*

*# Load the BERT model for sequence classification with 3 output labels*

The `BertForSequenceClassification` model is loaded with three output labels corresponding to the possible classes: entailment, contradiction, and neutral.

**3. Train the Model:**

Fine-tune the BERT model on the NLI dataset.

from transformers import Trainer, TrainingArguments

*# Set up training arguments*

*# Initialize the Trainer with the model, training arguments, and datasets*

*# Train the model*

The `Trainer` class from Hugging Face's Transformers library is used to fine-tune the model. It handles the training loop, evaluation, and saving of the model.

**4. Evaluate the Model:**

Evaluate the model's performance on the validation set.

*# Evaluate the model and print the results*

The model's performance is evaluated on the validation dataset, providing metrics such as accuracy and loss.

**Sample Input:**

Premise: "A man inspects the uniform of a figure in some East Asian country."

Hypothesis: "The man is sleeping."

**Expected Output:**

Label: Contradiction (since inspecting a uniform does not imply sleeping)

**C. SENTENCE PAIR CLASSIFICATION USING SIAMESE NETWORKS**

**Objective:**

Develop a Siamese Network to classify sentence pairs based on their semantic similarity.

**Expected Outcomes:**

Understanding of Siamese Networks for sentence pair classification.

Implementation of a Siamese Network model for text entailment.

**Required Dataset:**

A dataset with sentence pairs and similarity labels.

**How to Import the Dataset:**

*data = pd.read\_csv('sentence\_similarity\_dataset.csv')*

**Step-by-Step Instructions for Implementation:**

**1. Preprocess the Data:**

Tokenize the sentences and convert them into embeddings using a pre-trained model.

*from sentence\_transformers import SentenceTransformer*

*# Load a Sentence Transformer model for generating sentence embeddings*

*# Function to get sentence embeddings*

*# Apply the embedding function to each sentence in the pairs*

The `SentenceTransformer` is used to convert sentences into vector embeddings. These embeddings are a numerical representation of the sentences' semantic content.

**2. Define the Siamese Network Architecture:**

Use a simple MLP to measure the similarity between sentence embeddings.

*from tensorflow.keras import layers, models*

*# Define the Siamese Network architecture*

*# Input layers for the two sentences*

*# Dense layers for feature extraction*

*# Concatenate the extracted features and add a final dense layer for output*

*# Compile the model*

*return model*

*# Create the Siamese network model*

The Siamese network uses two identical sub-networks to process the sentence embeddings. The outputs are then combined and passed through further layers to produce a similarity score between 0 and 1.

**3. Train the Model:**

Train the Siamese Network on the training data.

The model is trained on the sentence embeddings with corresponding similarity labels. The `fit` method handles the training process, including splitting the data for validation.

**4. Evaluate the Model:**

Evaluate the model's performance on a test set.

The evaluation measures the model's accuracy on the test set, using embeddings from the sentence pairs and their corresponding labels.

**Sample Input:**

Sentence1: "A woman is playing a piano."

Sentence2: "A person is playing a musical instrument."

**Expected Output:**

Similarity Score: 0.8 (Indicating a high similarity, suggesting the sentences describe similar situations)

**WEEK 9: VIII. WORD AND SENTENCE EMBEDDING**

1. **BASIC WORD EMBEDDINGS WITH TF-IDF**

**Objective**

To create basic word embeddings using the Term Frequency-Inverse Document Frequency (TF-IDF) method, capturing the importance of words in a document relative to a corpus.

**Expected Outcomes**

Understanding of how TF-IDF weights represent word importance.

A matrix of TF-IDF vectors representing the words in the corpus.

**Required Dataset**

A text dataset, such as a collection of documents or sentences. For example, the 20 Newsgroups dataset can be used.

**How to Import the Dataset**

You can use the `sklearn.datasets` library to fetch the 20 Newsgroups dataset.

**Step-by-Step Instructions for Implementation**

**1. Load the Dataset**

*from sklearn.datasets import fetch\_20newsgroups*

*newsgroups = fetch\_20newsgroups(subset='train')*

*texts = newsgroups.data*

We use `fetch\_20newsgroups` to load a subset of the 20 Newsgroups dataset.

**2. Preprocess the Text Data**

*from sklearn.feature\_extraction.text import TfidfVectorizer*

*vectorizer = TfidfVectorizer(stop\_words='english', max\_features=1000)*

*X\_tfidf = vectorizer.fit\_transform(texts)*

We initialize the `TfidfVectorizer` with stop word removal and a maximum of 1000 features to focus on the most important words. Then, we fit and transform the text data into TF-IDF vectors.

**Sample Input:** A list of documents, e.g., `["This is a sample document.", "This document is another example."]`

**Expected Output:** A sparse matrix of TF-IDF values, where each row represents a document, and each column represents a word from the vocabulary.

1. **GENERATING WORD EMBEDDINGS USING WORD2VEC AND GLOVE**

**Objective**

To generate dense vector representations of words using Word2Vec and GloVe models, capturing semantic similarities between words.

**Expected Outcomes**

* Understanding of how Word2Vec and GloVe create embeddings.
* Ability to create and visualize word embeddings.

**Required Dataset**

A large corpus of text, such as Wikipedia articles or text from books.

**How to Import the Dataset**

You can use any large text corpus. Let's use a sample corpus loaded from a text file for demonstration.

**Step-by-Step Instructions for Implementation**

**1. Preprocess the Text Data**

We preprocess the text data using `simple\_preprocess` to tokenize the text.

**2. Train Word2Vec Model**

We train the Word2Vec model using the preprocessed corpus. The `vector\_size` parameter defines the dimensionality of the word vectors, `window` specifies the context window size, and `sg=1` indicates the use of the Skip-gram model.

**3. Train GloVe Model**

For GloVe, you can use the `glove-python-binary` library or download pre-trained vectors.

**Sample Input:** A list of tokenized sentences, e.g., `[['this', 'is', 'a', 'sample', 'document'], ['another', 'example', 'document']]`

**Expected Output:** Word vectors for each word in the vocabulary, with similar words having similar vectors.

1. **SENTENCE EMBEDDINGS WITH UNIVERSAL SENTENCE ENCODER**

**Objective**

To generate embeddings for entire sentences, capturing the semantic meaning of the sentence.

**Expected Outcomes**

* Understanding of sentence embeddings.
* Ability to generate and use sentence embeddings for various NLP tasks.

**Required Dataset**

A collection of sentences, such as news articles or social media posts.

**How to Import the Dataset**

You can use the same method as above or any other source of sentences.

**Step-by-Step Instructions for Implementation**

**1. Install and Import the Universal Sentence Encoder**

*import tensorflow as tf*

*import tensorflow\_hub as hub*

*# Load the Universal Sentence Encoder*

*embed = hub.load("*[*https://tfhub.dev/google/universal-sentence-encoder/4*](https://tfhub.dev/google/universal-sentence-encoder/4)*")*

We use TensorFlow Hub to load the Universal Sentence Encoder (USE).

**2. Generate Sentence Embeddings**

We pass the sentences to the USE model to get the embeddings.

**Sample Input:** A list of sentences, e.g., `["This is a sentence.", "Another example sentence."]`

**Expected Output:** Dense vectors representing each sentence, with semantically similar sentences having closer vectors.

**WEEK 10: IX. QUESTION ANSWERING**

1. **BASIC Q&A SYSTEM USING KEYWORD MATCHING**

**Objective**

To build a simple Question Answering (Q&A) system that provides answers based on keyword matching from a predefined set of questions and answers.

**Expected Outcomes**

* Ability to handle a basic Q&A interaction.
* Understanding of keyword matching techniques for information retrieval.

**Required Dataset**

A small dataset containing questions and corresponding answers. For simplicity, this can be hardcoded or stored in a simple file (e.g., JSON, CSV).

**How to Import the Dataset**

If using a file, the dataset can be imported using standard file reading methods in Python.

**Step-by-Step Instructions for Implementation**

**1. Dataset Preparation**

Create a JSON or CSV file with questions and their corresponding answers.

**2. Data Loading**

Read the dataset into a Python dictionary.

*import json*

*with open('qa\_dataset.json', 'r') as f:*

*qa\_data = json.load(f)*

**3. Keyword Matching Function**

Implement a function that matches keywords from the user's query to the questions in the dataset.

*def find\_answer(question, qa\_data):*

*for q, a in qa\_data.items():*

*if all(keyword in q.lower() for keyword in question.lower().split()):*

*return a*

*return "Sorry, I don't know the answer to that question."*

**4. Main Function**

Implement the main interaction loop.

*def main():*

*while True:*

*user\_question = input("Ask a question: ")*

*if user\_question.lower() in ['exit', 'quit']:*

*break*

*answer = find\_answer(user\_question, qa\_data)*

*print("Answer:", answer)*

*if \_\_name\_\_ == "\_\_main\_\_":*

*main()*

**Sample Input:** "What is Python?"

**Expected Output:** "Python is a high-level programming language."

1. **BUILDING A Q&A SYSTEM WITH BERT**

**Objective**

To develop a Q&A system using BERT (Bidirectional Encoder Representations from Transformers) for contextual question answering.

**Expected Outcomes**

Understanding of using pre-trained language models like BERT for Q&A tasks.

Ability to fine-tune BERT on a specific Q&A dataset.

**Required Dataset**

A dataset such as the SQuAD (Stanford Question Answering Dataset) provides context paragraphs along with questions and answers.

**How to Import the Dataset**

Use the `datasets` library or manual download and loading.

**Step-by-Step Instructions for Implementation**

**1. Environment Setup**

Install necessary libraries: `transformers`, `torch`, `datasets`.

**2. Data Loading**

Load the SQuAD dataset.

**3. Model and Tokenizer Initialization**

Initialize BERT model and tokenizer.

**4. Preprocessing the Data**

Tokenize the input questions and context.

**5. Training the Model**

Fine-tune BERT on the SQuAD dataset.

**6. Inference**

Use the fine-tuned model to answer questions.

**Sample Input:** Question: "What is BERT?", Context: "BERT is a pre-trained transformer model for natural language understanding."

**Expected Output:** "A pre-trained transformer model for natural language understanding."

1. **QUESTION ANSWERING ON SQUAD DATASET USING TRANSFORMERS**

**Objective**

To build an advanced Q&A system using the Transformers library with a pre-trained model on the SQuAD dataset.

**Expected Outcomes**

Proficiency in using the Transformers library for state-of-the-art Q&A systems.

Understanding of fine-tuning pre-trained models on domain-specific datasets.

**Required Dataset**

The SQuAD dataset (or similar) for training and evaluation.

**How to Import the Dataset**

Utilize the `datasets` library to import the SQuAD dataset.

**Step-by-Step Instructions for Implementation**

**1. Environment Setup**

Install necessary libraries: `transformers`, `datasets`, `torch`.

**2. Data Loading**

Load the SQuAD dataset.

**3. Model and Tokenizer Initialization**

Use a pre-trained model like `distilbert-base-uncased` for efficiency.

**4. Data Preprocessing**

Tokenize and prepare inputs for the model.

**5. Fine-Tuning the Model**

Train the model using the Trainer API from the Transformers library.

**6. Inference**

Answer questions using the fine-tuned model.

**Sample Input:** Question: "Who developed BERT?", Context: "BERT was developed by researchers at Google."

**Expected Output:** "researchers at Google."

**WEEK 11: X. MACHINE TRANSLATION**

1. **BASIC MACHINE TRANSLATION USING RULE-BASED METHODS**

**Objective**

To build a simple rule-based machine translation system that translates sentences from English to a target language using predefined grammar rules and vocabulary.

**Expected Outcomes**

* Understand the basics of rule-based machine translation.
* Implement a simple translator for a specific language pair.
* Translate basic sentences using the developed system.

**Required Dataset**

A bilingual dictionary for the source and target languages.

Grammar rules for sentence structure in both languages.

**How to Import the Dataset**

For simplicity, we'll define a small bilingual dictionary and grammar rules directly in the code.

**Step-by-Step Instructions**

**1. Define the Bilingual Dictionary:**

*dictionary = {*

*'hello': 'bonjour',*

*'world': 'monde',*

*'my': 'mon',*

*'name': 'nom',*

*'is': 'est'*

*}*

**2. Define Grammar Rules:**

We can use simple rules like:

SVO (Subject-Verb-Object) structure: "My name is" becomes "Mon nom est".

*grammar\_rules = {*

*'SVO': ['subject', 'verb', 'object']*

*}*

**3. Translation Function:**

Implement a function to translate English sentences using the dictionary and grammar rules.

*def translate(sentence):*

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

*translated\_words = [dictionary.get(word, word) for word in words]*

*return ' '.join(translated\_words)*

*# Example usage:*

*sentence = "Hello world"*

*print(translate(sentence)) # Output: bonjour monde*

**Sample Input:** "Hello world"

**Expected Output:** "Bonjour monde"

1. **ENGLISH TO FRENCH TRANSLATION USING SEQ2SEQ WITH ATTENTION**

**Objective**

To develop an English-to-French translation model using the Seq2Seq architecture with an Attention mechanism.

**Expected Outcomes**

* Understand Seq2Seq with Attention mechanism.
* Train a neural network model for translation.
* Translate sentences from English to French.

**Required Dataset**

Dataset: English-French sentence pairs (e.g., WMT dataset).

**How to Import the Dataset**

Use TensorFlow Datasets (TFDS) or download and preprocess the dataset manually.

*import tensorflow\_datasets as tfds*

*dataset, metadata = tfds.load('ted\_hrlr\_translate/en\_to\_fr', with\_info=True, as\_supervised=True)*

*train\_examples, val\_examples = dataset['train'], dataset['validation']*

**Step-by-Step Instructions**

**1. Preprocessing:**

Tokenize the text data.

Create input and output sequences.

**2. Model Architecture:**

Define the Seq2Seq model with Attention.

**3. Training:**

Train the Seq2Seq model using the prepared dataset.

*# Training loop here (omitted for brevity)*

**Sample Input:** "How are you?"

**Expected Output:** "Comment ça va?"

1. **NEURAL MACHINE TRANSLATION WITH TRANSFORMERS (ENGLISH TO GERMAN)**

**Objective**

To implement a Neural Machine Translation system using the Transformer architecture for English-to-German translation.

**Expected Outcomes**

* Understand the Transformer architecture.
* Train a Transformer model for machine translation.
* Translate English sentences to German.

**Required Dataset**

Dataset: English-German sentence pairs (e.g., WMT dataset).

**How to Import the Dataset**

Use TensorFlow Datasets (TFDS) or download and preprocess the dataset manually.

*import tensorflow\_datasets as tfds*

*dataset, metadata = tfds.load('wmt14\_translate/de-en', with\_info=True, as\_supervised=True)*

*train\_examples, val\_examples = dataset['train'], dataset['validation']*

**Step-by-Step Instructions**

**1. Preprocessing:**

Tokenize the text data.

Create input and output sequences.

**2. Model Architecture:**

Define the Transformer model.

**3. Training:**

Train the Transformer model using the prepared dataset.

# Training loop here (omitted for brevity)

**Sample Input:** "Hello, how are you?"

**Expected Output:** "Hallo, wie geht es dir?"

**WEEK 12: XI.DIALOGUE SYSTEM**

1. **BASIC RULE-BASED CHATBOT USING PYTHON NLTK**

**Objective:**

To build a simple rule-based chatbot using the Natural Language Toolkit (NLTK) library in Python. The chatbot will respond to user inputs based on predefined rules and keywords.

**Expected Outcomes:**

* Understand the basics of chatbot development.
* Learn to use NLTK for text processing and keyword extraction.
* Create a basic rule-based chatbot capable of a simple conversation.

**Required Dataset:**

No specific dataset is required; predefined rules and responses will be used.

**Importing the Dataset:**

Not applicable as we are using predefined rules.

**Implementation Steps:**

**1. Install NLTK:**

*bash*

*pip install nltk*

**2. Import Libraries and Load Data:**

*import nltk*

*from nltk.chat.util import Chat, reflections*

*# Example rules*

*pairs = [*

*(r"my name is (.\*)", ["Hello %1, How are you today?"]),*

*(r"hi|hey|hello", ["Hello", "Hey there"]),*

*(r"what is your name?", ["I am a bot created by [Your Name]."]),*

*(r"how are you?", ["I'm doing good. How about you?"]),*

*(r"sorry (.\*)", ["No problem", "It's okay", "You don't need to be sorry"]),*

*(r"quit", ["Bye! Take care."]),*

*]*

**3. Create the Chatbot:**

*def chatbot():*

*print("Hi, I'm the chatbot you created. Type 'quit' to exit.")*

*chat = Chat(pairs, reflections)*

*chat.converse()*

*if \_\_name\_\_ == "\_\_main\_\_":*

*chatbot()*

**Sample Input:** "Hello"

**Sample Output:** "Hello"

**Sample Input:** "My name is John"

**Expected Output:** "Hello John, How are you today?"

1. **BUILDING A CHATBOT USING SEQ2SEQ MODELS**

**Objective:**

To create a more advanced chatbot using Sequence-to-Sequence (Seq2Seq) models. This model will use encoder-decoder architecture with LSTM or GRU units to generate responses.

**Expected Outcomes:**

* Learn the architecture and implementation of Seq2Seq models.
* Develop a chatbot capable of generating more natural and context-aware responses.

**Required Dataset:**

Dialogue datasets like Cornell Movie-Dialogs Corpus.

**Importing the Dataset:**

Download and extract the dataset, then preprocess it to create input-output pairs for training.

**Implementation Steps:**

**1. Install Required Libraries:**

*bash*

*pip install tensorflow keras*

**2. Data Preprocessing:**

*# Load and preprocess the dataset (example code)*

*# Load the data from file, clean and preprocess it*

**3. Model Building:**

**4. Training the Model:**

**5. Generating Responses:**

*# Convert input\_text to sequence, pad it, and generate a response*

*pass*

**Sample Input:** "How are you?"

**Expected Output:** "I'm good, thank you!"

**C. CONVERSATIONAL AI WITH TRANSFORMER-BASED MODELS**

**Objective:**

To build an advanced conversational AI using Transformer-based models, such as GPT-3 or BERT, capable of understanding and generating human-like text.

**Expected Outcomes:**

* Understand the transformer architecture and its application in conversational AI.
* Create a chatbot with sophisticated conversational abilities.

**Required Dataset:**

Large conversational datasets like OpenSubtitles, Reddit conversations, or custom datasets.

**Importing the Dataset:**

Download and preprocess the dataset, focusing on creating meaningful input-output pairs.

**Implementation Steps:**

**1. Install Required Libraries:**

*bash*

*pip install transformers torch*

**2. Data Preprocessing:**

**3. Model Fine-Tuning:**

**4. Generating Responses:**

**Sample Input:** "What's the weather like today?"

**Expected Output:** "It's a sunny day, perfect for a walk!"